

Vademecum informatyki praktycznej

(aktualizacja: 2024-09-25)

W założeniu ma to być kompleksowy przegląd podstawowych zagadnień z zakresu elektroniki, programowania oraz użytkowania i administracji systemami unixowymi. Nie jest to tutorial, ani samouczek prowadzący krok po kroku w zgłębianie tajników "komputerologii", natomiast tekst ten może być wykorzystywany jako materiał pomocniczy do zająć o takiej tematyce, bądź też materiał służący przypomnieniu / uzupełnieniu posiadanej wiedzy na przedstawiane tematy. Korzystanie z tego dokumentu przy samodzielnej nauce jest także możliwe, należy jednak liczyć się z tym że konieczne może być sięganie do innych, bardziej szczegółowych źródeł, a co najważniejsze także duża ilość praktyki i prób samodzielnego rozwiązywania problemów - samo przeczytanie nie wystarczy.

Założeniem tego dokumentu było że ma być kompletny i zamknięty, jednym z efektów tego jest unikanie odsyłania do innych materiałów bezpośrednio w tekście, wszystkie odnośniki do zewnętrznych źródeł (za wyjątkiem powoływania się na dokumentację narzędzi/bibliotek) znajdują się wyłącznie w dziale "Literatura". Założeniem tego dokumentu było także że ma być w miarę uniwersalny i ponadczasowy (nie zdezaktualizować się po roku) dlatego też skupia się na bardziej uniwersalnych aspektach omawianych zagadnień, faktach, definicjach, itp bez odniesień w stylu "obecnie już", "nadal jest używany", itd, nie pokazuje też różnego rodzaju sztuczek czy obejść doraźnych problemów, ani nie omawia szczegółów technologii szybko zmieniających się.

Elektronika

Elektronika zajmuje się wytwarzaniem i przetwarzaniem sygnałów w postaci prądów i napięć elektrycznych. Zjawisko prądu związane jest z przepływem ładunku (z uporządkowanym ruchem nośników ładunku), aby wystąpiło konieczna jest różnica potencjałów (napięcie) pomiędzy końcami przewodnika, prowadzi ono do neutralizacji tej różnicy. Dlatego dla podtrzymania stałej różnicy potencjałów konieczne jest istnienie źródeł prądu, prowadzących do rozdzielania ładunków dodatnich od ujemnych.

Podstawy

Napięcie i prąd

Dla elementów liniowych (np. zwykły kawałek przewodu) zachodzi proporcjonalność natężenia prądu płynącego przez taki element do napięcia pomiędzy jego końcami: $R = \frac{U}{I}$.

Natężenie prądu elektrycznego I (określane skrótowo jako **prąd**)

jest to stosunek przemieszczonego ładunku do czasu jego przepływu.

Napięcie elektryczne U pomiędzy punktem A i B (jakiegoś obwodu)

jest to różnica potencjału elektrycznego w punkcie A i w punkcie B.

Potencjał elektryczny V w punkcie A

jest skalarną wielkością charakteryzującą pole elektryczne w danym punkcie. Odpowiada pracy którą trzeba by wykonać aby przenieść ładunek q z tego punktu do nieskończoności podzielonej przez wielkość tego ładunku (jest niezależny od wartości q).

W elektronice używa się wartości potencjałów względem umownego potencjału zerowego GND (co umożliwia traktowanie ich jako różnic potencjałów - napięć elektrycznych), w efekcie tego określenia "(stałe) napięcie" i "potencjał" bardzo często stosowane są zamiennie.

Masa, GND

umowny potencjał zerowy, względem którego wyraża się inne potencjały w układzie (co umożliwia traktowanie ich jako różnic potencjałów - napięć elektrycznych). Potencjał ten może być równy potencjałowi ziemi (masie ochronnej PE), bądź może być z nim nie związany (układy izolowane).

Podstawowym przykładem teoretycznych (w rzeczywistości mogą występować tylko elementy realizujące ich funkcję w pewnym zakresie i w pewnym przybliżeniu) elementów nieliniowych są:

Idealne źródło prądowe

wymusza przepływ określonego prądu przez przyłączony układ, niezależnie od spadków napięć w układzie (czyli może odłożyć się na nim dowolne napięcie).

Idealne źródło napięciowe

wymusza określoną różnicę potencjałów pomiędzy swoimi zaciskami (czyli może przez nie przepływać dowolne natężenie prądu).

Węzeł układu (sam w sobie, pomijając zjawiska pasożytnicze) nie jest w stanie gromadzić ładunku elektrycznego zatem: *Suma prądów wpływających do węzła jest równa sumie prądów wypływających z tego węzła.*

Jeżeli rozważamy obwód zamknięty od punktu A z potencjałem V_A to sumując napięcia na kolejnych elementach obwodu (oporach, źródłach napięciowych, etc) z uwzględnieniem ich znaku gdy wrócimy

do punktu A to potencjał nadal musi wynosić V_A , zatem: *Suma spadków napięć w zamkniętym obwodzie jest równa zeru.*

Opór, pojemność i indukcyjność

Pojęcia

Rezystancja (opór elektryczny)

wyraża trudność przepływu prądu przez dany przewodnik - im większy opór elementu to (przy takim samym przyłożonym napięciu) będzie płynął mniejszy prąd. $R = \frac{U}{I}$

Pojemność (wzajemna)

wyraża zdolność do gromadzenia ładunku przez dany element - im większa pojemność tym więcej ładunku (przy takim samym przyłożonym napięciu) zgromadzi element. $C = \frac{q}{U}$

Indukcyjność

wyraża zdolność do wytwarzania strumienia pola magnetycznego związanego z przepływem prądu przez dany element. $L = \frac{\Phi}{I}$

Obwody prądu stałego a zmiennego

Pojemność oraz indukcyjność wprowadzona do obwodu ma znaczenie tylko gdy zachodzi zmiana natężenia prądu płynącego przez obwód / zmiana wartości napięć odłożonych na jego elementach. W stanie ustalonym obwodu prądu stałego wprowadzona do obwodu pojemność oraz indukcyjność nie odgrywają roli, gdyż:

- pojemności zgromadziły już (stosowny do przyłożonego do nich napięcia) ładunek i nie pobierają prądu z obwodu,
- indukcyjności wytworzyły pole magnetyczne (stosowne do przepływającego przez nie prądu) i nie stanowią oporu dla płynącego prądu w obwodzie.

W takim przypadku można traktować pojemności jako rozwarcia, a indukcyjności jako zwarcia.

Impedancja

jest wielkością charakteryzującą zależność pomiędzy natężeniem prądu i napięciem uwzględniającą reaktancję obwodu. $Z = R + jX$

Reaktancja

jest wielkością charakteryzującą opór bierny elementów pojemnościowych (*kapacytancja* $X_C = -\frac{1}{\omega C}$) i indukcyjnych (*induktancja* $X_L = \omega L$).

Pulsacja ω

charakteryzuje szybkość zmian (jest proporcjonalna do odwrotności czasu trwania okresu zmiany). $\omega = \frac{2\pi}{T}$

Elementy rzeczywiste

Rezystor (opornik)

wprowadza do układu rezystancję związaną z swoją wartością nominalną. Typowo służy do ograniczania wartości prądu przez niego przepływającego. Symbole: , .

Powoduje wydzielanie się energii (cieplnej) związanej z stratami na rezystancji - moc wydzielana dana jest zależnościami: $P = UI = \frac{U^2}{R} = I^2 R$, czyli przy stałym napięciu przyłożonym do rezystora im większy jego opór tym mniejsza moc się wydzieli (gdyż popłynie mniejszy prąd), ale przy stałym prądzie płynącym przez rezystor moc rośnie wraz ze wzrostem oporu.

Kondensator

wprowadza do układu pojemność związaną z swoją wartością nominalną. Typowo służy do ograniczania zmian napięcia (poprzez gromadzenie energii w polu elektrycznym). Symbole: , . Czas potrzebny do zmiany napięcia na kondensatorze dany jest zależnością: $\Delta T = \frac{C \cdot \Delta U}{I}$

Cewka (dławik)

wprowadza do układu indukcyjność związaną z swoją wartością nominalną. Typowo służy do ograniczania zmian prądu (poprzez gromadzenie energii w polu magnetycznym). Symbol: . Czas potrzebny zmiany prądu płynącego przez cewkę (dławik stawia opór takiej zmianie tak jak kondensator zmianie napięcia) dany jest zależnością: $\Delta T = \frac{L \cdot \Delta I}{U}$.

Często cewka nawinięta jest na rdzeń ferrytowy, bardzo silnie zwiększa on indukcyjność cewki, ale ogranicza od góry częstotliwość pracy cewki. Ponadto przy odpowiednio dużym prądzie może dochodzić do nasycenia rdzenia, które objawia się tym, że zaczyna in hamować wzrost pola zamiast go poprawiać. Cewki możemy spotkać w urządzeniach takich jak przekładniki, czy styczniki. Nawinięte na odpowiednim rdzeniu pełnią one tam funkcję elektromagnesu

odpowiedzialnego za zmianę fizycznej pozycji styków prowadzącą do ich zwarcia lub rozwarcia (przełączania).

Rozłączanie cewki: Jako że cewka jest elementem który dąży do zachowania płynącego przez niego prądu, to w przypadku rozwarcia obwodu zawierającego cewkę napięcie na niej będzie rosło i bez problemów może wielokrotnie przekroczyć napięcie zasilania. Zjawisko to bywa użyteczne i jest wykorzystywane w niektórych układach (np. przetwornicach podnoszących napięcie), ale często bywa też niepożądane, a nawet bardzo szkodliwe - może prowadzić do uszkodzenia innych elementów w obwodzie (w szczególności elementu przełączającego). Aby mu przeciwdziałać stosuje się połączoną równolegle z cewką diodę, spolaryzowaną w taki sposób aby w normalnym stanie nie przewodziła, a po odłączeniu zasilania cewki pozwalała na jej rozładowanie.

Transformator

Układ cewek sprzężonych magnetycznie, z których część służy do wytwarzania energii pola magnetycznego, a część do jej odbierania. Typowo służy do przekazywania energii poprzez pole magnetyczne celem zmiany napięcia lub separacji galwanicznej obwodów.

W przypadku dwu uzwojeniowego transformatora zachodzi: $\frac{U_{we}}{U_{wy}} = \frac{I_{wy}}{I_{we}} = \frac{n_{we}}{n_{wy}} = z$, gdzie z to przekładnia transformatora, n_{we} to liczba uzwojeń strony pierwotnej (wejściowej), a n_{wy} to liczba uzwojeń strony wtórnej (wyjściowej).

Styk

jest to element służący do mechanicznego załączania lub odłączania prądu, nazywany bywa także przełącznikiem. Wyróżnia się styki monostabilne (chwilowe), które samoistnie (po ustaniu czynnika przełączającego) wracają do jednej ze swoich pozycji oraz styki bistabilne, które po przełączeniu pozostają w wybranej pozycji. Wyróżnia się także styki zwierne (normalnie rozwarne), rozwierne (normalnie zwarte) lub przełączne (następuje przełączanie pomiędzy dwoma lub więcej możliwymi wyjściami). Czynnikiem odpowiedzialnym za przełączanie styków może być manualne działanie (przełącznik naciskany, przekręcany, pociągany, itd przez człowieka) lub działanie elektromagnesu (cewki) - jest to wtedy przekaźnik elektromagnetyczny lub stycznik.

Stan zwarcia określa się często mianem "zamknięty" (ang. "close"), a rozwarcia mianem "otwarty" ("open"). Należy (zwłaszcza w systemach cyfrowych) pamiętać, że styki potrafią drgać - na ogół przy zmianie pozycji zamiast pojedynczego impulsu generują całą serię.

Elementy rzeczywiste oprócz wartości nominalnej (wprowadzanej przez nie rezystancji, pojemności lub indukcyjności) charakteryzują także:

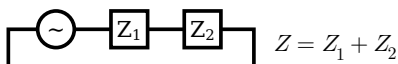
- wartości graniczne (takie jak maksymalna moc która może wydzielić się na rezystorze, napięcie przebicia kondensatora, maksymalny prąd przewodzenia cewki, temperaturowy zakres pracy, ...),
- impedancje pasożytnicze (każdy element rzeczywisty wprowadza zarówno niepożądaną rezystancję, jak i niepożądaną reaktancję nie związaną z jego wartością nominalną).

W większości przypadków impedancję pasożytniczą można pominąć. Jednak np. w przypadku rzeczywistych źródeł napięciowych i prądowych rezystancja wewnętrzna takiego źródła często odgrywa istotną rolę dla działania układu i nie może zostać pominięta.

Układy połączeń

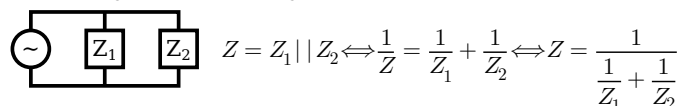
Połączenie szeregowe

jest to takie połączenie elementów że przepływa przez nie jednakowy prąd. Impedancje (zarówno rezystancyjne jak i reaktancyjne) tak połączonych elementów sumują się.



Połączenie równoległe

jest to takie połączenie elementów że przyłożone do nich napięcia są jednakowe (a prąd płynący w obwodzie ulega rozdzieleniu). Odwrotność impedancji takiego układu jest sumą odwrotności impedancji tak połączonych elementów.

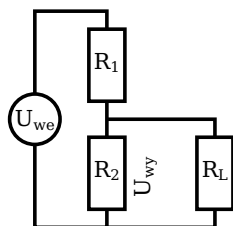


Dzielnik napięcia (rezystancyjny)

jest to szeregowe połączenie dwóch rezystorów. Poprzez dobór wartości rezystorów uzyskuje się pożądaną napięcie na wyjściu takiego dzielnika, które dane jest zależnością $U_{wy} = U_{we} \frac{R_2}{R_1 + R_2}$. Ze

względu na to iż jest ono proporcjonalne do napięcia przyłożonego do całego dzielnika może on służyć także do skalowania napięciowego sygnału wejściowego. Aby minimalizować wpływ wartości impedancji podłączonej do wyjścia dzielnika na napięcie wyjściowe należy zapewnić odpowiednio

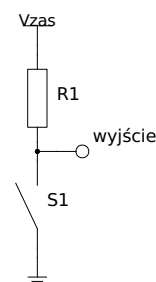
małe wartości rezystorów w dzielniku - tak aby prąd płynący przez R_2 był zdecydowanie większy (typowo 5 do 10 razy) od prądu płynącego przez obciążenie (R_L).



Rezystor podciągający

jest bardzo często używany w celu wymuszenia domyślnego poziomu napięcia na jakiejś linii. Jest to zasadniczo forma dzielnika napięcia w którym jeden z rezystorów został zastąpiony jakimś rodzaju przełącznikiem, czyli czymś co w zależności od swojego stanu ma prawie zerową albo (prawie) nieskończoną rezystancję.

Rozwiązanie takie ma zastosowanie głównie na jakichś liniach sygnalizacyjnych, z których nie jest pobierany żaden większy prąd. W efekcie, w układzie pokazanym obok jeżeli styk jest rozarty to prąd nie płynie, zatem spadek na rezystorze wynosi zero i na wyjściu mamy napięcie zasilania. Jeżeli styk zostanie zwarty prąd płynie, ale ze względu na małą rezystancję styku praktycznie całe napięcie odkłada się na rezystorze i na wyjściu mamy zero voltów.



Układ taki pozwala na przykład stosowanie zwykłego styku zwierzanego zamiast przełączalnego. Oczywiście możemy zamienić rezystor z przełącznikiem miejscami i wtedy domyślnym stanem (przy rozwartym styku) będzie zero voltów.

Dzielnik prądu (rezystancyjny)

jest to równoległe połączenie dwóch rezystorów. Poprzez dobór wartości rezystorów uzyskuje się pożądane natężenie prądu w wyjściowej gałęzi.

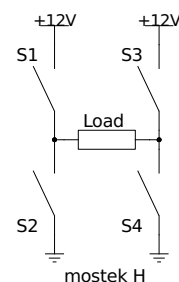
Obwód RC

jest to szeregowe połączenie rezystora i kondensatora. Obwód taki charakteryzuje się stałą czasową $\tau = RC$. Napięcie na kondensatorze w trakcie jego ładowania dane jest zależnością

$U_C(t) = U_{we} \cdot (1 - e^{-\frac{t}{\tau}})$, a w trakcie rozładowywania $U_C(t) = U_C(0) \cdot (e^{-\frac{t}{\tau}})$. W przypadku wyjścia na zaciskach rezystora stanowi filtr górnoprzepustowy, a w przypadku wyjścia na zaciskach kondensatora stanowi filtr dolnoprzepustowy, częstotliwość graniczna (zapewniająca tłumienie większe niż 3dB) takich filtrów wynosi $f_{gr} = \frac{1}{2\pi\tau}$.

Mostek H

jest to układ (oparty o 4 przełączniki, których rolę mogą pełnić klucze tranzystorowe) pozwalający na zmianę polaryzacji zasilania podłączonego do niego odbiornika. Układ taki złożony jest z dwóch identycznych gałęzi (S1 + S2 oraz S3 + S4, każda włączona pomiędzy dwoma biegunami zasilania). Pojedyncza taka gałąź nazywana jest pół-mostkiem i składa się z dwóch kluczy które powinny być sterowane przeciwstawnie (aby wyeliminować możliwość zwarcia zasilania z masą). Układ pół-mostka może być wykorzystywany także samodzielnie jako uniwersalny układ klucza pozwalającego na załączanie odbiornika zarówno od strony napięcia dodatniego jak i od strony masy (w zależności od sposobu jego podłączenia) lub przełączaniem dwóch odbiorników (jednego umieszczonego pomiędzy zasilaniem a wyjściem mostka, a drugiego pomiędzy wyjściem a masą). Rolę kluczy (przełączników) w ramach mostka mogą pełnić także tranzystory PNP (jako S1, S3) i NPN (jako S2, S4) albo analogicznie tranzystory P-MOSFET i N-MOSFET.



Dioda

Dioda idealna to element przewodzący prąd tylko w jednym kierunku. Rzeczywiste diody przewodzą prąd zdecydowanie chętniej w jednym kierunku niż w drugim (na ogół przewodzenie w kierunku zaporowym się pomija) ponadto charakteryzują je cechy zależne od technologii wykonania takie jak:

- spadek napięcia w kierunku przewodzenia (typowo dla diod  dioda  dioda Zenera krzemowych 0.6V - 0.7V, a dla diod Schottky'ego 0.3V)
- napięcie przebicia - napięcie, które przyłożone w kierunku zaporowym powoduje znaczące przewodzenie diody w tym kierunku - w większości przypadków parametr którego nie należy przekraczać, jednak wykorzystywane (i stanowiące ich parametr) w niektórych typach diod

dioda

LED

dioda Zenera

- maksymalny prąd przewodzenia
- czas przełączania (związany głównie z pasożytniczą pojemnością złącza) - zdecydowanie krótszy (około 100 ps) w diodach Schottky'ego niż w diodach krzemowych,.

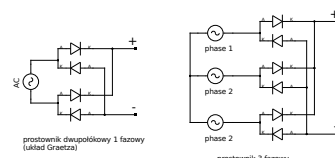
Ponadto stosowane są m.in.:

- diody Zenera - wykorzystuje się (charakterystyczną dla danego typu) wartość napięcia przebicia do uzyskania w układzie spadku napięcia o tej wartości,
- diody świecące (LED) - emitujące światło w trakcie przewodzenia (na elemencie występuje stały spadek napięcia, jasność zależy od natężenia prądu),
- fotodiody - będące detektorami oświetlenia (przewodzenie spolaryzowanej w kierunku zaporowym zależy od ilości padającego na element światła, niespolaryzowana pod wpływem oświetlenia staje się źródłem prądu).

prostownik

Jednym z zastosowań diod jest prostowanie prądów przemiennych (zmieniającego znak). Układy to realizujące określa się mianem prostowników.

Funkcję tą może pełnić nawet pojedyncza dioda - mamy wtedy do czynienia z prostownikiem jednopółkowy, charakteryzującym się tym że napięcie na jego wyjściu spada przez połowę okresu wynosi zero.



Lepszym i częściej stosowanym rozwiązaniem jest prostownik pełnookresowy (dwupółkowy). Najczęstszą jego realizacją jest tzw. mostek Graetza, czyli układ 4 diod połączonych w taki sposób iż dwie z nich zawsze (w każdym punkcie napięcia wejściowego) przewodzą. Wadą takiego układu jest znaczny spadek napięcia na mostku, wynoszący dwukrotność spadku napięcia na pojedynczej diodzie.

Istnieją również układy prostowników napięcia trójfazowego, charakteryzują się one m.in. niższymi tętnieniami napięcia wyjściowego - dla prostowników jednofazowych waha się ono (pomijając spadki na diodach) od 0 do $V_{LN}\sqrt{2}$, a dla pełnookresowego trójfazowego od $V_{LL}\sqrt{2}\sin 60$ do $V_{LL}\sqrt{2}$ (gdzie V_{LN} to napięcie skuteczne pomiędzy fazą a przewodem neutralnym, a $V_{LL} = V_{LN}\sqrt{3}$ to napięcie skuteczne międzyfazowe).

dzielnik napięcia z diodą Zenera

Wadą rezystancyjnego dzielnik napięcia złożonego z dwóch rezystorów była duża zależność napięcia wyjściowego od obciążenia. Zjawisko to można ograniczyć zastępując jeden z rezystorów (ten równolegle połączony z obciążeniem) diodą Zenera w polaryzacji zaporowej, która charakteryzuje się dość stałym spadkiem napięcia. Nadal nie jest to rozwiązanie idealne (ze względu na zależność napięcia przebicia diody Zenera od prądu), ale znacznie bardziej stabilne od poprzedniego.

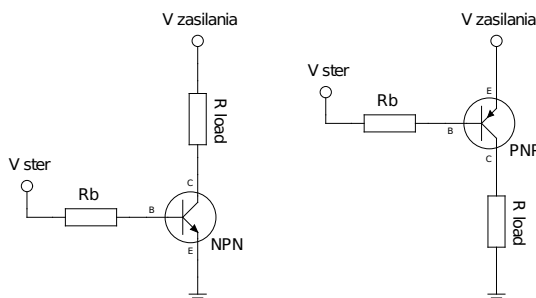
Tranzystor

Tranzystor jest to element o regulowanym elektrycznie przewodzeniu prądu (oporze), często wykorzystywany do wzmacniania sygnałów lub jako przełącznik elektroniczny.

NPN, PNP, JFET, MOSFET, ...

NPN

Prąd przepływający pomiędzy kolektorem a emiterem jest funkcją prądu przepływającego pomiędzy bazą a emiterem: $I_C = \beta I_B$. Napięcie pomiędzy kolektorem a emiterem wynosi: $U_{CE} = U_{zasilania} - I_C \cdot R_{load}$. Napięcie to nie może jednak spaść poniżej wartości minimalnej wynoszącej około 0.2V, gdy z powyższych zależności wynikałby taki spadek to tranzystor pracuje w stanie nasycenia i $U_{CE} \approx 0.2V$.



PNP

Podobnie jak w NPN tyle że prąd przepływający pomiędzy emiterem a kolektorem jest funkcją prądu przepływającego pomiędzy emiterem a bazą.

Spadek napięcia pomiędzy bazą a emiterem U_{BE} w tranzystorze NPN (między emiterem a bazą w PNP) wynosi nie więcej niż około 0.7V. Utrzymanie takiej wartości spadku napięcia powoduje powstanie określonego prądu bazy. Jeżeli spadek ten wynosi dużo mniej (np. zero), to prąd bazy nie płynie, a więc tranzystor nie przewodzi.

Zatem w układzie przedstawionym obok zmiana V_{ster} spowoduje taką zmianę prądu bazy I_B aby na R_b odłożyło się napięcie $V_{ster} - 0.7V$. Z tego też względu konieczne jest stosowanie rezystancji pomiędzy

bazą a napięciami sterującymi innymi od napięcia emitera.

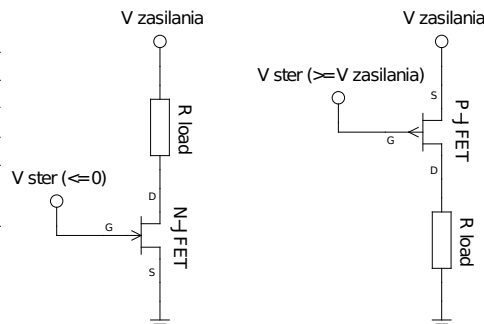
N-JFET

Prąd przepływający pomiędzy drenem (*drain*) a źródłem (*source*) jest funkcją napięcia pomiędzy bramką (*gate*) a źródłem (potencjału bramki względem źródła - U_{GS}), prąd bramki można uznać za pomijalnie mały. Tranzystor ten przewodzi tylko gdy napięcie pomiędzy bramką a źródłem jest większe od napięcia odcięcia ($U_{GS_{off}} < 0$), przewodzony prąd wzrasta wraz ze wzrostem U_{GS} , a maksimum jest osiągnięte dla $U_{GS} = 0$.

Gdy tranzystor jest spolaryzowany tak aby $U_{DS} > 0$ oraz $U_{GS} > U_{GS_{off}}$ to w zależności od wartości spadku napięcia na tranzystorze U_{DS} oraz wartości napięcia nasycenia $U_{sat} = U_{GS} - U_{GS_{off}}$ wyróżnia się dwa tryby pracy:

1. liniowy (gdy $U_{DS} > U_{sat}$), w którym $I_D = \beta \cdot U_{sat}^2$
2. nasycenia (gdy $U_{DS} \leq U_{sat}$), w którym $I_D = \beta \cdot U_{GS} \cdot [2U_{sat} - U_{DS}]$

gdzie $\beta = \frac{I_{DSS}}{U_{GS_{off}}^2}$



P-JFET

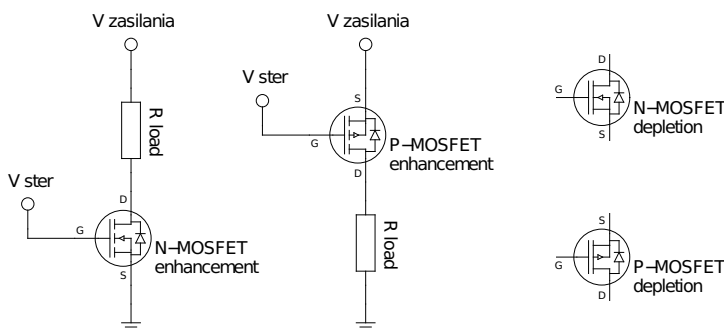
Podobnie jak N-JFET tyle że: prąd przepływa pomiędzy źródłem a drenem, przewodzony prąd wzrasta wraz ze zmniejszaniem U_{GS} (od $U_{GS_{off}} > 0$ do zera).

N-MOSFET

Prąd przepływający pomiędzy drenem (*drain*) a źródłem (*source*) jest funkcją napięcia pomiędzy bramką (*gate*) a źródłem (potencjału bramki względem źródła - U_{GS}), bramka jest izolowana (nie płynie przez nią prąd).

W kierunku dren $\rightarrow$ źródło tranzystor ten przewodzi gdy $U_{GS} > U_{GS(th)}$, natomiast w przeciwnym kierunku przewodzi zawsze. Dla tranzystorów N-MOSFET z kanałem wzbogacanym (*enhancement*) $U_{GS(th)} > 0$, a z kanałem zubożonym (*depletion*) $U_{GS(th)} < 0$.

Konkretna wartość $U_{GS(th)}$ zależna jest od konkretnego modelu tranzystora, innym istotnym parametrem związanym z sterowaniem tranzystorem jest maksymalna i minimalna dopuszczalna wartość napięcia U_{GS} .



P-MOSFET

Podobnie jak N-MOSFET tyle że:

1. regulowane jest przewodzenie w kierunku źródło $\rightarrow$ dren (a w kierunku dren $\rightarrow$ źródło przewodzi zawsze),
2. przewodzenie w kierunku źródło $\rightarrow$ dren ma miejsce gdy $U_{GS} < U_{GS(th)}$,
3. dla tranzystorów z kanałem wzbogacanym (*enhancement*) $U_{GS(th)} < 0$, a z kanałem zubożonym (*depletion*) $U_{GS(th)} > 0$.

Przykłady: BC846 (NPN, 65V, 0.1A), BC337 (NPN, 50V, 0.8A), BD139 (NPN, 80V, 1.5A); BC327 (PNP, 50V, 0.8A), BD140 (PNP, 80V, 1.5A); BS170 (N-MOSFET, 60V, 50mOhm, 0.5A), IRF540N (N-MOSFET, 100V, 44mOhm, 33A), IRFZ44N (N-MOSFET, 55V, 17.5mOhm, 49A); IRF9Z34N (P-MOSFET, 55V, 100mOhm, 19A), IRF5210 (P-MOSFET, 100V, 60mOhm, 40A), IRF5305 (P-MOSFET, 55V, 60mOhm, 31A); BF245 (N-JFET) i wiele innych

(inne) elementy mocy

Tyristor

Jest elementem przewodzącym wyłącznie w jednym kierunku (od anody do katody). Przewodzenie musi zostać zainicjowane podaniem (dodatniego względem katody) napięcia na bramkę. Wyzwolony (raz przyłożonym napięciem bramki) tyristor przewodzi do momentu spadku prądu poniżej minimalnej wartości przewodzenia lub wystąpienia odwrotnej polaryzacji. Zasadę działania tyristora obrazuje przedstawiony obok dwu-tranzystorowy model.

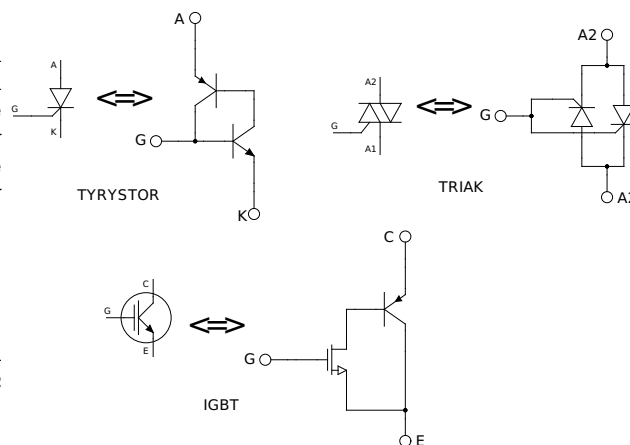
Triak

Jest elementem o dwukierunkowym przewodzeniu inicjowanym impulsem napięciowym podanym na bramkę. Zasadę jego działania obrazuje schemat zastępczy złożony z dwóch połączonych anty-równolegle tyrystorów. Używany jest jako elektroniczny przełącznik prądu przemiennego.

Przykłady: BTA16

Tranzystor IGBT

Jest to tranzystor bipolarny z izolowaną bramką. Z punktu sterowania należy patrzeć na niego jak na tranzystor typu N-MOSFET. Stosowany do przełączania w układach dużej mocy.



transoptory

Poprzez połączenie w jednej obudowie diody świecącej z fototranzystorem, fotodiodą (niekiedy wraz z tranzystorem do bramki którego jest podłączona), fototriakiem lub podobnym elementem (to znaczy takim którego przewodność zależy od padającego na niego oświetlenia) uzyskuje się element nazywany transoptorem. W układzie może być zawarty także dodatkowy stopień wyjściowy dostosowujący napięcia do poziomów wymaganych przez jakiś standard.

Transoptor zapewnia on izolację (typowo rzędu tysięcy woltów) galwaniczną dwóch części obwodu. Pozwala to m.in. na odbiór sygnałów wysokonapięciowych przez niskonapięciową część układu (dioda LED może być sterowana np. z napięcia 230V, a tranzystor może sterować układem 3.3V).

Ponadto transoptor zabezpiecza część odbiorczą (podłączoną do swojego wyjścia) przed przepięciami w części sterującej diodą LED. Jednak ze względu na fizyczną bliskość diody i elementu fotoelektrycznego przy napięciu przekraczającym wartość izolacji może zdarzyć się przebicie (pełną izolację optyczną zapewnia użycie połączeń światłowodowych).

Przykłady: 4N25 (analogowy), 4N35 (analogowy), 6N136 (cyfrowy), 6N137 (cyfrowy), MOC3023 (optotriak), MOC3063 (optotriak)

Klucz tranzystorowy

Klucz jest układem przełączającym wykorzystującym dwa skrajne stany pracy tranzystora - zatkania (tranzystor nie przewodzi), nasycenia (tranzystor przewodzi z minimalnymi ograniczeniami).

klucz NPN

Aby wprowadzić tranzystor NPN w stan zatkania należy podać na jego bazę potencjał mniejszy lub równy potencjałowi emitera (zakładamy że potencjał kolektora jest nie mniejszy niż emitera - co ma miejsce w typowych warunkach polaryzacji tranzystora NPN), czyli $U_{BE} \leq 0$.

Aby wprowadzić tranzystor NPN w stan nasycenia należy na jego bazę wprowadzić potencjał większy od potencjałów emitera i kolektora, uzyskuje się to poprzez wprowadzenie do tranzystora prądu bazy

$$I_B \gg \frac{U_{zasilania}}{\beta R_{load}}$$

klucz PNP

Aby wprowadzić tranzystor PNP w stan zatkania należy podać na jego bazę potencjał większy lub równy potencjałowi emitera (zakładamy że potencjał emitera jest nie mniejszy niż kolektora - co ma miejsce w typowych warunkach polaryzacji tranzystora PNP), czyli $U_{BE} \geq 0$.

Aby wprowadzić tranzystor PNP w stan nasycenia należy na jego bazę wprowadzić potencjał mniejszy od potencjałów emitera i kolektora, uzyskuje się to poprzez wyprowadzenie z tranzystora prądu bazy

$$I_B \gg \frac{U_{zasilania}}{\beta R_{load}}$$

klucz JFET

Aby wprowadzić w stan zatkania tranzystor N-JFET należy podać $U_{GS} \leq U_{GS_{off}}$, a dla tranzystora P-JFET należy podać $U_{GS} \geq U_{GS_{off}}$.

Aby wprowadzić w stan przewodzenia tranzystor N-JFET należy podać $U_{GS} \geq 0$, a dla tranzystora P-JFET należy podać $U_{GS} \leq 0$.

klucz N-MOSFET

Aby wprowadzić tranzystor N-MOSFET w stan zatkania należy podać $U_{GS} < U_{GS(th)}$. Dla tranzystorów:

- z kanałem wzbogaconym wystarczy obniżyć potencjał bramki do potencjału źródła (lub nawet wartości niewiele wyższej od niego)
- z kanałem zubożonym musi to być wartość poniżej potencjału źródła.

Aby wprowadzić tranzystor N-MOSFET w stan przewodzenia należy podać $U_{GS} \gg U_{GS(th)}$.

klucz P-MOSFET

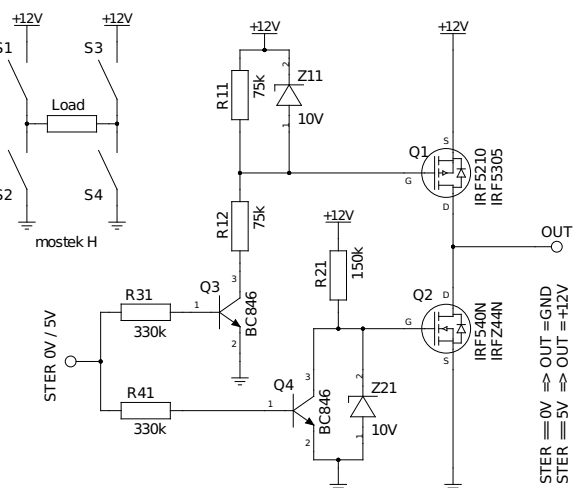
Aby wprowadzić tranzystor P-MOSFET w stan zatkania należy podać $U_{GS} > U_{GS(th)}$. Dla tranzystorów:

- z kanałem wzbogaconym wystarczy podnieść potencjał bramki do wartości potencjału źródła (lub nawet wartości niewiele niższej od niego)
- z kanałem zubożonym musi to być wartość powyżej potencjału źródła.

Aby wprowadzić tranzystor P-MOSFET w stan przewodzenia należy podać $U_{GS} \ll U_{GS(th)}$.

(pół)mostek H

Mostek H jest to układ (oparty o 4 przełączniki, których rolę mogą pełnić klucze tranzystorowe) pozwalający na zmianę polaryzacji zasilania podłączonego do niego odbiornika. Układ taki złożony jest z dwóch identycznych gałęzi (S1 + S2 oraz S3 + S4, każda włączona pomiędzy dwoma biegunami zasilania). Pojedyncza taka gałąź nazywana jest pół-mostkiem i składa się z dwóch kluczy które powinny być sterowane przeciwnie (aby wyeliminować możliwość zwarcia zasilania z masą). Układ pół-mostka może być wykorzystywany także samodzielnie jako uniwersalny układ klucza pozwalającego na załączanie odbiornika zarówno od strony napięcia dodatniego jak i od strony masy (w zależności od sposobu jego podłączenia) lub przełączania dwóch odbiorników (jednego umieszczonego pomiędzy zasilaniem a wyjściem mostka, a drugiego pomiędzy wyjściem a masą).



Na schemacie umieszczonym obok przedstawiona jest także przykładowa realizacja pół-mostka w oparciu o dwa klucze na tranzystorach MOSFET i sterujące nimi klucze NPN.

Zastosowanie kluczy NPN (tranzystory Q3 i Q4) pozwala na sterowanie z użyciem napięcia niższego od napięcia przełączanego.

W stanie zatkania Q3 i Q4:

- potencjał bramki Q1 równy jest potencjałowi zasilania (czyli także potencjałowi źródła), zatem Q1 (P-MOSFET wzbogacony) nie przewodzi;
- potencjał bramki Q2 ustala dzielnik złożony z R21 i Z21 (jest on wyższy od potencjału źródła, minimalnie o wartość napięcia zasilania, a maksymalnie o wartość spadku na Z21) i jest on wyższy o co najmniej $U_{GS(th)}$ od potencjału źródła, zatem Q2 (N-MOSFET wzbogacony) przewodzi.

W stanie nasycenia (przewodzenia) Q3 i Q4:

- potencjał bramki Q1 ustalany jest przez dzielnik R11 i R12 albo (jeżeli napięcie zasilania jest odpowiednio wysokie) przez dzielnik Z11 i R12 i jest on niższy o co najmniej $U_{GS(th)}$ od potencjału źródła, zatem Q1 przewodzi;
- potencjał bramki Q2 jest równy potencjałowi masy (czyli także potencjałowi źródła), zatem Q2 nie przewodzi.

Zastosowanie diod Zenera Z1 i Z2 zabezpiecza tranzystory MOSFET przed podaniem na bramkę zbyt dużego (co do wartości bezwzględnej) napięcia w stosunku co do źródła. Niestety wiąże się to także z zastosowaniem to zastosowaniem rezystora R12, który (tworząc dzielnik) powoduje iż układ nie będzie działał poprawnie dla niskich napięć zasilania toru prądowego - napięcie musi być większe od $2U_{GS(th)}$ zamiast od $U_{GS(th)}$. Ponadto wymaga to zastosowania osobnych gałęzi sterujących dla poszczególnych tranzystorów MOSFET (dwóch kluczy NPN), ponieważ obecność Z21 w gałęzi sterującej P-MOSFETem (Q1) przy wyższym napięciu zasilania powodowała by przepływ prądu przez tą gałąź niezależnie od Q3 co mogłoby prowadzić do przewodzenia Q1.

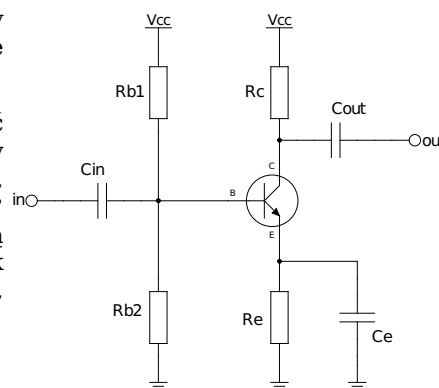
Wzmacniacz sygnału

punkt pracy

Aby móc wykorzystać tranzystor do wzmacniania sygnału konieczne jest wstępne ustalenie punktu

pracy tranzystora w ten sposób aby nie wchodził w stany nasycenia i zatkania oraz aby układ posiadał ustalone wzmocnienie.

W przypadku tranzystorów NPN i PNP można by próbować ustalić go z użyciem określonej wartości prądu bazy (pokazany powyżej układ z rezystorem bazy i rezystancją obciążenia), jednak ze względu na znaczny rozrzut wartości współczynnika β pomiędzy egzemplarzami układ taki cechowałby się małą powtarzalnością. W związku z tym stosuje się pokazany obok układ ze stałym potencjałem bazy i opornikiem emiterowym, gdzie $I_E = \frac{U_B - 0.6V}{R_e}$, $I_B = \frac{I_E}{\beta + 1}$, $I_C = I_E - I_B$, $\beta \gg 1$, zatem $I_C = I_E \frac{\beta}{\beta + 1} \approx I_E$.



wspólny emiter

Przedstawiony układ jest także przykładem jednego z 3 podstawowych układów pracy tranzystora bipolarnego (NPN, PNP) w funkcji wzmacniacza określanym "wspólny emiter". Układ tego typu charakteryzuje się wejściem sygnału na bazę tranzystora (napięcie bazy w stosunku co do napięcia emitera / masy dla układów NPN) i wyjściem wzmocnionego sygnału z kolektora (napięcie kolektorem w stosunku co do napięcia emitera / masy dla układów NPN).

W przypadku polaryzacji stałym prądem bazy emiter w układzie tym ma potencjał masy. W przypadku przedstawionej polaryzacji stałym potencjałem bazy aby w układzie tym móc uzyskać wzmocnienie konieczne jest zwarcie emitera do masy dla sygnałów zmiennych, realizowane przy pomocy kondensatora C_e .

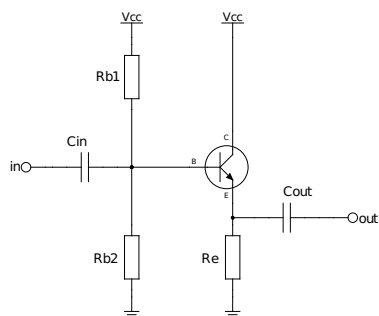
wspólny kolektor i wspólna baza

Kolejnymi dwoma z 3 podstawowych układów pracy są układy wspólnego kolektora i wspólnej bazy.

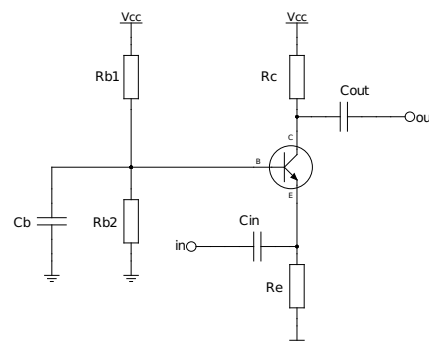
W układzie wspólnego kolektora sygnał wejściowy podawany jest między bazą a kolektor, natomiast wzmocniony sygnał odbierany jest z pomiędzy emitera i kolektora. Wzmacniacz w układzie wspólnego kolektora nie posiada wzmocnienia

napięciowego (napięcie wyjściowe jest wręcz pomniejszone o spadek na złączu PN), natomiast posiada znaczne wzmocnienie prądu. Układ często określany jest jako *wtórnik emiterowy*.

W układzie wspólnej bazy sygnał wejściowy podawany jest między emiter a bazą, natomiast wzmocniony sygnał odbierany jest z pomiędzy kolektora i bazy.



wspólny kolektor

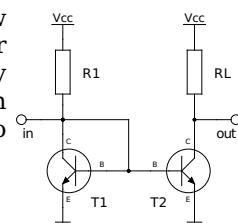


wspólna baza

Układy tranzystorowe

Lustro prądowe

T1 i T2 muszą mieć zapewnioną tą samą temperaturę (najlepiej być wykonane w ramach jednego układu scalonego. T1 pracuje jako dioda (stosujemy tranzystor aby zapewnić taką samą charakterystykę jak T2), która wraz z R1 tworzy sprzężony termicznie z T2 dzielnik sterujący tranzystorem T2. Układ ten sterowany prądem płynącym przez R1 powoduje przepływ (niemalże) takiego samego prądu poprzez RL.

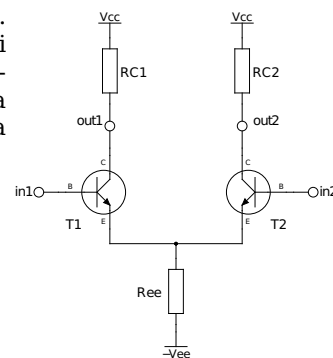


Wzmacniacz różnicowy

W przypadku układu z tranzystorami NPN "bardziej" przewodzić będzie tranzystor na wejściu którego jest większe napięcie (niż na tym drugim). Fakt tego że on przewodzi powoduje ustalenie się potencjału w węzle łączącym emiterów obu tranzystorów na wartość napięcia jego bazy pomniejszonego o spadek na przewodzącym złączu PN. W sytuacji gdy napięcia baz obu tranzystorów są odpowiednio bliskie przewodzić będą oba, zakres ten określa się strefą przejściową.

Układ ten może pełnić rolę klucza prądowego (gdy podany sygnał na tyle duży aby zatkać jeden z

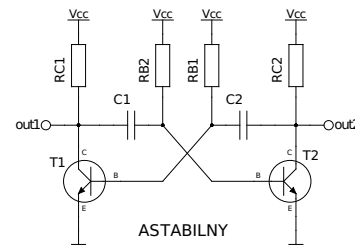
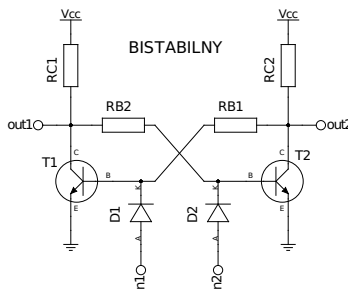
tranzystorów) lub wzmacniacza (gdy pracować będą oba tranzystory). Układ jest czuły na różnicę napięć przyłożonych do in1 i in2. Re_{ee} pełni rolę źródła prądowego (i powinien być tak dobrany aby wraz z wartością -V_{ee} zapewnić jego stabilność). Zastosowanie regulowanego źródła prądowego (tranzystor) zamiast Re_{ee} (realnie - w szeregu z Re_{ee}) pozwala na regulację wzmocnienia tego układu.



Przerzutniki

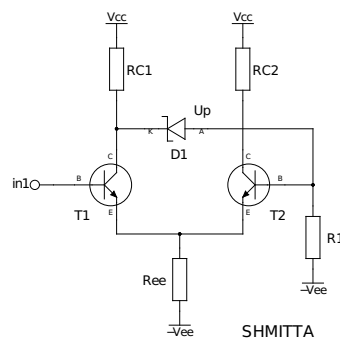
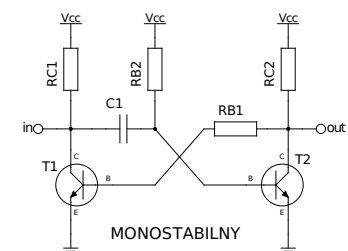
Przerzutnik bistabilny

Układ ten służy do zapamiętania stanu binarnego. Może być przełączony poprzez podanie krótkiego sygnału na którąś z wejść lub zwarcie któregoś wyjścia do masy. powoduje to rozpoczęcie przewodzenia przez wybrany z tranzystorów, co prowadzi do spadku napięcia na bazie drugiego tranzystora, prowadząc do jego zatkania i trwałego ustalenia stanu wysokiego na bazie wybranego tranzystora.



Przerzutnik monostabilny

Układ służy do generacji impulsu trwającego ustaloną długość. Działa on podobnie jak przedstawiony przerzutnik bistabilny, z taką różnicą iż krótkotrwałe zwarcie wejścia do masy powoduje obniżenie potencjału bazy T2 o napięcie do jakiego został naładowany C1 (w normalnym stanie $U_{CC} - U_{BE}$). Powoduje to zatkanie T2 i przewodzenie T1, następuje także ładowanie C1 z stopniowym zwiększaniem napięcia na bazie T2 aż do osiągnięcia sytuacji pierwotnej. Należy zwrócić uwagę że po zatkanie T1 (zaprzesztaniu generowania sygnału wyjściowego) układ nie osiągnął jeszcze gotowości do generacji następnego impulsu gdyż musi nastąpił ładowanie C1 przez RC1 (należy odczekać $t \gtrsim 5\tau = 5R_{C1}C_1$). Wadą takiego rozwiązania są duże ujemne napięcia na bazach tranzystorów w chwili przerzutu, co (przy większych napięciach zasilania) może doprowadzić do ich przebicia.



Przerzutnik astabilny

Przerzutnik ten służy do generacji sygnału prostokątnego o zadanym wypełnieniu. Działanie układu jest podobne do opisanego przerzutnika monostabilnego z tym że po (ponownym) nasyceniu drugiego tranzystora powstały spadek napięcia rozpoczyna taki sam proces dla pierwszego tranzystora i kondensatora podłączonego do jego bazy. Dodatkową wadą tego rozwiązania jest problem startu przy wypełnieniu 1/2 (jednakowych elementach) i łagodnym włączaniu.

Przerzutnik Shmitta

Służy od do uzyskania histerezy - większe napięcie powoduje przejście w stan wysoki, a mniejsze od niego w stan niski. Pozwala to m.in. na odszumianie sygnałów. Idea działania polega na zmianie potencjału drugiego wejścia wzmacniacza różnicowego zależnie od jego stanu. Przesuwnik poziomów zrealizowany na D1 oraz R1 służy do dostosowania napięcia wyjściowego do zakresu histerezy (wynosi on $U_{CC} - U_p$ = wartość górna, $U_{CC} - I_{EE}R_{C1} - U_p$ = wartość dolna).

Wzmacniacz operacyjny

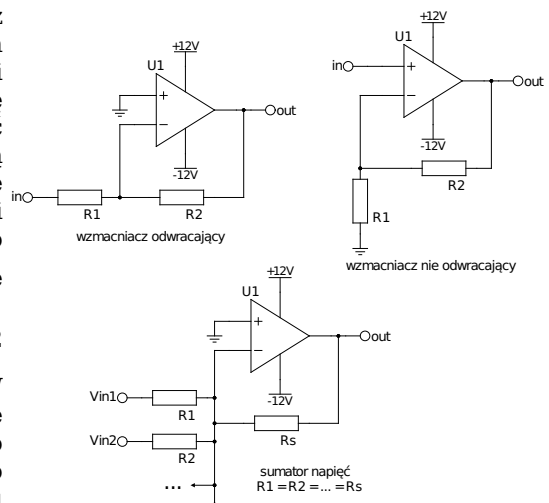
Jest to układ służący do wzmacniania bardzo wiele razy różnicy napięć wejściowych, oparty na układzie wzmacniacza różnicowego z lustrem prądowym. Na stopniu wejściowym posiada on wzmacniacz różnicowy z lustrem, dalej jest układ wzmacniający o dużym wzmocnieniu, a na stopniu wyjściowym wtórnik emiterowy. Charakteryzuje się dużą rezystancją wejściową i bardzo dużą wartością wzmocnienia.

Jako iż wzmocnienie wzmacniaczy operacyjnych jest bardzo duże nie da się w praktyce wykorzystać ich

bezpośrednio do wzmacniania sygnału. Robi się to z wykorzystaniem sprzężenia zwrotnego, czyli podaniem przeskalowanego sygnału wyjściowego na wejście. Dzięki wykorzystaniu ujemnego sprzężenia układ zachowuje się stabilnie dążąc do utrzymania różnicy napięć wejściowych bliskiej zera oraz posiada bardzo małą (prawie zerową) rezystancję wyjściową. Wzmocnienie zależy od parametrów dzielnika występującego w pętli sprzężenia zwrotnego i dla wzmacniacza odwracającego fazę wynosi $k_u = -\frac{R_2}{R_1}$, a dla wzmacniacza nie

odwracającego fazy $k_u = 1 + \frac{R_2}{R_1}$ (aby uzyskać wtórnik R2

należy zastąpić zwarcie, a R1 rozwarciem). W wzmacniaczu odwracającym rezystancja wejściowa $R_{we} = R_1$, natomiast w odwracającym jest to zasadniczo tylko rezystancję sumacyjną wzmacniacza różnicowego (różnicowej praktycznie nie widać bo dzięki sprzężeniu zwrotnemu napięcie na niej bliskie zeru). Częstotliwość graniczną oblicza się z zależności $f_g = \frac{f_T}{1 + \frac{R_2}{R_1}}$.



Niekiedy stosuje się także tzw "R3" podłączony do nieodwracającego wejścia wzmacniacza, ma to na celu minimalizację wpływu prądów wejściowych (nie wpływa na napięcie niesymetryczności, które jest cechą samego wzmacniacza), wtedy $R_3 = R_1 \parallel R_2$ gdy sprzężenie DC, lub $R_3 = R_2$ gdy sprzężenie AC (źródło sygnału oddzielone kondensatorem). Ze względu na stosowanie sprzężenia zwrotnego nie można na układ taki podawać sygnału którego przesunięcie fazowe na wzmacniaczu byłoby większe niż 135° , aby nie doprowadzić do wzbudzenia się układu. Uzyskuje się to poprzez zapewnienie wzmocnienia mniejszego od jedności dla częstotliwości przy których byłoby takie lub większe przesunięcie fazowe.

W oparciu o wzmacniacz operacyjny w układzie odwracającym można zrealizować sumator napięć wejściowych. Suma prądów wpływających przez rezystory R1, R2, R3, itd, ze względu na dużą rezystancję wejściową wzmacniacza musi być równa prądowi wypływającemu przez Rs. Napięcie na Rf musi być równe napięciu wyjściowemu (wejscie nie odwracające podłączone do masy, a układ dąży do zerowego napięcia różnicowego na wejściach wzmacniacza). Zatem $\frac{U_1}{R_1} + \frac{U_2}{R_2} + \dots = -\frac{U_{wy}}{R_s} \Leftrightarrow U_{wy} = -R_s(\frac{U_1}{R_1} + \frac{U_2}{R_2} + \dots)$ co przy równych wartościach rezystancji daje: $U_{wy} = U_1 + U_2 + \dots$.

Przykłady: LM358; TL081

Komparator analogowy

Jest to układ służący do porównywania dwóch napięć wejściowych, oparty (podobnie jak wzmacniacz operacyjny) na układzie wzmacniacza różnicowego z lustrem prądowym. Charakteryzuje się dużą rezystancją wejściową (małym poborem prądu z wejść), szybką i stromą zmianą wyjścia w odpowiedzi na zmianę sygnału wejściowego. Oprócz podstawowego układu pracy komparator może być stosowany m.in. w układach:

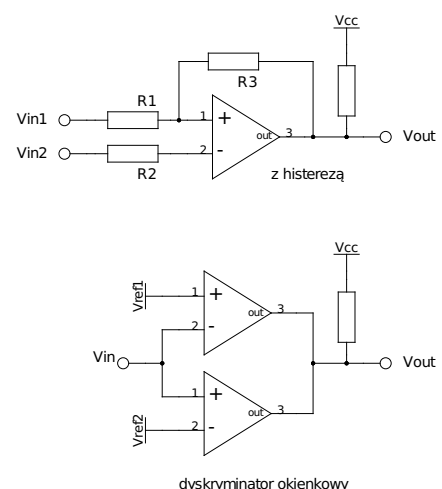
z histerezą:

- dzielnik R1, R3 zapewnia sprzężenie zwrotne, $R_3 \gg R_1$ (rzędu

mega omów), wartość histerezy dana jest zależnością: $V_H = V_{cc} \frac{R_1}{R_1 + R_3}$

- rezystor R2 zapewnia jednakowe rezystancję obu wejść komparatora i wynosić powinien: $R_2 \approx R_1$

dyskryminator okienkowy:



układ złożony z dwóch komparatorów wykrywający to iż napięcie znajduje się w przedziale pomiędzy V_{ref1} a V_{ref2} , jeżeli $V_{ref1} > V_{ref2}$ to znajdowanie się napięcia w zakresie będzie sygnalizowane stanem wysokim, w przeciwnym razie stanem niskim.

detektora przejścia przez zero:

zasadniczo dowolny układ (podstawowy, z histerezą, ...) wtedy gdy napięciem z którym porównujemy jest potencjał zera woltów (masa)

Przedstawione schematy są dla komparatora z wyjściem NPN (zwierającym do masy). Dla komparatorów z innym typem wyjścia mogą wymagać pewnych modyfikacji.

Przykłady: LM239; LM393; LM311

Cyfrowa

Algebra Boole'a i system dwójkowy

podstawowe operacje i elementy neutralne

suma logiczna (OR, +,)	$a \text{ OR } 1 = 1$	$a \text{ OR } 0 = a$	$a \text{ OR } a = a$
iloczyn logiczny (AND, *, &)	$a \text{ AND } 1 = a$	$a \text{ AND } 0 = 0$	$a \text{ AND } a = a$
negacja (NOT, ~, ^, !)	$\text{NOT } 1 = 0$	$\text{NOT } 0 = 1$	$\text{NOT } (\text{NOT } a) = a$

własności działań

łączność	$(a \text{ OR } b) \text{ OR } c = a \text{ OR } (b \text{ OR } c)$	$(a \text{ AND } b) \text{ AND } c = a \text{ AND } (b \text{ AND } c)$
przemienność	$a \text{ OR } b = b \text{ OR } a$	$a \text{ AND } b = b \text{ AND } a$
rozdzielność	$a \text{ AND } (b \text{ OR } c) = (a \text{ AND } b) \text{ OR } (a \text{ AND } c)$	$a \text{ OR } (b \text{ AND } c) = (a \text{ OR } b) \text{ AND } (a \text{ OR } c)$
absorpcja	$a \text{ AND } (b \text{ OR } a) = a$	$a \text{ OR } (b \text{ AND } a) = a$
negacja sumy i iloczynu	$\text{NOT } (a \text{ OR } b) = (\text{NOT } a) \text{ AND } (\text{NOT } b)$	$\text{NOT } (a \text{ AND } b) = (\text{NOT } a) \text{ OR } (\text{NOT } b)$
pochłanianie	$a \text{ OR } (\text{NOT } a) = 1$	$a \text{ AND } (\text{NOT } a) = 0$

dodatkowe operacje

alternatywa wykluczająca (XOR)	$a \text{ XOR } b = (a \text{ AND } (\text{NOT } b)) \text{ OR } (b \text{ AND } (\text{NOT } a))$	$a \text{ XOR } 0 = a$	$a \text{ XOR } a = 0$
$a \text{ NAND } b = \text{NOT } (a \text{ AND } b)$	$(a \text{ NAND } b) \text{ NAND } (c \text{ NAND } d) = (a \text{ AND } b) \text{ OR } (c \text{ AND } d)$		
$a \text{ NOR } b = \text{NOT } (a \text{ OR } b)$	$(a \text{ NOR } b) \text{ NOR } (c \text{ NOR } d) = (a \text{ OR } b) \text{ AND } (c \text{ OR } d)$		
$a \text{ XNOR } b = \text{NOT } (a \text{ XOR } b)$			

tablice prawdy

a	b	a OR b	a AND b	a NOR b	a NAND b	a XOR b	a XNOR b	NOT a
0	0	0	0	1	1	0	1	1
0	1	1	0	0	1	1	0	
1	0	1	0	0	1	1	0	
1	1	1	1	0	0	0	1	0

reprezentacja liczb

Pojedynczą cyfrę systemu dwójkowego (przybierającą wartość 0 albo 1) określa się mianem *bitu*, liczby reprezentowane są jako ciągi takich cyfr. Terminem *bajt* określa się zazwyczaj ciąg o długości 8 bitów (ale w niektórych systemach ciąg o innej długości).

Podstawowym sposobem zapisy liczb całkowitych nie ujemnych w systemie dwójkowym jest *naturalny*

kod binarny (NKB), w którym np. 4 bitowy ciąg $a_3 a_2 a_1 a_0$ reprezentuje liczbę $2^0 \cdot a_0 + 2^1 \cdot a_1 + 2^2 \cdot a_2 + 2^3 \cdot a_3$.

Podstawowym sposobem zapisu liczb całkowitych (ze znakiem) jest *kod uzupełnień do dwóch (U2)* w którym n-bitowa liczba reprezentowana przez ciąg $a_{n-1} \dots a_3 a_2 a_1 a_0$ będzie miała wartość $2^0 \cdot a_0 + 2^1 \cdot a_1 + 2^2 \cdot a_2 + \dots + 2^{n-2} \cdot a_{n-2} - 2^{n-1} \cdot a_{n-1}$. Jako że najstarszy bit wchodzi z ujemną wagą, jego ustawienie na 1 oznacza liczbę ujemną (ale nie jest to kod znaku). Warto zauważyć kompatybilność z NKB.

Liczby zapisywane w tych kodowaniach systemu dwójkowego oznaczają się często przy pomocy prefiksu "0b" albo sufiksu "b", np. $0b101 = 101b$ reprezentuje liczbę 5 w systemie dziesiętnym ($2^0 \cdot 1 + 2^1 \cdot 0 + 2^2 \cdot 1 = 5$).

Oprócz podanych istnieje jeszcze kilka stosowanych sposobów zapisu liczb binarnych takich jak (dla liczb bez znaku): kod "1 z n", kod Graya, kod Johnsona, (dla liczb ze znakiem): kod znak-moduł, kod uzupełnień do jedności (U1). Odmiernym zagadnieniem jest kodowanie liczb zmiennoprzecinkowych.

zapis hexalny (szesnastkowy)

Celem skrócenia zapisu liczb binarnych często zapisuje się je w postaci szesnastkowej, jest on wygodniejszy od dziesiętkowego gdyż każda cyfra systemu szesnastkowego rozkłada się na dokładnie 4 bity, co pozwala na niezależne konwertowanie poszczególnych cyfr szesnastkowych / ciągów 4 bitowych i ich łączenie w dłuższe ciągi.

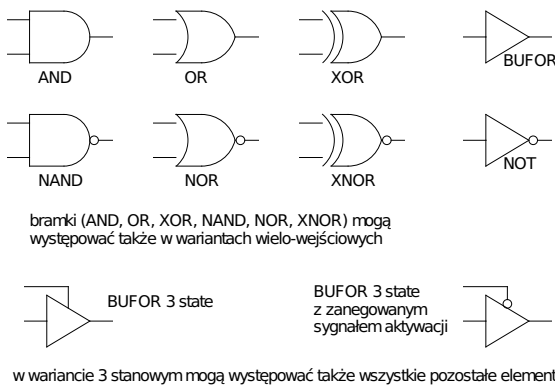
Cyfry o wartościach powyżej 9 zapisuje się jako kolejne małe lub wielkie litery a, b, c, d, e, f. Liczby zapisywane w systemie szesnastkowym oznaczają się przy pomocy prefiksu "0x" lub "#" albo sufiksu "h" np. $0xc7 = \#c7 = c7h$ reprezentuje liczbę 199 w systemie dziesiętkowym ($16^0 \cdot 7 + 16^1 \cdot 12 = 199$). Konwersja na system binarny może odbywać się niezależnie dla każdej cyfry, jako że: $0xc = 0b1100$ oraz $0x7 = 0b0111$ to $0xc7 = 0b11000111$.

reprezentacja elektryczna

Typowo logicznej 1 odpowiada stan wysoki (napięcie dodatnie), a logicznemu 0 stan niski (potencjał masy). W przypadku logiki odwróconej (ujemnej) logicznej 1 odpowiada stan niski a logicznemu zeru wysoki.

Bramki

Bramki są układami elektronicznymi realizującymi podstawowe, opisane powyżej funkcje logiczne. Obok zostały przedstawione podstawowe symbole poszczególnych bramek w wariantach dwu wejściowych, spotkać się można także z symbolami z zanegowanymi wejściami - w takiej konwencji np. bramka AND reprezentowana jest przez NAND z zanegowanymi wejściami. Bramki (z wyjątkiem buforów oraz bramki NOT), mogą występować także w wariantach wielo-wejściowych (ze względu na łączność podstawowych operacji nie ma wątpliwości co do wyniku jaki powinna dawać np. 8 wejściowa bramka OR). Na ogół w pojedynczym układzie scalonym znajduje się kilka jednakowych bramek.

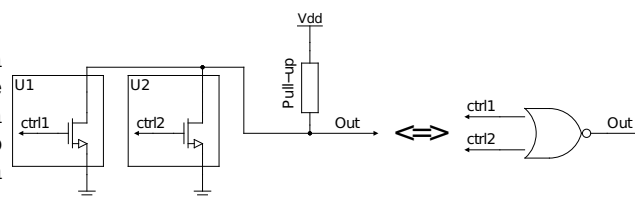


trój-stanowe

Typowa bramka wymusza (w sposób silny) na swoim wyjściu stan wysoki lub niski, co uniemożliwia bezpośrednie łączenie wyjść bramek. Bramki trój-stanowe mają możliwość skonfigurowania wyjścia w stan *wysokiej impedancji* czyli nie wymuszania żadnej jego wartości. Sterowanie załączeniem bądź wyłączeniem wyjścia (przełączeniem w stan wysokiej impedancji) odbywa się przy pomocy zewnętrznego sygnału sterującego "output enabled" ("OE"), sygnał ten może występować w postaci prostej i zanegowanej. Pozwala to na podłączanie do jednej linii wielu bramek i decydowaniu która z nich będzie nią sterować.

open collector / drain

Są kolejnym rodzajem bramek których wyjścia można podłączać do wspólnej linii. Układy te posiadają wyjście w postaci tranzystora zwierającego linię wyjściową do masy, z tego względu samodzielnie zapewniają jedynie stan niski wyjścia (są w stanie wymusić stan niski, ale nie mają możliwości wymuszenia stanu wysokiego).



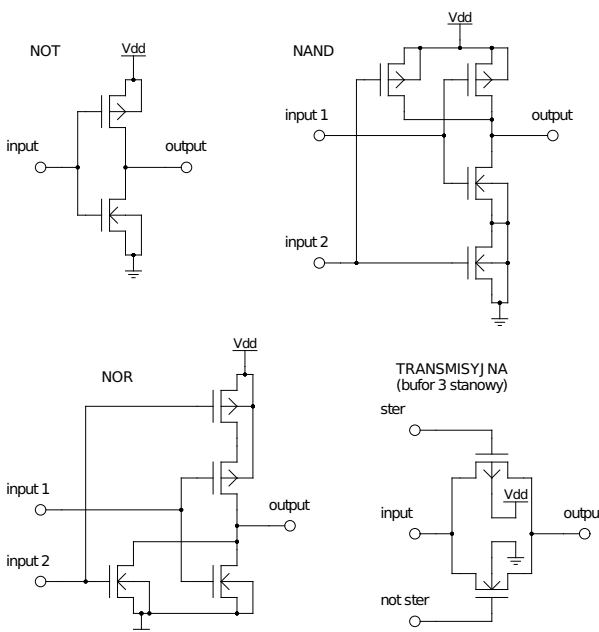
Stan wysoki musi zostać zapewniony zewnętrznym rezystorem podciągającym. Pozwala to stosować

na takiej linii inny poziom stanu wysokiego niż na wejściach takiej bramki oraz pozwala na sterowanie wspólną linią przez wiele bramek (czyli łączenie wyjść bramek, jednak w odróżnieniu od bramek trójstanowych nie wymaga dodatkowych sygnałów sterujących).

Na schemacie obok przedstawiono dwa układy (U1 i U2) typu open-drain sterujące wspólną linią wyjściową w układzie *suma na drucie*. Jeżeli jeden z podłączonych do linii układów będzie miał wewnętrzne wyjście ("ctrlX") w stanie wysokim to jego wyjście OC będzie zwarte do masy (negacja na tranzystorze N-MOS lub NPN), wtedy też cała linia będzie w stanie niskim.

budowa wewnętrzna

Przedstawiony powyżej układ sumy na drucie jest bardzo prostą (jedno tranzystorową) realizacją bramki logicznej realizującą funkcję logiczną NOT OR (z punktu widzenia wejść *ctrl1* i *ctrl2* oraz wyjścia *Out*). W podobny sposób można zrealizować bramkę AND (negując wejścia, np. przy pomocy jednego tranzystora). Jeszcze bardziej uproszczoną realizację można uzyskać stosując diody pozwalające na wpływanie prądu do węzła (funkcja OR) lub wypływanie z niego (funkcja AND).



Po prawej przedstawione zostały schematy ideowe inwertera, dwóch podstawowych bramek (NOR i NAND) oraz bramki transmisyjnej (bufora 3 stanowego) w technologii CMOS.

Działanie tych bramek (za wyjątkiem transmisyjnej) polega na otwieraniu tranzystorów podłączonych do napięcia które chcemy otrzymać na wyjściu, a zamykaniu prowadzących do napięcia przeciwnego. W szczególności bramka NOT stanowi pół-mostek H pomiędzy stanem wysokim a stanem niskim. Dzięki zastosowaniu tranzystorów PMOS polaryzowanych Vdd oraz NMOS polaryzowanych GND obie gałęzie operują na tym samym sygnale wejściowym (nie jest wymagana jego negacja). Szeregowe łączenie tranzystorów zapewnia że należy otworzyć oba aby otworzyć daną drogę, a równoległe że otwarcie danej drogi powodowane jest otwarciem pojedynczego tranzystora. Dzięki zastosowaniu technologii MOS i podłączaniu wejść bramki tylko do bramek tranzystorów wejścia praktycznie nie pobierają prądu (istotnym wyjątkiem jest chwila zmiany sygnału).

Działanie bramki transmisyjnej polega na przepuszczaniu lub nie (w zależności od stanu wejścia sterującego) sygnału z wejścia na wyjście. Bramka taka nie regeneruje sygnału. Ponadto w uproszczonym (jedno tranzystorowym) rozwiązaniu prowadzi ona do degradacji sygnału wartość w przybliżeniu równą napięciu progowemu tranzystora. Dlatego też na ogół występuje wraz z bramką NOT (bufor 3 stanowy z negacją) lub dwiema szeregowo połączonymi bramkami NOT (bufor 3 stanowy bez negacji).

wejście Schmitta

Układy z wejściem Schmitta zapewniającym histerezę sygnału wejściowego stosowane są w przypadku mało stromych, zaszumionych sygnałów dla których układy z normalnym wejściem mogłyby zmienić kilkakrotnie stan wyjścia na jednym zboczach sygnału wejściowego.

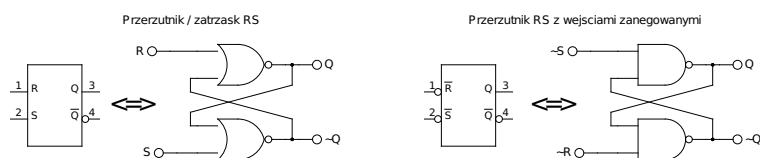
Przykłady: 74HC00 (4 x 2-input NAND); 74HC1G00 (1 x 2-input NAND); 74HC20 (2 x 4-input NAND); 74HC30 (1 x 8-input NAND); 74HC03 (4 x 2-input NAND, open-drain); 74HC132 (4 x 2-input NAND, Schmitt); 74HC04 (6 x NOT); 74HC1G04 (1 x NOT); 74HC14 (6 x NOT, Schmitt); 74HC1G14 (1 x NOT, Schmitt); 74HC02 (4 x 2-input NOR); 74HC1G02 (1 x 2-input NOR); 74HC08 (4 x 2-input AND); 74HC1G08 (1 x 2-input AND); 74HC32 (4 x 2-input OR); 74HC1G32 (1 x 2-input OR); 74HC86 (4 x 2-input XOR); 74HC1G86 (1 x 2-input XOR); 74HC7226 (4 x 2-input XNOR); 74HC7014 (6 x Schmitt);

Przerzutniki i rejestry

przerzutnik / zatrask RS

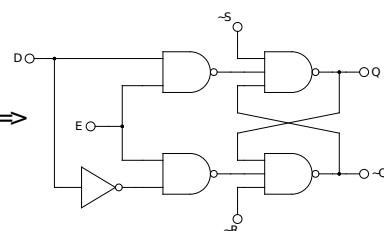
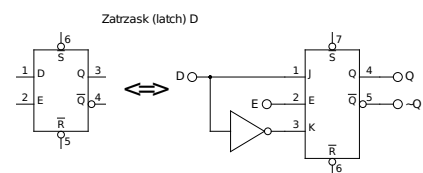
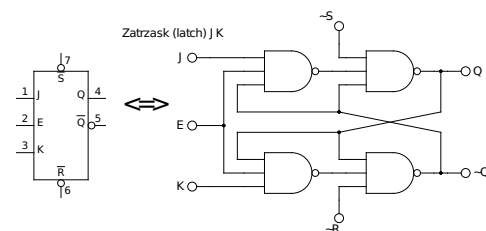
RS Flip-flop (RS Latch) jest podstawowym układem służącym do zapamiętania jednego bitu informacji. Posiada on dwa wejścia (set i reset) i dwa wyjścia (Q i NOT Q), wejścia mogą reagować na stan wysoki (oznaczane jako S i R) lub niski (oznaczane jako wejścia zanegowane $\sim S$ i $\sim R$), jedno z wyjść może być jedynie wewnętrzne (nie wyprowadzone na zewnątrz układu). Podanie stanu wysokiego na wejście S (niskiego na $\sim S$) powoduje wystawienie stanu wysokiego na wyjściu Q, a podanie stanu wysokiego na wejście R (niskiego na $\sim R$) powoduje wystawienie stanu niskiego na wyjściu Q. Stan na wyjściu Q nie zmienia się po zmianie wejść S i R na stan niski (zostaje zapamiętany).

Na schemacie przedstawiono dwubramkową budowę przerzutnika RS zarówno w wariantcie z wejściami niezanegowanymi jak i w wariantcie z wejściami zanegowanymi.

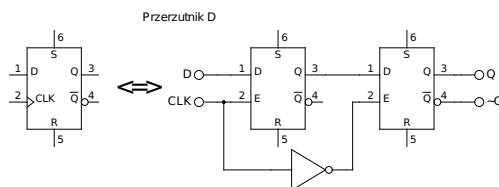


zatrząsk a przerzutnik

Zatrząsk jest elementem reagującym na poziom sygnału na wejściu "enable" (E). W przypadku niezanegowanego wejścia E, jeżeli jest ono w stanie wysokim sygnał na wyjściach (Q i NOT Q) jest funkcją sygnałów wejściowych, natomiast stan niski wejścia E blokuje zmianę sygnału wyjściowego (zostaje on zapamiętany).



Przerzutnik jest elementem reagującym na zbocze sygnału na wejściu "clock" (CLK). W zależności od konstrukcji może reagować na zbocze narastające, opadające albo na oba (wtedy na jednym realizuje odczyt wejść a na drugim zmianę stanu wyjść).



zatrząsk i przerzutnik JK

Posiada dwa wejścia sygnałowe (J i K) oraz wejście "enable" (E) w przypadku zatrząsku lub wejście "clock" (CLK) w przypadku przerzutnika. Może także posiadać asynchroniczne (niezależne od stanu wejścia E / CLK) wejścia reset i set (zanegowane lub proste).

J	K	Q_{n+1}	Opis
0	0	Q_n	brak zmiany
0	1	0	reset
1	0	1	set
1	1	$\text{NOT } Q_n$	zmiana wyjścia na przeciwny

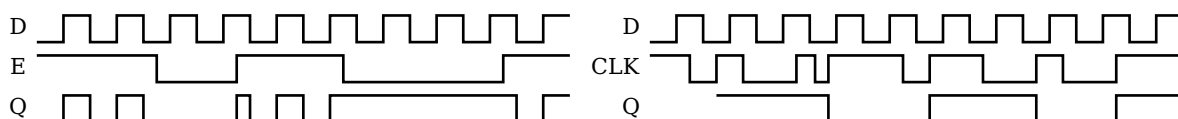
Stan wejść J = K = 1 dozwolony jest tylko w przypadku przerzutnika, gdyż wtedy sygnał CLK decyduje o momencie zmiany wyjścia Q na przeciwny. W przypadku zatrząsku, przy wysokim E wystąpiłyby oscylacje na wyjściu.

Przykłady: 74HC73 (2 x przerzutnik JK, zbocze opadające, wejścia ~R); 74HC107 (2 x przerzutnik JK, zbocze opadające, wejścia ~R); 74HC112 (2 x przerzutnik JK, zbocze narastające, wejścia ~S i ~R); 74HC112 (2 x przerzutnik JK, zbocze opadające, wejścia ~S i ~R)

zatrząsk i przerzutnik D

Posiada jedno wejście sygnałowe "data" (D) oraz wejście "enable" (E) w przypadku zatrząsku lub wejście "clock" (CLK) w przypadku przerzutnika. Może także posiadać asynchroniczne (niezależne od stanu wejścia E / CLK) wejścia reset i set (zanegowane lub proste). Obniżenie sygnału E lub zbocze sygnału CLK powodują zapamiętanie (i wystawienie na wyjściu Q) stanu wejścia D.

W przypadku zaprezentowanej realizacji przerzutnika z dwóch zatrząsków sygnał wejściowy zostanie odczytany na zboczu opadającym zegara a wystawiony na wyjście na zboczu narastającym.



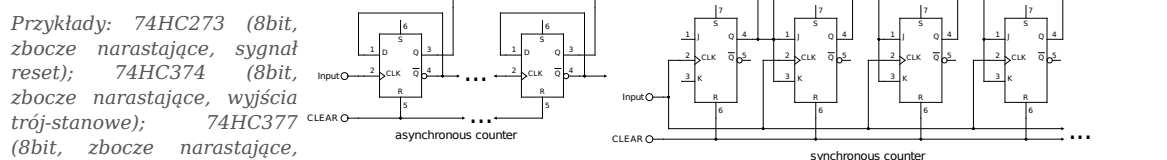
Przykłady: 74HC74 (2 x przerzutnik D, zbocze narastające, wejścia $\sim S$ i $\sim R$); 74HC173 (4 x przerzutnik D, zbocze narastające, sygnał input-enable, wspólny reset i clock, wyjścia Q trój-stanowe); 74HC174 (6 x przerzutnik D, zbocze opadające, wspólny reset i clock); 74HC175 (4 x przerzutnik D, zbocze narastające, wspólny reset i clock, wyjścia Q i NOT Q);

rejestry

Mianem rejestru n-bitowego określa się zespół n przerzutników (rzadziej zatrzasków), często z uwspólnionym sterowaniem (sygnały clock, set, reset, etc) służący do zapamiętania n-bitowej wartości (liczby). W zależności od sposobu zapisu i odczytu można wyróżnić:

rejestry równoległe

Posiadają taką samą liczbę wejść jak i wyjść, sygnał na i-tym wyjściu jest bezpośrednio powiązany z sygnałem z i-tego wejścia (jest sygnałem zapamiętanym z tego wejścia).



Przykłady: 74HC273 (8bit, zbocze narastające, sygnał reset); 74HC374 (8bit, zbocze narastające, wyjścia trój-stanowe); 74HC377 (8bit, zbocze narastające, sygnał reset, sygnał input-enable); 74HCT534 (8bit, zbocze narastające, wyjścia trój-stanowe zanegowane); 74HC574 (8bit, zbocze narastające, wyjścia trój-stanowe); 74HC373 (8bit, zatrzask przeźroczysty w stanie wysokim, wyjścia trój-stanowe); 74HC563 (8bit, zatrzask przeźroczysty w stanie wysokim, wyjścia trój-stanowe zanegowane); 74HC573 (8bit, zatrzask przeźroczysty w stanie wysokim, wyjścia trój-stanowe);

rejestry szeregowe serial-input

Z kolejnymi sygnałami zegarowymi odczytywany jest stan wejścia szeregowego, a stan poprzedni przenoszony jest do kolejnego przerzutnika w ramach rejestru. W ten sposób po n cyklach zegara n-bitowy rejestr ma zapisaną nową zawartość. Często posiada zespolony z nim rejestr równoległy zapobiegający zmianie stanu wyjść w trakcie ładowania danych z wejścia szeregowego przepisanie danych z rejestru przesuwanego do rejestru odpowiedzialnego za sterowanie wyjściami sterowane jest osobnym sygnałem zegarowym.

Przykłady: 74HC164 (8 bit, bez rejestru wyjściowego); 74HC194 (4 bit, wejście równoległe, przesuwanie w prawo i w lewo); 74HC195 (4 bit, wejście równoległe); 74HC594 (8 bit, zatrzask wyjściowy z resetem, wyjście szeregowe, sygnał resetu); 74HC595 (8 bit, zatrzask wyjściowy z wyjściami trój-stanowymi, wyjście szeregowe, sygnał resetu); 74HC4094 (8 bit, zatrzask wyjściowy z wyjściami trój-stanowymi, wyjście szeregowe, zatrzask wyjścia szeregowego);

rejestry szeregowe parallel-input serial-output

Z kolejnymi sygnałami zegarowymi na wyjście szeregowe wystawiany jest stan kolejnego z rejestrów wejściowych. Wariant asynchroniczny posiada osobny sygnał powodujący odczyt wejść do rejestru (sygnał działa jak "enable" w zatrzaskach). Wariant synchroniczny posiada sygnał decydujący o tym czy na zboczu zegara dokonywany jest odczyt wejść czy też przesuwanie zawartości rejestru umożliwiając odczyt z wyjścia szeregowego.

Przykłady: 74HC165 (asynchroniczne wczytanie danych wejściowych, wejście szeregowe, sygnał clock-enable); 74HC166 (synchroniczne wczytanie danych wejściowych, wejście szeregowe, sygnał clock-enable); 74HC589 (dodatkowy zatrzask wejściowy, asynchroniczne wczytanie danych wejściowych, wejście szeregowe, wyjścia trój-stanowe); 74HC597 (dodatkowy zatrzask wejściowy, asynchroniczne wczytanie danych wejściowych, wejście szeregowe, sygnał clock-enable);

liczniki

Z kolejnymi sygnałami zegarowymi zwiększana jest o jeden wartość rejestru. Prostszy w budowie licznik asynchroniczny ma większe (i w dodatku rosnące wraz z bitowością licznika) ograniczenia dotyczące szybkości zliczania od licznika synchronicznego, ze względu na opóźnienie z jakim dochodzi zliczany sygnał (CLK) do kolejnych stopni licznika.

Przykłady: 74HC93 (2x 4bit synchroniczny, zbocze opadające); 74HC161 (4bit synchroniczny, zbocze narastające, preset wartością początkową, wejście zezwolenia na zliczanie, sygnał przepełnienia); 74HC163 (4bit synchroniczny, zbocze opadające, preset wartością początkową, wejście zezwolenia na zliczanie, sygnał przepełnienia); 74HC191 (4bit synchroniczny, zbocze narastające, liczący w górę i w dół - sygnał wyboru, wejście zezwolenia na zliczanie, preset wartością początkową); 74HC192 (4bit synchroniczny, zbocze narastające, liczący w górę i w dół - osobne zegary, wejście zezwolenia na zliczanie, preset wartością początkową, sygnał przepełnienia przy 9 - BCD); 74HC193 (4bit synchroniczny, zbocze narastające, liczący w górę i w dół - osobne zegary, wejście zezwolenia na zliczanie, preset wartością początkową, sygnał przepełnienia przy 15); 74HC393 (2x 4bit asynchroniczny, zbocze opadające, sygnał resetu); 74HC40103 (8 bit synchroniczny); 74HC4017 (dekadowy - wartości odpowiada numer aktywnego wyjścia, 10 wyjść); 74HC4020 (14 bitowy asynchroniczny); 74HC4040 (12 bitowy asynchroniczny); 74HC390; 74HC4024; 74HC4060; 74HC4518; 74HC4520;

Przetworniki ADC i DAC

Przetwornik analogowo-cyfrowy (ADC) służy do konwersji sygnału analogowego na postać cyfrową. Realizowane jest to poprzez pomiar napięcia, na ogół w regularnych odstępach czasowych (celem uzyskania przebiegu sygnału a nie tylko wartości chwilowej). Przetwornik ADC o porównaniu bezpośrednim realizowany jest w oparciu o zespół komparatorów analogowych (od ich liczby zależy bitowość danego przetwornika, ale ich liczba nie jest równa bitowości) które używają różnych napięć referencyjnych (typowo uzyskiwanych z jednego napięcia referencyjnego poprzez dzielnik). Stan z komparatorów podawany jest na enkoder celem konwersji do kodu binarnego. Inne stosowane sposoby realizacji przetworników ADC opierają się o pojedynczy komparator i podawanie na niego narastającego napięcia referencyjnego oraz zliczanie liczby kroków podnoszenia tego napięcia bądź podawaniu kolejnych napięć z przetwornika DAC i wyszukiwaniu tego które jest najbliższe wejściowemu.

Przetwornik cyfrowo-analogowy (DAC) służy do konwersji sygnału cyfrowego na analogowy. Oparty jest na zasadzie sumatora napięć, którego wejścia załączane są w zależności od ustawienia lub nie danego bitu w konwertowanej wartości. Typowo zamiast stosowania różnych wartości napięć i jednakowych rezystorów (jak w sumatorze), stosuje się różne wartości załączanych rezystorów i jednakowe napięcie do którego są podłączone.

pomiar napięcia i prądu

Pomiar napięcia realizuje bezpośrednio przetwornik ADC. W przypadku konieczności pomiaru wysokich napięć stosuje się przekładniki napięciowe, będące w istocie transformatorami o dobrze ustalonych parametrach pomiarowych. W przypadku małych napięć konieczne może okazać się ich wzmocnienie np. z użyciem wzmacniacza operacyjnego.

Pomiar prądu może być realizowany na kilka sposobów:

- jako pomiar napięcia na rezystancji włączonej w mierzony obwód; umożliwia pomiar prądów zmiennych i stałych
- poprzez przekładnik prądowy - transformator włączony szeregowo w obwód lub toroidalną cewkę przez którą przeprowadzony jest przewód w którym dokonywany jest pomiar prądu (pojedyncze uzwojenie pierwotne); stosuje się tylko dla prądów przemiennych (zmienny prąd w przewodzie powoduje powstanie indukcji magnetycznej która wymusza przepływ prądu w obwodzie pomiarowym)
- z wykorzystaniem efektu Halla (prąd może przepływać bezpośrednio przez układ pomiarowy, może płynąć pętlą ścieżki umieszczonej z drugiej strony płytki drukowanej niż czujnik efektu Halla lub może być realizowany analogicznie do toroidalnego przekładnika prądowego); umożliwia pomiar prądów zmiennych i stałych

Przykłady: ACS712 (czujnik prądu AC i DC oparty na efekcie Halla z wyjściem napięciowym, oporność toru pomiarowego 1.2 mΩ, wersje: 5A (185 mV/A), 20A (100 mV/A) i 30A (66 mV/A); ACS770 (czujnik prądu AC i DC oparty na efekcie Halla z wyjściem napięciowym, oporność toru pomiarowego 0.1 mΩ, wersje: 50A, 100A i 200A); WCS1800 (przekładnik prądowy z wbudowanym czujnikiem prądu AC i DC opartym na efekcie Halla z wyjściem napięciowym, czułość: 66 mV/A, 25A AC / 35A DC); WCS1600 (przekładnik prądowy z wbudowanym czujnikiem prądu AC i DC opartym na efekcie Halla z wyjściem napięciowym, 70A AC / 100A DC); SS49E (czujnik efektu Halla); SS411P (czujnik efektu Halla); TLE4905L (czujnik efektu Halla);

Transmisja

Sterowanie linią

bufory

Bufor jest to układ przekazujący sygnał logiczny z wejścia na wyjście. Bufor może służyć do:

- regeneracji sygnału,
- uniemożliwieniu wprowadzenia sygnału z jego strony wyjściowej na wejściową,
- decydowania o jego przepuszczeniu lub nie (trój-stanowy),
- decydowania o kierunku przepuszczenia sygnału (dwa trój-stanowe albo trój-stanowy dwukierunkowy),

- dwukierunkowy bufor trójstanowy
 $OE \equiv 1 \ \&\& \ DIR \equiv 0 \Rightarrow B \text{ do } A$
 $OE \equiv 1 \ \&\& \ DIR \equiv 1 \Rightarrow A \text{ do } B$
-

The figure contains two main logic diagrams, each with a functional block representation on the left and a detailed gate-level implementation on the right.

Top Diagram: Priority Encoder 4 to 2

- Functional Block (Left):** A box labeled "priority encoder 4 to 2". It has four inputs: IN3, IN2, IN1, and IN0. It has three outputs: VALID, OUT1, and OUT0.
- Gate-Level Implementation (Right):** Shows the internal logic using four 2-input AND gates and two 2-input OR gates.
 - AND1 (top) takes IN3 and IN2 as inputs. Its output goes to OUT1.
 - AND2 (middle) takes IN2 and IN1 as inputs. Its output goes to OUT0.
 - AND3 (bottom) takes IN1 and IN0 as inputs. Its output goes to OUT0.
 - AND4 (right) takes IN3 and IN0 as inputs. Its output goes to OUT1.
 - OR1 (left) takes the outputs of AND1 and AND2 as inputs. Its output is VALID.
 - OR2 (right) takes the outputs of AND3 and AND4 as inputs. Its output is VALID.

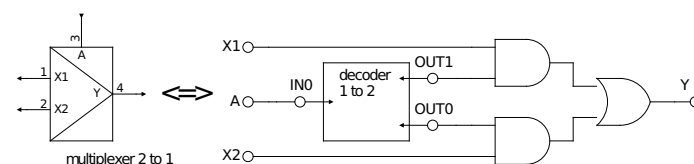
Bottom Diagram: Decoder 2 to 4

- Functional Block (Left):** A box labeled "decoder 2 to 4". It has two inputs: OE1 and OE2. It has four outputs: OUT3, OUT2, OUT1, and OUT0.
- Gate-Level Implementation (Right):** Shows the internal logic using four 2-input AND gates.
 - AND1 (top) takes OE1 and OE2 as inputs. Its output goes to OUT3.
 - AND2 (middle) takes OE1 and NOT OE2 as inputs. Its output goes to OUT2.
 - AND3 (bottom) takes NOT OE1 and OE2 as inputs. Its output goes to OUT1.
 - AND4 (right) takes NOT OE1 and NOT OE2 as inputs. Its output goes to OUT0.

Enkoder " n to m " jest to układ o n wejściach, który na swoim m bitowym wyjściu wystawia numer (typowo) wejścia o najwyższym numerze, które znajduje się w stanie niskim. Możliwe są też warianty wystawiające numer pierwszego (a nie ostatniego) w kolejności wejścia lub wybierające wejście ze stanem wysokim.

jeżeli IN1, IN2 traktować jako adres, to:

- OE1 można traktować jako sygnał poprawności adresu (HAS_INPUT z encodera)
- OE2 można traktować jako linie transmitująca dane do urządzenia o podanym adresie



demultiplexer oparty o bufor 3 stanowe

demultiplexer oparty o multiplexser

Przykłady: 74HC148 (8 to 3, priority, active low); 74HC147 (9 + "none" to 4 / 10 to 4, priority, active low);

Dekoder "m to n" jest układem o działaniu przeciwnym do enkodera. Aktywuje on wyjście o numerze odpowiadającym wartości na m-bitowym wejściu adresowym. Typowo posiada także wejście zezwolenia na aktywację wyjść, które może zostać użyte do podłączenia informacji że którekolwiek z wejść enkodera było w stanie aktywnym lub do podłączenia sygnału danych z multipleksowanej linii celem jej demultipleksacji.

Multiplexer cyfrowy (jednokierunkowy) na wyjście kopiuje stan wskazanego (poprzez adres podany na wejście adresowe) wejścia. W przypadku braku sygnału "enable" w zależności od rozwiązania wyjście pozostanie w stanie niskim lub wysokiej impedancji.

Demultiplekser cyfrowy (jednokierunkowy) to zazwyczaj układ dekodera w którym na wejście enabled

podany jest sygnał z multipleksowanej linii. Nie wybrane wyjścia pozostają w stanie niskim lub wysokim (zależnie od użycia nieodwracającego lub odwracającego dekodera). Cyfrowe demultipleksery z 3 stanowym wyjściem są rzadkością. Demultipleksację można rozwiązać także przy pomocy odpowiednio sterowanych (np. z dekodera adresu) buforów trój-stanowych lub dwu-wyjściowych multipleksorów.

Przykłady: 74HC151 (1 x "8 to 1", multiplekser cyfrowy jednokierunkowy, wyjścia proste i zanegowane); 74HC153 (2 x "4 to 1", multiplekser cyfrowy jednokierunkowy, wspólne wejścia adresowe); 74HC157 (4 x "2 to 1", multiplekser cyfrowy jednokierunkowy, wspólne wejścia adresowe); 74HC251 (1 x "8 to 1", multiplekser cyfrowy jednokierunkowy, wyjścia trój-stanowe proste i zanegowane); 74HC253 (2 x "4 to 1", multiplekser cyfrowy jednokierunkowy, wyjścia trój-stanowe, wspólne wejścia adresowe, osobne wejścia output-enable); 74HC257 (4 x "2 to 1", multiplekser cyfrowy jednokierunkowy, wyjścia trój-stanowe, wspólne wejścia adresowe i output-enable); 74LVC1G18 (1 x "1 to 2", demultiplekser cyfrowy jednokierunkowy, wyjścia trój-stanowe, można zastąpić układem połączonych 74HC1G125 z 74HC1G126);

(de)multipleksery analogowy

Multiplekser analogowy (dwukierunkowy) działa na zasadzie przełącznika łączącego wskazane wejście z wyjściem, dzięki elektrycznemu "zwarciu" (na ogół rezystancja takiego zwarcia to kilkadziesiąt omów) wejścia z wyjściem transmisja może odbywać się w obu kierunkach. Pozwala to także na wykorzystanie tego samego układu w roli multipleksera i demultipleksera.

Przykłady: 74HC4053 (3 x "2 to 1 / 1 to 2", multiplekser analogowy dwukierunkowy, osobne wejścia adresowe, wspólne wejście enable, trój-stanowy); 74HC4052 (2 x "4 to 1 / 1 to 4", multiplekser analogowy dwukierunkowy, wspólne wejścia adresowe i enable, trój-stanowy); 74HC4051 (1 x "8 to 1 / 1 to 8", multiplekser analogowy dwukierunkowy, trój-stanowy); 74HC4067 (1 x "16 to 1" / 1 to 16", multiplekser analogowy dwukierunkowy, trój-stanowy); 74HC4066 (4 x analog switch on-off, dwukierunkowy, osobne wejścia sterujące); 74HC1G66 (1x analog switch on-off dwukierunkowy); DG212B (4 x analog switch on-off, dwukierunkowy, osobne wejścia sterujące);

Medium transmisyjne

Mianem medium transmisyjnego określa się nośnik informacji służący do łączności pomiędzy systemami. Wyróżnić można następujące media transmisyjne:

- przewód elektryczny, m.in.:
 - zwykły przewód, na ogół wielożyłowy (tzw "taśmy", przewody telefoniczne)
 - kabel typu skrętka (także w różnych wariantach ekranowania)
 - kabel współosiowy (koncentryczny)
 - kabel symetryczny "twin-lead"
- światłowodów:
 - wielomodowy
 - jednomodowy
- bezprzewodowe (zarówno radiowe jak i optyczne):
 - dookolne (izotropowe)
 - kierunkowe (w szczególności optyczne linie laserowe)

W przypadku przewodów typu skrętka stosowana jest transmisja różnicowa to znaczy informacja kodowana jest w różnicy napięć pomiędzy dwoma przewodami wchodzącymi w skład pojedynczej skręconej pary. Żaden z przewodów wchodzących w skład pary nie jest powiązany z potencjałem masy, nie ma też znaczenia różnica potencjału pojedynczego przewodu w stosunku co do masy (ważna jest tylko wzajemna różnica potencjałów), sygnał nadawany w jednym z nich jest w przeciw-fazie do sygnału nadawanego w drugim. Służy to eliminacji zakłóceń elektromagnetycznych (zakłócenie indukcyjne się w obu przewodach jednakowo, więc różnica nie ulega zmianie) oraz zakłóceń wzajemnych, przesłuchów (sumaryczny prąd płynący parą jest bliski zeru). Typowo linie takie korzystają z dedykowanych układów nadawczo-odbiorczych opartych na zasadzie wzmacniacza różnicowego.

Połączenia światłowodowe charakteryzują się m.in.:

- pełną separacją galwaniczną (opto-izolacją) pomiędzy łączonymi systemami
- bardzo dużą odpornością na zakłócenia
- dużą przepustowością i dużą prędkością transmisji
- możliwością bezpośredniej (bez stosowania wzmacniaczy umieszczonych w środku linii) transmisji na bardzo duże odległości

Topologia połączeń i typy transmisji

Obok przedstawione zostały główne topologie połączeń:

linear bus (magistrala)

wszystkie urządzenia są podłączone do jednej linii (wspólnego medium transmisyjnego), okablowanie nie wyróżnia punktu centralnego

daisy chain

struktura okablowania podobna jak w magistrali, ale medium transmisyjne jest podzielone (połączenie n urządzeń składa się z n-1 łącz punkt-punkt pomiędzy urządzeniami)

ring (pierścień)

topologia daisy chain w której końce są połączone, odporna to na pojedyncze uszkodzenie

star (gwiazda)

wszystkie podłączenia biorą początek w węźle centralnym, w zależności od konstrukcji węzła centralnego może być realizowana w oparciu o współdzielone medium lub połączenia punkt-punkt

mesh (krata)

każde urządzenie ma bezpośrednie połączenie punkt-punkt do każdego innego urządzenia (połączenie n urządzeń wymaga $n(n-1)/2$ połączeń punkt-punkt)

Ponadto występują topologie mieszane złożone z opisanych powyżej: gwiazda wielokrotna (tzn. taka gdzie niektóre z węzłów stanowią punkty centralne kolejnych gwiazd), magistrala lub ring pomiędzy punktami centralnymi gwiazd, gwiazda w której w węzłach występują magistrale lub pierścienie, itd.

Wyróżnić można typy transmisji:

simplex

umożliwia tylko transmisję jednokierunkową

half-duplex

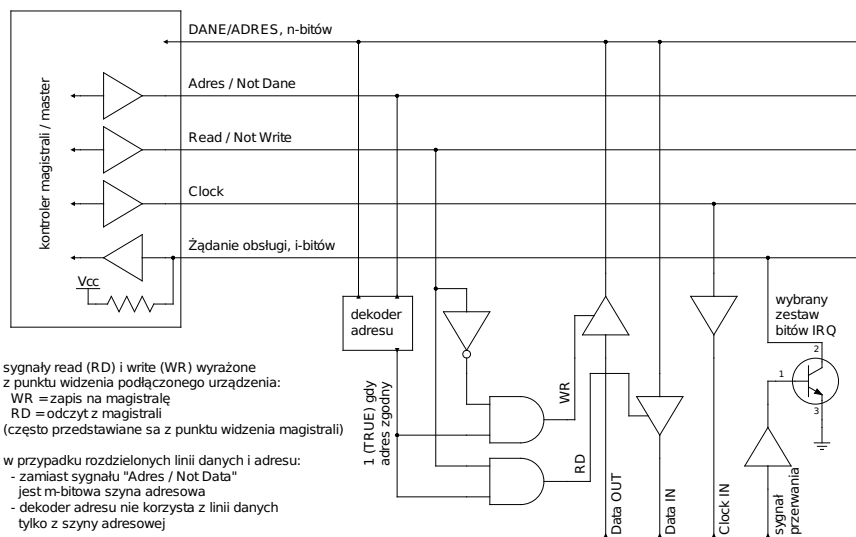
umożliwia transmisję dwukierunkową, ale tylko w jedną stronę równocześnie

full-duplex

umożliwia pełną transmisję dwukierunkową (równoczesne nadawanie i odbiór)

Magistrala równoległa

Magistrala równoległa jest zespołem linii, wraz z układami nimi sterującymi, umożliwiającym równoległe przesyłanie danych (w jednym czasie / takcie zegara na magistrali wystawiane / przesyłane jest całe n-bitowe słowo). Można wyróżnić szyny sterującą (kierunek przepływu, żądania obsługi, etc), adresową (adres układu który ma prawo nadawać i danych (przesyłane dane). Często szyna adresowa współdzieli linie transmisyjne z szyną danych. Do realizacji magistrali (celem umożliwiania podłączenia wielu układów) stosuje się zazwyczaj bufony trój-stanowe, a do zapewnienia współdzielonej szyny żądania obsługi (interrupt request) często układy typu open-collector.



Typowy układ realizacji magistrali half-duplex ze współdzielonymi liniami danych i adresu przedstawiony został na schemacie zamieszczonym obok.

Zadaniem dekodera adresu jest ustalenie czy wystawiony na magistrali adres (w trakcie wysokiego stanu linii "Adres / Not Dane") jest adresem danego urządzenia i zapamiętanie tej informacji do czasu wystawienia nowego adresu. Informacja ta jest wykorzystywana do sterowania dwukierunkowym buforem trój-stanowym (jako sygnał enable).

O kierunku działania bufora decyduje sygnał "Read / Not Write". Przy magistralach o ustalonym protokole transmisyjnym sterowanie kierunkiem może być zależne od wykonywanej komendy (po ustawieniu adresu urządzenie odczytuje z magistrali polecenie i w zależności od niego steruje kierunkiem bufora - odczytuje lub zapisuje dane na magistralę).

Zastosowanie kilku linii typu OC do odbierania żądań obsługi pozwala (na podstawie tego które z tych

linii znalazły się w stanie niskim na identyfikację urządzenia lub grupy urządzeń, z której niektóre zgłaszają żądanie obsługi.

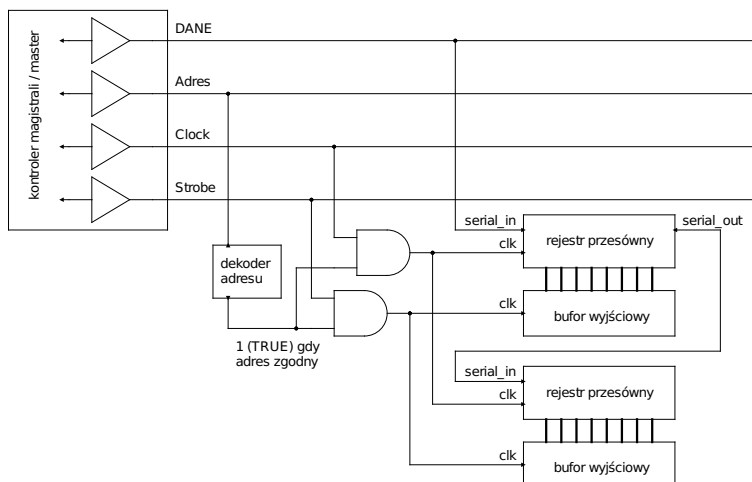
W przypadku prostych urządzeń wejścia / wyjścia zamiast bufora dwukierunkowego może być umieszczony np. jednokierunkowy bufor (lub n-bitowy rejestr) z wyjściami trój-stanowymi, który wystawia dane na magistralę w oparciu o sygnał zapisu na magistralę (WR) oraz zegar (clock) albo n-bitowy rejestr do którego zapisywane są dane z magistrali w oparciu o sygnał RD i Clock.

Magistrala szeregowo

W magistrali szeregowo dane przesyłane są bit po bicie w pojedynczej linii. Podobnie jak w magistrali równoległej oprócz linii danych mogą występować także linie sterujące. Prostą realizację magistrali szeregowo zapewniają rejestry przesuwne.

Przykładowy układ realizacji magistrali simplex (jednokierunkowej) z rozdzielonymi szynami danych i adresową został na schemacie zamieszczonym obok.

W prezentowanym przykładzie oprócz adresu master wystawia trzy sygnały - dane, zegar i strobe. Z każdym taktom zegara na linii danych wystawiany jest kolejny bit który jest wczytywany do zespołu rejestrów. Sygnał strobe służy do przepisania wartości z rejestrów przesuwnych do rejestrów wyjściowych, takie rozwiązanie zapobiega zmianom wyjść w trakcie przesyłania nowych danych poprzez szynę szeregowo, jest ono jednak opcjonalne.



W zależności od konstrukcji dekodera adresu szyna adresowa może być równoległa (w najprostszym przypadku - przez całą transmisję do danego urządzenia jego adres musi być wystawiony na szynie a dekodery jest układem bramek NOT i wielowejściowej bramki AND) lub szeregowo (w takim wypadku powinna posiadać własny zegar lub sygnał informujący o nadawaniu adresu z taktami zegara głównego, a dekodery powinien być wyposażony w rejestr przesuwny do odebrania i przechowywania aktualnego adresu z magistrali). Natomiast jeżeli magistrala byłaby oparta tylko na połączonych szeregowo rejestrach (wyjście serial-out do wejścia serial-in) to szyna adresowa nie jest potrzebna, ale konieczne może być każdorazowe wpisanie wszystkich wartości na szynę (czas zapisu rośnie z ilością podłączonych rejestrów).

Standardowe interfejsy

Istnieje wiele zstandaryzowanych interfejsów zarówno szeregowych jak i równoległych, wśród najważniejszych należy wymienić:

SPI (Serial Peripheral Interface)

jest to szeregowo magistrala full-duplex działająca w układzie master-slave złożona z linii zegarowej (SCLK), nadawania przez mastera (MOSI), odbioru przez mastera (MISO) oraz linii służących do aktywacji urządzenia slave (SS / CS).

I²C (TWI)

jest to szeregowo magistrala half-duplex złożona z linii sygnałowej (SDA) i zegara (SCL) posiadająca zdefiniowany format ramki wraz z adresowaniem. Z wyjątkiem bitu startu i stopu stan linii danych może zmieniać się tylko przy niskim stanie linii zegarowej. Nadajniki są typu open-drain przez co realizowany jest iloczyn na drucie, co pozwala na wykrywanie kolizji (jeżeli dany nadajnik nie nadaje zera a linia jest w stanie zera to nadaje także ktoś inny). Pozwala to także na uzyskanie układów multimaster, pomimo iż typowo na magistrali takiej występuje tylko jeden układ master (nadający sygnał zegara i inicjujący transmisję).

1 wire (one-wire)

jest to szeregowo magistrala half-duplex złożona z jedynie z linii sygnałowej (która może także służyć do zasilania urządzeń) posiadająca zdefiniowany format ramki wraz z adresowaniem. Standardowe nadawanie jest realizowane jako open-drain (wyjątkiem jest nadawanie tzw power-byte).

USART

jest to uniwersalny synchroniczny i asynchroniczny nadajnik i odbiornik, umożliwia realizację szeregowo transmisji synchronicznej (zgodnie z zegarem) lub asynchronicznej (wykrywanie początku ramki na podstawie linii danych). Interfejs korzysta z rozdzielonych linii nadajnika i odbiornika (wyjście danych TxD oraz wejście danych RxD, co umożliwia realizację transmisji full-duplex) oraz może korzystać z dodatkowych sygnałów sterujących (wyjście RTS informujące o

gotowości do odbioru oraz wejście CTS informacji o gotowości odbioru / zezwolenia na nadawanie). Niekiedy dostępne jest także wyjście załączenia nadajnika używane do pracy w trybie half-duplex (linie TxD i RxD połączone buforem trój-stanowym nadajnika).

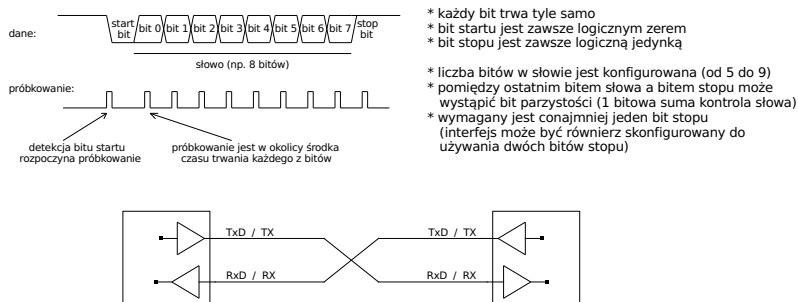
Interfejs ten najczęściej wykorzystywany jest w trybie asynchronicznym jako UART. W połączeniach UART zarówno nadajnik jak i odbiornik muszą mieć ustawione takie same parametry transmisji (szybkość, znaczenie 9 bitu (typowo bit parzystości, ale może także oznaczać np. pole adresowe), itp).

Głównymi standardami elektrycznymi dla UART są: poziomy napięć układów elektronicznych

używających tych portów (3.3V, 5V), RS-232 (w pełnym wariacie używa sygnałów kontroli przepływu, poziom logiczny

1 wynosi od -15V do -3V, a poziom logiczny 0 od +3V do +15V), RS-422 (transmisja różnicowa full-duplex pomiędzy dwoma urządzeniami) i RS-485 (transmisja różnicowa half-duplex w oparciu o szynę łączącą wiele urządzeń, kompatybilny elektrycznie z RS-422), możliwa jest też transmisja światłowodowa i bezprzewodowa.

Universal Asynchronous Receiver-Transmitter



CAN

jest to dwu-przewodowa magistrala multi-master działająca na zasadzie rozgłoszeniowej, obsługująca 11-to lub 29-bitowe identyfikatory komunikatów (nie są to adresy odbiorców), bazuje na niej wiele protokołów komunikacyjnych (np. CANopen).

Ethernet

jest to interfejs szeregowy przeznaczony dla sieci komputerowych, występuje w różnych odmianach od których zależy wykorzystywane medium transmisyjne oraz topologia połączeń. Posiada ustalony format ramki (niezależny od odmiany) wraz z adresacją (ramka zawiera zarówno MAC adres nadawcy, jak i odbiorcy).

W przypadku połączeń przewodowych (zarówno elektrycznych jak i optycznych) najczęściej stosowane są połączenia punkt-punkt w topologii gwiazdy, można się także spotkać z układem typu ring. Pętle w topologii fizycznej są zabronione i w przypadku ich wystąpienia konieczne jest stosowanie protokołu ich wykrywania i rozłączania (dotyczy to także połączenia typu ring).

Typowo połączenia fizyczne realizowane są z wykorzystaniem switchy odpowiedzialnych za przesyłanie ramek na odpowiednie porty w oparciu o adresy MAC. W tym celu switch pamięta z którego portu przychodzą ramki z danym adresem źródłowym i tam wysyła ramki z takim adresem docelowym, jeżeli switch nie posiada informacji gdzie wysłać daną ramkę wysyła ją na wszystkie swoje porty.

ISA

magistrala równoległa stworzona dla architektury PC (x86), początkowo z 8-bitową szyną danych i niezależną 20 bitową szyną adresową, następnie rozszerzona do 16 bitowej szyny danych i 24 bitowej szyny adresowej, wariacie 16 bitowym taktowana zegarem 8Mhz (wcześniej 4.77Mhz, a następnie 6Mhz).

W skład magistrali wchodzi m.in. następujące sygnały sterujące związane z szyną adresową i szyną danych:

- ALE - polecenie zatrzaśnięcia adresu / adres poprawny (wystawiane przez CPU)
- IOR / IOW - odczyt / zapis w przestrzeni IO (2 osobne sygnały wystawiane przez CPU)
- MEMR / MEMW - odczyt / zapis w przestrzeni pamięci (2 osobne sygnały, wystawiane przez CPU)
- IO CH RDY - rządanie opóźnienia w obsłudze odczytu / zapisu (wystawiane przez kartę)
- IOCS16 / MEMCS16 - informacja o obsłudze w trybie 16 bitowym (wystawiane przez kartę)
- DRQ*, DACK*, AEN, TC - sygnały DMA

Ponadto złącze magistrali zapewnia dostęp do 6 (w podstawowej 62 stykowej części złącza) + dodatkowych 5 (w rozszerzającej do 16 bitów 36 stykowej części złącza) linii przerwań IRQ*, sygnału MASTER pozwalającego na przejęcie sterowania nad systemem przez procesor

umieszczony na karcie oraz napięć zasilających +12V, -12V, +5V, -5V, kilku linii masy (GND) i zegara systemowego oraz zegara synchronizowanego z zegarem procesora.

Magistrala (w innej postaci złącza) obecna jest także w standardzie PC/104. Standard *EISA* określa rozszerzoną, kompatybilną w dół (także pod względem gniazda), wersję z 32 bitową szyną danych.

PCI

- magistrala równoległa występująca w odmianie 32 i 64 bitowej, kilka sygnałów związanych z arbitrażem prowadzonych jest do każdego gniazda / urządzenia PCI bezpośrednio z kontrolera magistrali,
- linie przerwań są przepieplatanie między slotami, specyfikacja dopuszcza też przerwania sygnalizowane w ramach komunikatu (przesyłanego szyną danych);
- stosowanie mostków PCI pozwala na rozgałęzianie / łączenie magistrali (zarówno szeregowo jak i równoległe);
- często spotykane są karty PCI o niepełnych wymiarach (prawie zawsze krótsze od maksymalnej dopuszczalnej długości, często także niższe), magistrala PCI jest także spotykana w ramach innych typów złącz - takich jak Mini-PCI, czy PCMCIA

PCI Express

- szeregowy interfejs różnicowy full-duplex do połączeń punkt-punkt, w zależności od wariantu złożony z 1, 4, 8 lub 16 zestawów par nadawania i odbioru;
- stosowanie mostków pozwala na rozgałęzianie / łączenie magistrali oraz np. podłączanie klasycznych magistral PCI;
- standardowe złącze zapewnia także m.in. sygnał zegara referencyjnego, resetu, sygnały SMBus (I2C);
- standard występuje na kilku rodzajach złącz (oprócz standardowego) także m.in. miniPCI Express, SATA Express, M.2, Thunderbolt / USB-C.

IDE, ATA, P-ATA

- wywodzący się z magistrali ISA standard służący zapewnieniu komunikacji z urządzeniami pamięci masowej takimi jak dyski twarde, czy napędy optyczne;
- określa także pełny protokół komunikacji z urządzeniami;
- stosowany także (z niewielkimi zmianami) w kartach Compact Flash.

SATA

- szeregowy interfejs różnicowy full-duplex do połączeń punkt-punkt, złożony z jednej pary nadawczej, jednej pary odbiorczej oraz linii GND;
- służący do podłączania urządzeń pamięci masowej takich jak dyski twarde, czy napędy optyczne;
- standard SATA określa także pełny protokół komunikacji z urządzeniami;
- standard występuje na kilku rodzajach złącz - oprócz standardowego także m.in. eSATA (zewnętrzne), mSATA, SATA Express, M.2.

USB

- szeregowy interfejs różnicowy half-duplex do połączeń punkt-punkt posiadający (w wersjach do 2.0 włącznie) jedną parę używaną zarówno do nadawania jak i odbioru;
- pozwala na tworzenie topologii gwiazdy z wykorzystaniem hub'ów USB;
- stosuje kilka rodzajów złącz oraz występuje w ramach złącz innych standardów (np. w M.2).

TMDS, HDMI, DVI

- TMDS jest standardem jednokierunkowej różnicowej transmisji szeregowej, stosowany w interfejsach HDMI i DVI-D;
- korzysta z osobnej pary różnicowej dla każdego koloru składowego RGB (lub YCbCr) oraz osobnej pary dla danych zegarowych;
- pomocniczo (dla dwukierunkowego przekazywania parametrów kontrolnych/sterujących związanych z trybem wyświetlania) wykorzystywany jest także dwuprzewodowy interfejs DDC oparty na I2C;
- dane w kanałach TMDS przesyłane są w postaci (zakodowanych przy pomocy 8b/10b) 10 bitowych ciągów, które mogą być interpretowane jako 8 bitowa wartość koloru, 2 bitowa wartość sterująca (na kanale zerowym wykorzystywane do synchronizacji poziomej i pionowej);
- złącze DVI pozwala na przesyłanie dwóch kanałów cyfrowego wideo (6 kanałów TMDS) oraz przesyłanie analogowego sygnału VGA (RGB, Hsync, Vsync), standard DVI zakłada wykorzystanie łącz TMDS do przesyłania wyłącznie obrazu RGB;

- standard HDMI pozwala na przesyłanie z użyciem TMDS obrazu RGB albo YCbCr, ponadto standard HDMI korzystając z bitów sterujących transmisji TMDS i kodowania 4b/10 pozwala na przesyłanie poprzez TMDS dźwięku oraz danych sterujących CEC, ponadto złącze HDMI pozwala na przesyłanie (osobną parą) zwrotnego sygnału audio lub sygnału Ethernetowego;

DisplayPort

- interfejs złożony z 1 do 4 szybkich jednokierunkowych kanałów różnicowej transmisji pakietowej oraz 1 dwukierunkowego kanału pomocniczego (także z transmisją różnicową)
- wersja "Dual-mode" (DisplayPort++) pozwala na przesyłanie sygnałów zgodnych z HDMI / jednokanałowym DVI-D
- występuje także w złączu Thunderbolt / USB-C

Rezystory terminujące

Niektóre ze standardów interfejsów komunikacyjnych przewidują kończenie swoich magistral rezystorem terminującym. Zastosowanie takiego rezystora ma na celu eliminację odbić sygnału, które mogłyby powstać na końcu linii transmisyjnej.

Zjawisko to występuje w przypadku *linii długich*, czyli takich których długość jest zbliżona lub większa od długości fali związanej z przesyłanym sygnałem. Jeżeli rozważymy np. impuls o czasie trwania 1µs to zajmie on na kablu odcinek o długości około 200m (zależy to od prędkości rozchodzenia się fali elektromagnetycznej w ośrodku który stanowi dany przewód). Zatem dla sygnału 1MHz (czyli takiego gdzie pojedyncze impulsy są właśnie długości 1µs) przewód o długości kilkuset metrów będzie linią długą.

Odbicia te wynikają z faktu, iż przemieszczanie się sygnału (np. naszego impulsu 5V o czasie trwania 1µs) wzdłuż przewodu związane jest z ładowaniem kolejnych pojemności pasożytniczych, związanych z odcinkiem przewodu do którego dociera sygnał. Dzieje się to kosztem rozładowania pojemności odcinka przewodu który sygnał już opuścił.

W momencie gdy sygnał trafia na koniec przewodnika nie ma możliwości rozładowania tej pojemności na kolejny odcinek przewodu, więc ładunek z nią związany „rozplywa się po kablu” powodując powstanie odbicia. Odbicie takie (biegnące od końca przewodu w stronę nadajnika) nakłada się na kolejne impulsy naszego sygnału (biegnące od nadajnika) i powoduje zakłócenia w ich odbiorze (interpretacji).

Zastosowanie odpowiedniego rezystora na końcu linii pozwala na rozładowanie tej pojemności (tak jakby był tam kolejny nieskończony odcinek przewodu) i eliminację odbicia. Wartość tej rezystancji jest charakterystyczna dla danego przewodu i określana przez parametr nazywany *impedancją falową*.

Rezystor terminujący stanowi obciążenie dla nadajnika i powinien on być stosowany tylko na końcach magistrali, czyli np. na ostatnich urządzeniach podłączonych do magistrali (a nie przy każdym urządzeniu do niej włączonym).

Jeżeli długość linii jest dużo mniejsza (dla sygnałów prostokątnych przyjmuje się że około 20-40 razy) od długości odcinka jaką zajmuje pojedynczy impuls (np. linia długości 3m, dla przykładowego sygnału 1MHz) to nie ma potrzeby stosowania rezystorów terminujących (często nawet gdy w ogólności dany standard je przewiduje), gdyż stan całej linii jest spójny i wymuszany przez nadajnik (nie jest to przypadek linii długiej).

Standard I2C nie przewiduje rezystorów terminujących (i nie powinny być w nim używane, zwłaszcza że są to linie open drain i powstawałby dzielnik z rezystorem pull-up). Wynika to z tego iż przy maksymalnej prędkości tego interfejsu za linie długie należałoby uznać odcinki co najmniej kilkunastometrowe, a z innych względów standard ten posiada ograniczenie do kilku metrów.

Standard RS-485 przewiduje stosowanie rezystorów terminujących 120 Ω, jednak w przypadku krótkich połączeń i/lub małych prędkości transmisji mogą one być pominięte.

Układy programowalne

Układy programowalne można podzielić na dwie podstawowe grupy, czyli układy z programowalną strukturą logiczną oraz systemy procesorowe.

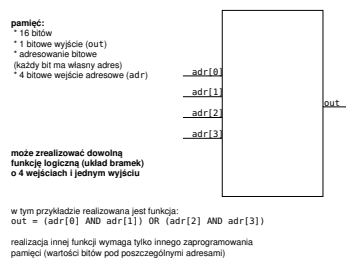
układy o programowalnej strukturze (PLD)

Układy z programowalną strukturą logiczną, oparte są na tym że wewnątrz takiego układu programujemy jakiś układ bramek logicznych, przerzutników i tym podobnych elementów oraz ich połączeń.

Do programowania tego typu układów służą języki opisu sprzętu typu HDL (hardware description language). Najczęściej jest to VHDL lub Verilog i zamiast wykonywanego kodu opisuje strukturę układu

logicznego (połączenia bramek, tablice prawdy, etc), która następnie jest programowana w fizycznej kości. Pozwala to na zaprogramowanie jakiegoś algorytmu, który realizowany będzie czysto sprzętowo. Taka realizacja sprzętowa, dzięki urownolegleniu wielu procesów jest zazwyczaj znacznie szybsza w działaniu niż wersja programowa.

Najprostszym koncepcyjnie sposobem realizacji czegoś takiego jest układ pamięciowy, który pozwala na zrealizowanie dowolnej funkcji logicznej, czyli dowolnego układu bramek. Jeżeli weźmiemy pamięć która będzie miała 2^n bitów i będzie adresowana n-bitową szyną adresową, to z każdym adresem związany jest jakiś jeden bit i każdy bit odpowiada jednemu adresowi. Pamięć tego typu pozwala na zapisanie tabeli prawdy dowolnej funkcji która posiada n wejść i jedno wyjście – każdemu wejściu odpowiada jeden bit adresu, a związana z daną kombinacją wejść, wartość wyjścia zapisana jest w tej pamięci.



Układy tego typu mają spore zastosowanie praktyczne i pozwalają na konstruowanie układów działających szybciej niż układy procesorowe. Zastosowanie programowalnych układów logicznych jest prostsze i szybsze niż konstruowanie takich rozwiązań z pojedynczych elementów takich jak bramki, a przy produkcji na małą i średnią skalę także tańsze od projektowania i produkcji dedykowanych układów scalonych. Pozwala też na aktualizację takiego „sprzętu” poprzez zaprogramowanie poprawionej jego wersji.

Do kategorii tej zaliczają się układy typu:

- SPLD
 - PLE - programowalna matryca bramek OR
 - PAL i GAL - programowalna matryca AND z dodatkowymi bramkami OR (często także obudowana rejestrami i multiplexerami na wyjściach)
 - PLA - programowalne matryce AND i OR
- CPLD
- FPGA - programowalny element pamięciowy (możliwość zdefiniowania dowolnej - na ogół 4 wejściowej - funkcji w każdym elemencie logicznym, programowalne połączenia między elementami logicznymi i pinami, itd)

systemy procesorowe

Układy procesorowe to systemy wykonujące ciąg instrukcji pobieranych z jakiejś pamięci. Składają się one z procesora, pamięci programu i pamięci danych. Procesor odpowiedzialny jest za interpretację i wykonywanie kolejnych instrukcji, pamięć przechowuje instrukcje i dane. W zależności od stosowanej architektury kod programu, a raczej kod maszynowy utworzony w wyniku kompilacji kodu programu, może znajdować się albo w tej samej pamięci w której przechowujemy również dane albo może mieć osobną, wydzieloną pamięć.

Systemami procesorowymi są zarówno systemy komputerowe, takie jak PCty czy laptopy, są nimi telefony komórkowe, zarówno smartfony jak i te bardziej prymitywne, są nimi duże systemy obliczeniowe, a także różnego rodzaju mikrokontrolery znajdujące się w sterownikach przemysłowych, centralkach alarmowych, telewizorach, sprzęcie AGD i tak dalej.

Procesor pracuje w cyklach rozkazowych, w ramach których przetwarza pojedynczą instrukcję. Cykl taki może trwać od 1 do kilku lub więcej cykli zegarowych i typowo składa się z następujących kroków:

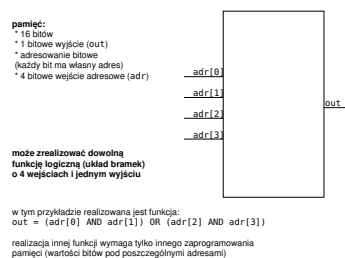
1. pobranie instrukcji z pamięci - realizowane jest poprzez wystawienie na szynę adresową zawartości *licznika programu* (zawierające adres instrukcji do wykonania) oraz wygenerowanie cyklu odczytu z pamięci, po wykonaniu odczytu danych następuje ich zapamiętanie w *rejestrze instrukcji* oraz zwiększenie wartości *licznika programu* o jeden; (zawartość rejestru *licznika programu* po resecie procesora określa skąd pobierana będzie pierwsza instrukcja, pod takim adresem zazwyczaj umieszczana jest jakaś pamięć typu ROM lub flash)
2. dekodowanie instrukcji - układ dekodera (np. oparty o PLA) dokonuje zdekodowania instrukcji

znajdującej się w *rejestrze instrukcji* i konfiguracji procesora w zależności od jej kodu i (opcjonalnie) jej argumentów; może to być np.:

- odpowiednie ustawienie multiplekserów pomiędzy rejestrami a jednostką ALU oraz wystawienie odpowiedniego kod operacji dla ALU (celem wykonania operacji arytmetycznej na wartościach rejestrów)
- wystawienie zawartości wskazanego rejestru na szynę adresową, podłączenie wskazanego rejestru do szyny danych oraz skonfigurowanie operacji odczytu/zapisu (celem wykonania odczytu lub zapisu wartości rejestru z/do pamięci)

3. wykonanie instrukcji - realizacja wcześniej zdekodowanej instrukcji zgodnie z ustawioną konfiguracją procesora

Instrukcje skoku polegają na załadowaniu nowej wartości do *licznika programu*, w przypadku skoków warunkowych jest to uzależnione od stanu *rejestru flag*, które ustawiane są w oparciu o wynik ostatniej operacji wykonywanej przez ALU.



Trochę uproszczony model działania procesora może wyglądać w sposób następujący: Procesor wystawia na szynę adresową magistrali umożliwiającą dostęp do pamięci adres będący zawartością tak zwanego licznika programu. Jest to adres instrukcji, która ma zostać pobrana i wykonana. Po wystawieniu tego adresu generowany jest cykl odczytu na tej szynie. A odczytany kod instrukcji zapamiętywany jest w rejestrze instrukcji, następuje też zwiększenie licznika programu o 1 po to żeby móc w następnym cyklu pobrać kolejną instrukcję.

Następnym krokiem jest dekodowanie instrukcji - znajdujący się w procesorze układ dekodera dokonuje zdekodowania instrukcji znajdującej się we wspomnianym rejestrze i konfiguracji procesora w zależności od jej kodu i opcjonalnie argumentów. Konfiguracja ta może polegać na ustawieniu odpowiednich multiplekserów, czyli właśnie takich przełączników elektronicznych pomiędzy rejestrami a jednostką wykonującą obliczenia (tak zwanym ALU - jednostką arytmetyczno-logiczną), wystawieniu odpowiedniego kodu operacji dla ALU (celem wykonania operacji na przykład na zawartości określonych rejestrów). Może to być także podpięcie wskazanego rejestru do szyny danych (celem skomunikowania go z pamięcią - wykonania odczytu lub zapisu do pamięci, bądź celem załadowania jego wartości do licznika programu aby wykonać skok) i tym podobne operacje. Ostatnim krokiem takiego cyklu jest wykonanie tych operacji, czyli wykonanie wcześniej zdekodowanej instrukcji, zgodnie z ustawioną konfiguracją procesora.

Przedstawiony model działania jest przykładowym i w rzeczywistym procesorze może to wyglądać odmiennie - np. długość instrukcji może być większa niż długość słowa używanego przez procesor / szerokość szyny danych co rozbudowuje fazę pobierania instrukcji z pamięci (instrukcja będzie ładowana w kilku fazach, gdyż cała instrukcja nie mieści się jednorazowo na szynie danych), mogą występować instrukcje bardziej złożone (np. operacje wykonywane z argumentem pobieranym z pamięci a nie rejestru), może także występować więcej faz (np. poprzez wydzielenie faz dostępu do pamięci, czy zapisywania wyników działania instrukcji).

Procesor może również działać potokowo, czyli fazy mogą się na siebie nakładać - podczas gdy jedna instrukcja jest wykonywana to kolejna może być już zdekodowana. Oczywiście wtedy procesor musi wykonywać pewne założenia co do prawdziwości skoków warunkowych, czyli na przykład wykonuje instrukcje skoku warunkowego, ale dekoduje już instrukcję która jest po niej, tak jakby skok miał nie zajść. W takiej sytuacji jeżeli skok zachodzi to wtedy ta zdekodowana instrukcja jest unieważniona, potok należy zapełnić od nowa i nie udaje się wykorzystać przyspieszenia związanego z wielopotokowością. Na takiej pracy potokowej opiera się tak zwany Hyper Threading dostępny w niektórych procesorach, jednak jak się przekonaliśmy taka praca potokowa nie jest równoważna z większą ilością niezależnych rdzeni.

Magistrala pomiędzy procesorem a pamięcią często jest magistralą równoległą, na której mogą się znaleźć też inne urządzenia wejść wyjść dostępne pod innymi adresami. W niektórych konstrukcjach mają one wydzieloną przestrzeń adresową, a w innych wspólną z pamięcią.

Mikrokontrolery

Przykładem systemu mikroprocesorowego, będącego jednocześnie przykładem układu typu system on chip, są mikrokontrolery. W jednym układzie scalonym zawierają one procesor, pamięć operacyjną RAM, pamięć stałą typu flash do przechowywania kodu programu (nie jest on w ich wypadku ładowany do RAMu, tylko wykonywany bezpośrednio z pamięci flash). Bardzo często w układzie tym znajdują się również różnego rodzaju układy wejścia-wyjścia, takie jak porty GPIO, które umożliwiają ustawienie lub odczyt stanu danej nóżki układu, mogą nimi być bardziej zaawansowane interfejsy jak porty szeregowy USART, I2C, SPI. Mogą to być także przetworniki analogowo-cyfrowe (umożliwiające pomiar wartości jakiegoś napięcia – sygnału analogowego) i cyfrowo analogowe (umożliwiające wystawienie napięcia o danej wartości).

Przykłady: ATmega8A (ADC (6-8ch), 3xTimer, I2C, USART, SPI, 28-32pin, 8kB Flash, 1kB RAM, 8bit, 16MHz), STM32F072V8 (1xADC (12ch), 1xDAC (2ch), 12xTimer, 2xI2C, 4xUSART, 2xSPI, CAN, USB, 100pin, 64kB Flash, 16kB RAM, 32bit, 48MHz), STM32F103VC (3xADC (21ch), 2xDAC, 11xTimer, 2xI2C, 5xUSART, 3xSPI, CAN, USB, 100pin, 256kB Flash, 48kB RAM, 32bit, 72MHz) i wiele innych

Programowanie

Zagadnienia są ilustrowane przykładowymi kodami w jednym lub kilku z pośród następujących języków: C++ (niekiedy z podziałem na podejście w stylu C i w stylu C++), Python, Bash, PHP i JavaScript. Wybór takiego zbioru języków wynika z odmiennych cech każdego z nich:

- C++ / C są najpopularniejszymi językami kompilowanymi do kodu maszynowego (a jeżeli traktować je łącznie to najpopularniejszymi językami w ogóle), pozwalają na stosowanie niskopoziomowych mechanizmów (łącznie z wstawkami assemblerowymi), są użyteczne do bezpośredniego programowania sprzętu (bez warstwy systemu operacyjnego) czy też tworzenia systemów operacyjnych
- Python - najpopularniejszy język skryptowy, wygodny (w miarę łatwy / szybki) w pisaniu i nauce, oferuje bardzo wiele gotowych bibliotek, pozwala na dość prostą integrację z kodem C/C++
- Bash - najpopularniejsza powłoka tekstowa, stanowiąca jednocześnie (specyficzny) język programowania
- PHP - popularny i wygodny język do tworzenia serwisów www
- JavaScript - najpopularniejszy język skryptowy dla WWW, element HTML5

Ponadto każdy z tych języków wykorzystywany jest w wielu projektach (od jądra Linuxa do systemu MediaWiki) i ich przynajmniej zgrubna znajomość może okazać się przydatna.

Język C++ oraz C są językami kompilowanymi to znaczy (po zmodyfikowaniu źródeł) przed uruchomieniem programu konieczne jest dokonanie tłumaczenia kodu źródłowego na kod maszynowy przy pomocy odpowiedniego programu (np. `clang++` lub `g++`). Kompilacja C i C++ przebiega kilku etapowo. W pierwszej kolejności wywoływany jest preprocesor, który jest odpowiedzialny za włączanie plików określonych poprzez `#include` (jest to literalne włączenie zawartości wskazanego pliku w danym miejscu, obsługę rozwijania stałych makr preprocesora (definiowanych z użyciem `#define`) oraz kompilację warunkową z wykorzystaniem poleceń takich jak `#ifdef` czy `#if`. Kompilatory pozwalają na uzyskanie nie tylko wynikowego pliku binarnego, ale także plików po przetworzeniu przez preprocesor czy też po konwersji na assembler.

Python, Bash i PHP są językami interpretowanymi, czyli uruchamianiu podlega bezpośrednio kod źródłowy który jest analizowany i wykonywany przez interpreter danego języka. Mogą one uruchamiać kod z podanego pliku lub interaktywnie wprowadzany w konsoli interpretera. Kod wprowadzany interaktywnie może być wykonywany bezpośrednio po jego wprowadzeniu (tzn. wprowadzeniu znaku nowej linii, a w przypadku bloków instrukcji - zakończenia takiego bloku). W Bashu i Pythonie jest to zachowanie domyślne, natomiast w przypadku PHP wymaga podania opcji `-a` (jej podanie zakłada że podawany kod jest tylko częścią ujętą w `<?php` i `?>`, bez tej opcji kod zostanie zinterpretowany i wykonany dopiero po wczytaniu znaku końca pliku, który można wprowadzić `Ctrl+D`).

Zarówno kompilatory jak i interpretery mogą generować błędy lub ostrzeżenia dotyczące przetwarzanego kodu, przy jego poprawianiu ważne jest ich czytanie z próbą zrozumienia (niekiedy potrafią być dziwne i odnosić się nie do tego co jest przyczyną problemu). Rozwiązując problem należy skupić się na pierwszym błędzie, gdyż kolejne mogą być jego wynikiem (jest to szczególnie ważne w przypadku kompilacji C++ / C, gdzie możemy uzyskać bardzo wiele błędów). Programy mogą także generować błędy typu "run time error" w trakcie działania, przykładami takich błędów są rzucane i nie obsługiwane wyjątki oraz błędy związane z odwołaniami do pamięci (np. przekroczeniem zakresu tablicy w C) komunikowane zazwyczaj jako "Segmentation fault".

Pamięć danych - zmienne

Wszelkie dane na których operuje program komputerowy przechowywane są w jakimś rodzaju pamięci - najczęściej jest to pamięć operacyjna. W pewnych sytuacjach niektóre dane mogą być przechowywane np. tylko w rejestrach procesora lub rejestrach urządzeń wejścia-wyjścia.

W programowaniu na poziomie wyższym od kodu maszynowego i asemblera używa się pojęcia zmiennej i (niemal zawsze) pozostawia kompilatorowi/interpretatorowi decyzję o tym gdzie ona jest przechowywana. Oczywiście wyjątkiem są grupy zmiennych, czy też buforów alokowane w sposób jawny w pamięci. Ze względu na ograniczoną liczbę rejestrów procesora większość zmiennych (w szczególności tych dłużej istniejących i większych) będzie znajdowała się w pamięci i będą przenoszone do rejestrów celem wykonania jakiejś operacji na nich po czym wynik będzie przenoszony do pamięci.

Z każdą zmienną przechowywaną w pamięci związany jest adres pamięci pod którym się ona znajduje. Niektóre z języków programowania pozwalają na odwoływanie się do niego poprzez wskaźnik na zmienną lub referencję do zmiennej (odwołania do adresu zmiennej mogą wymusić umieszczenie jej w pamięci nawet gdyby normalnie znajdowała się tylko w rejestrze procesora).

Wszystkie dane są zapisywane w postaci liczb lub ciągów liczb. Typ zmiennej (jawny lub nie) informuje o tym jakiej długości jest dana liczba i jak należy ją interpretować (jak należy interpretować ciąg liczb).

/// Przykład C++

```
int main() {
    // liczba całkowita ze znakiem
    int    liczbaA = -34;
    // liczba rzeczywista (pojedynczej precyzji)
    float  liczbaB = 673.1;
    // 8 bitowa liczba całkowita bez znaku
    uint8_t liczbaC = 0xf3;

    // zmienna napisowa "C NULL-end string"
    char*  napisA = "q we";
    // zmienna napisowa typu "C++ string"
    std::string napisB = "a bc";
}
```

Przykład Python

```
# dynamiczne typowanie - typ określany jest
# na podstawie wartości zapisywanej do zmiennej
```

```
zmiennaA = -91.7
zmiennaB = "qa z"
```

Przykład Bash

```
# dynamiczne typowanie - typ określany jest
# na podstawie wartości znajdującej się w zmiennej
#
# zasadniczo wszystkie zmienne są napisami, a interpretacja
# ma miejsce przy ich użyciu a nie przy tworzeniu

# obsługiwane liczby całkowite oraz napisy
# brak obsługi liczb zmiennoprzecinkowych
#
# brak spacji pomiędzy nazwą zmiennej a znakiem równości
# w operacji przypisania jest wymogiem składniowym

zmiennaA=-91
zmiennaB="qa  z"
zmiennaC=98.6 # to będzie traktowane jako napis a nie liczba

# Odwołanie do zmiennej odbywa się z użyciem znaku dolara,
# po którym występuje nazwa zmiennej. Nazwa może być ujęta
# w klamry. Rozwijaniu ulegają nazwy zmiennych znajdujące
# się w napisach umieszczonych w podwójnych cudzysłowach.

# Umieszczenie odwołania do zmiennej w podójnych
# cudzysłowach zabezpiecza białe znaki (spacje nowe linie)
# przy przekazywaniu do funkcji i programów (w tym przy
# przekazywaniu do echo, celem wypisywania).

echo $zmiennaA ${zmiennaA}AA
echo "$zmiennaA ${zmiennaA}AA"
echo '$zmiennaA ${zmiennaA}AA'

# Jeżeli chcemy aby zmienna była widoczna przez programy
# uruchamiane z naszej powłoki należy ją wyeksportować za
# pomocą polecenia `export zmienna` (nazwa bez znaku dolara).
```

/// Przykład PHP

```
<?php
```

```
# dynamiczne typowanie - typ określany jest
# na podstawie wartości zapisywanej do zmiennej

$zmiennaA = -91.7
$zmiennaB = "qa z"

?>
```

```
/// Przykład JavaScript
```

```
var a = 13.3
var b = "qa z"
```

Kod programu - instrukcje, ...

Program komputerowy składa się z instrukcji wykonywanych kolejno przez procesor. Instrukcje te można podzielić na:

- instrukcje przenoszenia danych (m.in. pomiędzy pamięcią i rejestrami procesora),
- instrukcje modyfikujące dane (wykonujące na nich jakieś operacje arytmetyczne, logiczne itp.),
- instrukcję modyfikującą sterowanie w programie.

Dwie pierwsze kategorie instrukcji odpowiadają za modyfikowanie wartości zmiennej na skutek przypisań wartości oraz operacji wykonywanych na zmiennych. Trzecia kategoria to instrukcje odpowiedzialne za warunkowe lub bez warunkowe modyfikowanie licznik programu. Mogą występować także instrukcje złożone realizujące zadania z różnych grup w ramach jednej instrukcji (np. instrukcje wykonywane warunkowo).

Podstawowe operacje arytmetyczne, bitowe i logiczne

Z punktu widzenia procesora (kodu maszynowego) podstawowymi operacjami wykonywanymi na danych (oprócz przenoszenia) są typowe operacje arytmetyczne i logiczne oraz operacje binarne. Wiele architektur posiada także bardziej złożone instrukcje (np. wykonywane nie na pojedynczych liczbach, a całych wektorach liczb lub też wykonujących jakąś bardziej złożoną operację matematyczną w sposób sprzętowy), które odgrywają istotną rolę w optymalizacji kodu na daną architekturę.

Z punktu widzenia języków programowania najczęściej za podstawowe operacje uważa się:

- operacje arytmetyczne: dodawanie, odejmowanie, mnożenie, dzielenie całkowite, dzielenie zmiennoprzecinkowe, obliczanie reszty z dzielenia, niekiedy podnoszenie do potęgi
- operacje logiczne: AND, OR, NOT, niekiedy XOR, porównania (równość, nierówność, większe, mniejsze, większe równe, mniejsze równe)
- operacje bitowe (wykonywane niezależnie na każdym bicie argumentów): AND, OR, NOT, XOR, przesunięcia bitowe

Dość istotnym wyjątkiem są niektóre języki skryptowe (takie jak bash), których głównym zadaniem jest uruchamianie innych programów i to jest operacją podstawową w tych językach.

```
/// Przykład C++
```

```
#include <stdio.h>
```

```
int main() {
    double a = 12.7, b = 3, c, d, e;
    int x = 5, y = 6, z;

    // dodawanie, mnożenie, odejmowanie zapisuje się
    // i działają one tak jak w normalnej matematyce:
    e = (a + b) * 4 - y;

    // dzielenie zależy od typów argumentów
    d = a / b; // będzie dzieleniem zmiennoprzecinkowym bo a i b są typu float
    c = x / y; // będzie dzieleniem całkowitym bo x i y są zmiennymi typu int
    b = (int)a / (int)b; // będzie dzieleniem całkowitym
    a = (double)x / (double)y; // będzie dzieleniem zmiennoprzecinkowym

    // reszta z dzielenia (tylko dla argumentów całkowitych)
    z = x % y;

    // wypisanie wyników
    printf("%d %f %f %f %f %f\n", z, e, d, c, b, a);

    // operacje logiczne:
    // ((a większe równe od 0) AND (b mniejsze od 2)) OR (z równe 5)
    z = (a >= 0 && b < 2) || z == 5;
    // negacja logiczna z
```

```

x = !z;

printf("%d %d\n", z, x);

// operacje binarne:
// bitowy OR 0x0f z 0x11 i przesunięcie wyniku o 1 w lewo
x = (0x0f | 0x11) << 1;
// bitowy XOR 0x0f z 0x11
y = (0x0f ^ 0x11);
// negacja bitowa wyniku bitowego AND 0xffff i 0x0f0
z = ~(0xffff & 0x0f0);

printf("%x %x %x\n", x, y, z);

// uwaga: powyższy program może nie wykonywać obliczeń w czasie działania
// ze względu na optymalizację i fakt iż wyniki wszystkich operacji
// są znane w momencie kompilacji programu
}

```

Przykład Python

```

a = 12.7
b = 3
x = 5
y = 6

# dodawanie, mnożenie, odejmowanie zapisuje się
# i działają one tak jak w normalnej matematyce:
e = (a + b) * 4 - y

# dzielenie zmiennoprzecinkowe
c = x / y

#dzielenie całkowite
b = a // b

# reszta z dzielenia
z = x % y;

# wypisanie wyników
print(e, c, b, z)

# operacje logiczne:
# ((a większe równe od 0) AND (b mniejsze od 2)) OR (z równe 5)
z = (a>=0 and b<2) or z == 5;
# negacja logiczna z
x = not z;

print(z, x);

# operacje binarne:
# bitowy OR 0x0f z 0x11 i przesunięcie wyniku o 1 w lewo
x = (0x0f | 0x11) << 1;
# bitowy XOR 0x0f z 0x11
y = (0x0f ^ 0x11);
# negacja bitowa wyniku bitowego AND 0xffff i 0x0f0
z = ~(0xffff & 0x0f0);

print(hex(x), hex(y), hex(z & 0xffff));
# wypisując z musimy określić jego bitowość

# wieloargumentowa operacja przypisania
# może być użyta do zamiany wartości pomiędzy dwoma zmiennymi
# bez jawnego używania zmiennej tymczasowej
print(a, b)
a, b = b, a
print(a, b)
# oczywiście można w jej ramach używać więcej niż dwóch zmiennych

```

Przykład Bash

```

a=12; b=3; x=5; y=6

# aby wykonać działania arytmetyczne należy
# umieścić je wewnątrz $( ( i ) )

# dodawanie, mnożenie, odejmowanie zapisuje się
# i działają one tak jak w normalnej matematyce:
e=$(( ($a + $b) * 4 - $y ))

```

```

# dzielenie całkowite
c=$(( $x / $y ))

# wypisanie wyników
# (zwróć uwagę na wynik wypisania niezainicjalizowanej zmiennej z)
echo $e $c $z

# operacje logiczne obsługiwane są komendą test
# lub operatorem [ ] wynik zwracany jest jako kod powrotu
# należy zwrócić uwagę na escapowanie odwrotnym ukośnikiem
# nawiasów i na to że spacje mają znaczenie

# ((a większe równe od zera) AND (b mniejsze od dwóch)) OR (z równe 5)
[ \(( $a -ge 0 -a $b -lt 2 \) ) -o $z -eq 5 ]; z=$?

# negację realizuje !, ale wynikiem negacją dowolnej liczby jest FALSE
# więc nie da się zanegować z jak w pozostałych przykładach

echo $z
# bash stosuje logikę odwróconą 0 == TRUE, coś nie zerowego to FALSE

# bash nie obsługuje liczb zmiennoprzecinkowych ani operacji bitowych
# nieobsługiwane operacje można wykonać za pomocą innego programu np:
a=`echo 'print(3/2)' | python3`
b=$(echo '3/2' | bc -l)
echo $a $b

# ujęcie polecenia w znaki ` powoduje podstawienie w tym miejscu
# standardowego wyjścia tego polecenia (w tym wypadku zapisania go
# do zmiennej), alternatywną składnią jest $( polecenie )
# pokazany na przykładzie zmiennej b

```

/// Przykład JavaScript

```

var a = 12.7, b = 3, c = 13, d, e;

// dodawanie, mnożenie, odejmowanie zapisuje się
// i działają one tak jak w normalnej matematyce:
e = (a + b) * 4 - c;

// dzielenie zawsze jest zmiennoprzecinkowe
d = a / b;
c = b / c;

// wypisanie wyników
console.log(
  "e=" + e + "\n" +
  "d=" + d + "\n" +
  "c=" + c + "\n"
);

// operacje logiczne:
// ((a większe równe od 0) AND (b mniejsze od 2)) OR (c równe 5)
var z = (a>=0 && b<2) || c == 5;
// negacja logiczna z
a = !z;

console.log(
  "z=" + z + "\n" +
  "a=" + a + "\n"
);

// operacje binarne:
// bitowy OR 0x0f z 0x11 i przesunięcie wyniku o 1 w lewo
a = (0x0f | 0x11) << 1;
// bitowy XOR 0x0f z 0x11
b = (0x0f ^ 0x11);
// negacja bitowa wyniku bitowego AND 0xffff i 0x0f0
c = ~(0xffff & 0x0f0);

console.log(
  "a=0x" + a.toString(16) + "\n" +
  "b=0x" + b.toString(16) + "\n" +
  "c=0x" + (c & 0xffff).toString(16) + "\n"
);

```

Przepływ sterowania w programie - skoki, warunki, pętle, funkcje

Licznik programu (program counter, instruction pointer lub instruction address register) jest rejestrem procesora który określa adres następnej (w niektórych architekturach aktualnej) instrukcji która ma zostać przetworzona procesor.

Skoki bezwarunkowe, instrukcje warunkowe, pętle, wywołania funkcji są realizowane poprzez modyfikację licznika programu. W przypadku wywołań funkcji dodatkowo wykonywane są operacje związane z obsługą stosu (zachowywaniem stanu rejestrów, umieszczaniem argumentów na stosie, ...). Instrukcja *goto* (realizująca skok bezwarunkowy) jest pełnoprawną instrukcją skoku, jedyną wadą jej stosowania jest to że przy niewłaściwym / zbyt częstym wykorzystywaniu (zamiast wywołań funkcji, warunków i pętli) kod programu staje się mniej czytelny.

W większości przypadków pętle realizowane są na poziomie kodu maszynowego jako zestaw instrukcji (np. inkrementacji zmiennej, sprawdzania warunku, skoku), jednak w niektórych rozwiązaniach pętle (np. typu "powtórz n razy") mogą być realizowane sprzętowo przy pomocy pojedynczej instrukcji.

```
/// Przykład C++
#include <stdio.h>

int main() {
    int i, j, k;

    // instrukcja warunkowa if - else
    if (i<j) {
        puts("i<j");
    } else if (j<k) {
        puts("i>=j AND j<k");
    } else {
        puts("i>=j AND j>=k");
    }

    // podstawowe operatory logiczne
    if (i<j || j<k)
        puts("i<j OR j<k");
    // innymi operatorami logicznymi są && (AND), ! (NOT)

    // pętla for
    for (i=2; i<=9; ++i) {
        if (i==3) {
            // pominięcie tego kroku pętli
            continue;
        }
        if (i==7) {
            // wyjście z pętli
            break;
        }
        printf(" a: %d\n", i);
    }

    // pętla while
    while (i>0) {
        printf(" b: %d\n", --i);
    }

    // pętla do - while
    do {
        printf(" c: %d\n", ++i);
    } while (i<2);

    // instrukcja wyboru switch
    switch(i) {
        case 1:
            puts("i==1");
            break;
        default:
            puts("i!=1");
            break;
    }

    goto ETYKIETA;
    puts("to się nigdy nie wykona");
    puts("bo wcześniej robimy bezwarunkowe goto");

    ETYKIETA:
```

```
    puts("a to się wykona");
}
```

Przykład Python

```
i, k, j = 0, 0, 0 # wielokrotne przypisanie
# najpierw oblicza wartości wyrażeń po prawej,
# potem przypisuje. Pozwala na a, b = b, a
# celem zamiany wartości zmiennych

# instrukcja warunkowa if - else
if i<j :
    print("i<j")
elif j<k :
    print("i>=j AND j<k")
else:
    print("i>=j AND j>=k")

# podstawowe operatory logiczne
if i<j or j<k:
    print("i<j OR j<k")
# innymi operatorami logicznymi są `and` oraz `not`

# pętla for
for i in range(2, 9):
    if i==3:
        # pominięcie tego kroku pętli
        continue;
    if i==7:
        # wyjście z pętli
        break;
    print(" a:", i)
else:
    # nie wejdzie tutaj bo pętla kończy
    # się z użyciem break
    print("else w pętli")

# pętla while
while i>0 :
    i = i - 1;
    print(" b:", i)
else:
    # wejdzie tu gdy warunek i>0
    # nie będzie spełniony
    # zarówno przy pierwszym
    # jak i kolejnych sprawdzeniach
    print("else w pętli")
```

Przykład Bash

```
# pętla for
for (( i=0 ; $i<=20 ; i++ )) ; do
    echo $i;
done

# w bardziej "shellowym" stylu:
for i in `seq 0 20`; do
    echo $i;
done

# pętla for po liście rozdzielanej spacjami
# wypisanie nazw wszystkich plików z /tmp
for nazwa in /tmp/* ; do
    echo $nazwa;
done

# pętla while z read (po liniach pliku)
cat /etc/fstab | while read słowo reszta; do
    echo $reszta;
done
# powyższa pętla wypisze po kolei wszystkie
# wiersze pliku przekazanego przez stdin
# (cat nazwa_pliku |) z pominięciem
# pierwszego słowa (które wczytywane było do
# zmiennej słowo)
# domyślnym separatorem pól dla komendy read
# jest dowolny ciąg białych znaków (spacji i
# tabulatorów) można go zmienić przy pomocy
```

```

# zmiennej IFS:
IFS=:
while read a b c d; do
    echo $c
done < /etc/passwd
unset IFS

# Należy mieć na uwadze, że w konstrukcjach typu
# while read, pętla while uruchamiana może być w
# procesie potomnym obecnej powłoki.
# Efektem tego jest iż w niektórych przypadkach
# wykonywane modyfikacje zmiennych wewnątrz
# takiej pętli nie będą widoczne poza nią.

# instrukcja if - else
if [ "$xx" = "kot" -o "$xx" = "pies" ]; then
    echo "kot lub pies";
elif [ "$xx" = "ryba" ]; then
    echo "ryba"
else
    echo "coś innego"
fi

# spacje wokół i wewnątrz nawiasów kwadratowych
# przy warunku są istotne składniowo, zawartość
# nawiasów kwadratowych to tak naprawdę
# argumenty dla komendy test, zatem wywołanie:
# if [ "$xx" = "ryba" ]; then
# jest równoważne:
# if test "$xx" = "ryba"; then
# a więcej na temat warunków można znaleźć w:
# man test

# jako warunek może wystąpić dowolne polecenie
# wtedy sprawdzany jest jego kod powrotu
# 0 oznacza prawdę / zakończenie sukcesem
# wartość nie zerowa fałsz / błąd
if grep '^root:' /etc/passwd > /dev/null; then
    echo /etc/passwd zawiera użytkownika root;
fi

# istnieje możliwość skróconego zapisu warunków
# z użyciem łączenia instrukcji przy pomocy:
# && wykonaj instrukcję występującą po prawej
# gdy poprzednia zwróciła zero (true)
# || (wykonaj instrukcję występującą po prawej
# gdy poprzednia zwróciła nie zero -- false
[ "$xx" = "ryba" ] && echo '$xx = to ryba'
grep '^root:' /etc/passwd > /dev/null && \
    echo /etc/passwd zawiera użytkownika root;

# instrukcja case
# (w odróżnieniu od switch z C obsługuje napisy)
case $xx in
    kot | pies)
        echo "kot lub pies"
        ;;
    ryba)
        echo "ryba"
        ;;
    *)
        echo "cos innego"
        ;;
esac

```

/// Przykład PHP

```
<?php
```

```
$i=$j=$k=0;
```

```

// instrukcja warunkowa if - else
if ($i<$j) {
    echo("i<j\n");
} else if ($j<$k) {
    echo("i>=j AND j<k\n");
} else {

```

```

    echo("i>=j AND j>=k\n");
}

// pętla for
for ($i=2; $i<7; ++$i) {
    if ($i==$j) {
        // pominięcie tego kroku pętli
        continue;
    }
    printf(" a: %d\n", $i);
}

// pętla while
while ($i>0) {
    if ($i==$k) {
        // wyjście z pętli
        break;
    }
    printf(" b: %d\n", --$i);
}

// pętla do - while
do {
    printf(" c: %d\n", ++$i);
} while ($i<2);

// instrukcja wyboru switch
switch($i) {
    case 1:
        echo("i==1\n");
        break;
    default:
        echo("i!=1\n");
        break;
}

?>

```

/// Przykład JavaScript

```

var i, j, k;

// instrukcja warunkowa if - else
if (i<j) {
    console.log("i<j\n");
} else if (j<k) {
    console.log("i>=j AND j<k\n");
} else {
    console.log("i>=j AND j>=k\n");
}

// podstawowe operatory logiczne
if (i<j || j<k)
    console.log("i<j OR j<k\n");
// innymi operatorami logicznymi są && (AND), ! (NOT)

// pętla for
for (i=2; i<=9; ++i) {
    if (i==3) {
        // pominięcie tego kroku pętli
        continue;
    }
    if (i==7) {
        // wyjście z pętli
        break;
    }
    console.log(" a: ", + i + "\n");
}

// pętla while
while (i>0) {
    console.log(" b: " + --i + "\n");
}

// pętla do - while
do {
    console.log(" c: " + ++i + "\n");
} while (i<2);

```

```
// instrukcja wyboru switch
switch(i) {
    case 1:
        console.log("i==1\n");
        break;
    default:
        console.log("i!=1\n");
        break;
}
```

Typowo funkcja przyjmuje określoną ilość argumentów (może ich nie posiadać) i zwraca pojedynczą wartość. Istnieje możliwość zwracania przez funkcję struktury złożonej z kilku wartości, jednak na ogół odebranie z funkcji dodatkowych wyników działania odbywa się poprzez argument typu wskaźnikowego, wskazujący na zmienną w której mają zostać zapisane. Wiele języków pozwala także na definiowanie funkcji przyjmujących dowolną ilość argumentów.

```
/// Przykład C++
#include <stdio.h>

#include <stdarg.h>
// potrzebne dla obsługi dowolnej ilości argumentów

// funkcja bezargumentowa niezwracająca wartości
void f1() {
    puts("ABC");
}

// funkcja dwuargumentowa zwracająca wartość
int f2(int a, int b) {
    return a*2.5 + b;
}

// funkcja z jednym argumentem obowiązkowym
// i jednym opcjonalnym
float f3(int a, int b=1) {
    puts("F3");
    return a*2.5 + b;
}

// funkcja z dwoma argumentami wymaganymi
// i dowolną ilością argumentów opcjonalnych
float f4(int a, int b, ...) {
    float ret;

    va_list vl;
    va_start(vl, b);

    // w tym miejscu potrzebujemy znać ilość
    // oraz typy argumentów
    for (int i=0; i<a; i++) {
        ret += b * va_arg(vl, double);
    }
    va_end(vl);

    return ret;
}

int main() {
    f1();

    int a = f2(3, 6);
    // zwracaną wartość można wykorzystać
    // (jak wyżej) lub zignorować:
    f3(0);

    float b = f4(2, 1, 2.8, 3.5);

    printf("%d %f\n", a, b);
}
```

Przykład Python

```
# funkcja bezargumentowa, zwracająca wartość
def f1():
```

```

    print("AA")
    return 5

a = f1()
print(a)

# funkcja przyjmująca jeden obowiązkowy
# argument oraz dwa opcjonalne
def f2(a, b=2, c=0):
    print(a*b+c)

f2(3)
f2(3, 3)
# można pominąć dowolne z argumentów z wartością
# domyślną odwołując się do pozostałych nazwami
f2(2, c=1)
# można podawać argumenty w dowolnej kolejności
# odwołując się do nich nazwami
f2(b=3, a=2)

# nieokreślona ilość argumentów pozycyjnych
def f(*a):
    for aa in a:
        print(aa)

f(1, "y", 6)
# ale nie: f(1, "y", u="p")

# nieokreślona ilość argumentów nazwanych
def f(**a):
    for aa in a:
        print(aa, "=", a[aa])

f(a="y", u="p")
# ale nie: f(1, u="p")

# nieokreślona ilość argumentów pozycyjnych i nazwanych
def f(*a1, **a2):
    print(a1)
    print(a2)

f(1, "y", 6)
f(a="y", u="p")
f(1, "y", u="p")

# można też wymusić ileś argumentów jawnych
def f(x, *a1, y="8", **a2):
    print(x, y)
    print(a1)
    print(a2)

f(1, "y", 6)
f(1, "y", u="p")
f(1, "z", y="y", u="p")
# ale nie: f(a="y", u="p")

```

Przykład Bash

```

# w bashu każda funkcja może przyjmować
# dowolną ilość parametrów pozycyjnych
# (w identyczny sposób obsługiwane są
# argumenty linii poleceń dla całego skryptu)
f1() {
    echo "wywołano z $# parametrami"
    echo "parametry to: $@"

    [ $# -lt 2 ] && return;

    # można odwoływać się do pojedynczych parametrów
    echo "drugi: $2"
    echo "pierwszy: $1"

    # albo kolejnych w pętli
    for a in "$@"; do echo $a; done

    # lub z użyciem polecenia shift
    for i in `seq 1 $#`; do

```

```

    echo $1
    shift # powoduje zapomnienie $1
           # i przenumowanie pozostałych
           # argumentów pozycyjnych o 1
           # wpływa na wartości $@ $# itp
done

# funkcja może zwracać tylko wartość numeryczną
# tzw kod powrotu
return 83
}

# wywołanie - tak jak komendy, czyli bez nawiasów,
# a argumenty rozdzielane białym znakiem (spacją)
f1 aaa 3 t 56

# kod powrotu ostatnio wywołanej komendy lub funkcji
# uzyskuje się poprzez $?
echo $?

# często funkcje (tak jak wiele komend) wynik swojego
# działania zwracają poprzez standardowe wyjście

f2() { echo "Uwolnić mrożone truskawki$1"; }

# pozyskać go można poprzez `` lub $( )
# zapis z użyciem $( ) może być zagnieżdżony

a=`f2 '!!!'`
echo $a

b=$(f2 '!')
echo $b

# w jednolinijkowym zapisie definicji funkcji (jak
# miało to miejsce dla f2) spacje po { i przed } są
# obowiązkowe, podobnie jak średniki po instrukcjach

```

/// Przykład PHP

<?php

```

# funkcja bezargumentowa, zwracająca wartość
function f1() {
    print("AA\n");
    return 5;
}

$a = f1();
print($a);
print("\n");

# funkcja przyjmująca jeden obowiązkowy
# argument oraz dwa opcjonalne
function f2($a, $b=2, $c=0) {
    print($a**$b+$c);
    print("\n");
}

f2(3);
f2(3, 3);

# nieokreślona ilość argumentów pozycyjnych
function f3() {
    $num = func_num_args();
    print("ilość argumentów:\n");
    print( $num );
    print("\n");
    for($i=0; $i<$num; ++$i) {
        print("argument $i: ");
        print(func_get_arg($i));
        print("\n");
    }
}

f3();

```

```
f3("a", 1, "c", 5);

# jeden wymagany, jeden opcjonalny
# i nieokreślona ilość dodatkowych
function f4($a, $b="16") {
    # tablica z wszystkimi argumentami
    # jest alternatywną metodą dostępu
    # do wszystkich argumentów w stosunku
    # co do pokazanej w f3()
    $args = func_get_args();
    print_r($args);
}

f4("a", 1, "c", 5);

?>
```

```
/// Przykład JavaScript

function f1() {
    console.log("ABC\n");
}

// funkcja dwuargumentowa zwracająca wartość
function f2(a, b) {
    return a*2.5 + b;
}

// funkcja z jednym argumentem obowiązkowym
// i jednym opcjonalnym
function f3(a, b=1) {
    console.log("F3\n");
    return a*2.5 + b;
}

// funkcja z dowolną ilością argumentów
function f4() {
    var ret = 0;

    for(var i=0; i<arguments.length; i++) {
        ret += arguments[i];
    }

    return ret;
}

f1();
// zwracaną wartość można wykorzystać:
var a = f2(3, 6);
// lub zignorować:
f3(0);
var b = f4(2, 1, 2.8, 3.5);

console.log("a=" + a + " b=" + b + "\n");
```

Punkt startu

Jako że program komputerowy jest sekwencją wykonywanych instrukcji musi rozpoczynać się od określonego miejsca.

W przypadku języków interpretowanych jest to na ogół początek pliku z kodem programu. Często w pierwszej linii znajduje się specjalny komentarz (postaci `#!/ścieżka/do/programu`) stanowiący informację dla programu uruchamiającego skrypt jakiego interpretera ma użyć w celu wykonania kodu zawartego w pliku (niekiedy zawiera też opcje z jakimi należy uruchomić interpreter).

Natomiast w przypadku skompilowanego kodu C/C++ punktem startu jest funkcja `main`. Zakończenie tej funkcji oznacza zakończenie programu, a wartość przez nią zwracana odpowiedzialna jest za tzw. kod powrotu przekazany procesowi wywołującemu program.

Wyjątki

Języki wyższego poziomu (np. C++, Python) pozwalają na stosowanie mechanizmu wyjątków do obsługi błędów. Wyjątek powoduje przerwanie wykonywania kodu w momencie gdy jest generowany i „zwinięcie” stosu wykonywanych funkcji aż do tej w której jest obsługiwany, a jeżeli takiej nie ma to zakończenie programu.

```
/// Przykład C++
```

```

#include<iostream>
// generowanie wyjątków
void wyj(int i) {
    switch(i) { // rzuca wyjątek w zależności od i
        case 0:
            throw "Ala ma kota";
        case 1:
            std::cout << "wypisze sie (0) ?" << std::endl;
            throw "Kot ma Ale";
        case 2:
            throw 13;
    }
    std::cout << "wypisze sie (1) ?" << std::endl;
}

int main() {
    for (int j=0; j<4; j++) {
        std::cout << std::endl << "j = " << j << std::endl;
        try {
            wyj(j);
            std::cout << "wypisze sie (2) ?" << std::endl;
            wyj(1);
        } catch (char const* opis) {
            std::cout << "WYJATEK: " << opis << std::endl;
        } catch (...) {
            std::cout << "inny wyjątek" << std::endl;
        }
        // aby zaoszczędzić jednego kopiowania należy przechwytywać wyjątek przez referencję
        // ale należy pamiętać iż ze względu na mechanizm działania wyjątków
        // wartość ta i tak będzie kopiowana
    }
}

```

Przykład Python

Prawie wszystkie błędy w Pythonie mają postać wyjątków,
które mogą zostać obsłużone blokiem try/except.

```

try:
    a = 5 / 0
except ZeroDivisionError:
    print("dzielenie przez zero")
except:
    print("inny błąd")

```

Przy obsłudze błędów może przydać się instrukcja pusta "pass",
która w tym przypadku pozwala na zignorowanie obsługi danego błędu.

```

try:
    słownik["a"] += 1
except:
    pass

```

Powyższy kod zwiększy wartość związaną z kluczem "a" w słowniku "słownik",
jednak gdy napotka błąd (np. słownik nie zawiera klucza "a") zignoruje go.

Możemy także generować wyjątki z naszego kodu, służy do tego instrukcja raise,
której należy przekazać obiektem dziedziczącym po \python{BaseException} np:

```

\begin{CodeFrame*}[python]{}
raise BaseException("jakiś błąd")

```

Rekurencja

Rekurencja jest mechanizmem, polegającym na wywoływaniu funkcji przez samą siebie, który ma na celu implementację niektórych algorytmów. Może być zastosowana chyba w każdym języku pozwalającym na definiowanie funkcji. Ograniczeniem jest maksymalna ilość wywołań funkcji związana z wielkością stosu - w przypadku zbyt wielu wywołań kolejnych funkcji program zakończy się błędem. Każdy algorytm zapisywany rekurencyjnie może zostać zamieniony na algorytm iteracyjny (wykorzystujący pętle).

Przykład Python

```

# obliczanie silni z użyciem rekurencji
def silnia(n):
    # każda rekurencja musi mieć warunek końca

```

```

if n == 1:
    return 1
else:
    return n*silnia(n-1)

```

```
silnia(20)
```

Przykład Python

iteracyjne obliczanie silni

```

def silnia(n):
    s = 1
    for i in range(2,n+1):
        s = s*i
    return s

```

```
silnia(20)
```

Więcej o (bardziej złożonych) zmiennych

Grupowanie zmiennych

Struktury i klasy

Struktura jest złożonym typem danych służącym do grupowania powiązanych ze sobą logicznie zmiennych. Zmienne wchodzące w skład struktury (*pole*) identyfikowane są nazwami i mogą być różnych typów. Struktura zajmuje ciągły obszar pamięci, w którym umieszczane są wartości kolejnych pól.

Klasa oprócz pól może zawierać także funkcje (metody) - zarówno operujące na tych polach jak i z nimi nie powiązane. W niektórych przypadkach także struktury mogą zawierać metody - np. w C++ nie ma silnego rozróżnienia pomiędzy strukturami a klasami (różnią się tylko domyślną widocznością pól i metod).

Zmienne których typem jest jakaś klasa często określa się jako *obiekt* danej klasy. Niekiedy mówi się także o instancji danej klasy (reprezentowanej przez zmienną).

Niektóre z języków pozwalają na definiowanie w ramach struktury lub klasy pól i/lub metod *statycznych*, które nie są powiązane z żadnym istniejącym egzemplarzem danej struktury czy klasy - są wspólne dla wszystkich i można się do nich odwoływać bez istnienia obiektu.

/// Przykład C++

```
#include <iostream>
```

```
#include <stdint.h>
```

```

struct NazwaStruktury {
    // pola składowe
    int a;
    std::string d;

    // zmienna statyczna
    // wspólna dla wszystkich obiektów tej klasy
    static int x;

    // stała
    static const int y = 7;

    // pola binarne (jedno i trzy bitowe)
    uint8_t mA :1;
    uint8_t mB :3;

    // metody składowe
    void wypisz() {
        std::cout << " a=" << a << " d=" << d << "\n";
    }

    // deklaracja metody
    // definicja musi być podana gdzieś indziej
    int getSum(int b) ;

    /// metody statyczna
    static void info() {
        std::cout << "INFO\n";
    }
}

```

```

// konstruktor i destruktor
NazwaStruktury(int aa = 0) {
    std::cout << "konstruktor\n";
    a = aa;
    d = "abc ...";
}
~NazwaStruktury() {
    // potrzebny gdy klasa tworzy jakieś
    // obiekty które należy usuwać, itp
    std::cout << "destruktor\n";
}
};

// definicja zmiennej statycznej z nadaniem jej wartości
// jest to niezbędne aby była ona widoczna ...
int NazwaStruktury::x = 13;

// wcześniej zadeklarowane metody
// możemy definiować także poza deklaracją klasy
int NazwaStruktury::getSum(int b) {
    return a + b;
}

int main() {
    // korzystanie ze struktur
    NazwaStruktury s;
    s.a = 45;
    s.wypisz();

    // korzystanie z metod statycznych
    NazwaStruktury::info();
    // a także poprzez obiekt danej klasy
    s.info();
}

```

Przykład Python

```

class NazwaKlasy:
    # pola składowe
    a=0
    d="ala ma kota"

    # metody składowe
    def wypisz(self):
        print(self.a + self.b)
    # warto zauważyć jawny argument
    # w postaci obiektu tej klasy
    # w C++ także występuje ale nie jest
    # jawnie deklarowany, ani nie trzeba
    # się nim jawnie posługiwać

    # metody statyczna
    @staticmethod
    def info():
        print("INFO")

    # konstruktor (z jednym argumentem)
    def __init__(self, x = 1):
        print("konstruktor", self.a , self.d)
        # i kolejny sposób na utworzenie
        # pola składowego klasy
        self.b = 13 * x

# korzystanie z klasy
k = NazwaKlasy()
k.a = 67
k.wypisz()

# do metod można odwoływać się także tak:
# (jawne użycie argumentu w postaci obiektu klasy)
NazwaKlasy.wypisz(k)

# korzystanie z metod statycznych
NazwaKlasy.info()

print("k jest typu:", type(k))
print("natomiast k.a jest typu:", type(k.a))

```

obiekty można rozszerzać o nowe składowe i funkcje:

```
k.b = k.a + 10  
print(k.b)
```

/// Przykład PHP

```
<?php
```

```
class NazwaKlasy {  
    public $a;  
    public $d = "tekst";  
  
    # metody składowe  
    public function wypisz() {  
        echo "a=" . $this->a . " d=" . $this->d . "\n";  
    }  
    # warto zauważyć jawne odwołania do składowych  
    # poprzez zmienną $this  
  
    # metody statyczna  
    public static function info() {  
        echo "INFO\n";  
    }  
}
```

// korzystanie z klasy

```
$k = new NazwaKlasy;  
$k->a = 87;  
$k->wypisz();
```

// korzystanie z metod statycznych

```
NazwaKlasy::info();  
// a także poprzez zmienną przechowującą nazwę klasy  
$t="NazwaKlasy";  
$t::info();
```

```
?>
```

Metody wchodzące w skład struktury oraz pola statyczne nie wpływają na rozmiar obiektów których jest typem.

Tablice

Tablica jest strukturą danych w której elementy (takiego samego typu) są ułożone w porządku liniowym i są dostępne za pomocą indeksów (kluczy). Typowo tablica indeksowana jest liczbami całkowitymi nie ujemnymi oraz zajmuje ciągły obszar pamięci.

Dostęp do elementów tablicy odbywa się w oparciu o obliczanie ich adresu na podstawie zależności: $\text{AdresElementu} = \text{AdresPoczatkuTablicy} + \text{IndexElementu} * \text{RozmiarElementu}$.

/// Przykład C++

```
#include <iostream>  
#include <vector>
```

```
int main() {  
    // klasyczna tablica  
    int t[4] = {1, 8, 3, 2};  
    std::cout << t[2] << " -> ";  
    t[2] = 55;  
    std::cout << t[2] << " = " << *(t+2) << "\n";  
  
    // jest ona podobnie jak struktura ciągłym obszarem pamięci,  
    // możliwe jest zatem traktowanie takiej tablicy jako struktury  
    struct Struktura {  
        int a, b, c, d;  
    };  
  
    Struktura *tt = (Struktura*)t;  
    std::cout << tt->a << " " << tt->c << "\n";  
  
    // dynamicznie alokowana tablica C++ STL  
    std::vector<int> v(4);  
    v[3] = 21;  
    std::cout << v[3] << "\n";  
}
```

```

### Przykład Bash
a[0]="a b c"
a[1]=123
a[2]=zz
a[3]=qq

x=1
echo "wybrane elementy tablicy: " ${a[$x]} ${a[0]}
echo "cała tablica: " ${a[@]}
echo "ilość elementów w tablicy tablica: " ${#a[@]}

/// Przykład JavaScript
var t = ['a b c', 8, 3, 2];

console.log(
  "pierwszy element: " + t[0] + "\n" +
  "długość tablicy: " + t.length + "\n"
);

// usuwamy 2 elementy od pozycji 1
t.splice(1, 2)

// dodajemy nowe elementy
t[5] = "pp";
t[7] = "ww";

// wypisanie wszystkich elementów
for (var i=0; i<t.length; ++i) {
  console.log("t["+ i +"]=" + t[i] + "\n");
}

// za pomocą for..in
for (var ii in t) {
  console.log("t["+ ii +"]=" + t[ii] + "\n");
}

```

Listy

Lista jest strukturą danych w której elementy są ułożone w porządku liniowym na zasadzie poprzedni-następny. Wyróżnia listy:

jednokierunkowe

w których w oparciu o element listy można uzyskać tylko element następny lub ustalić że jest ostatni - można przemieszczać się po takiej liście tylko w jednym kierunku

dwukierunkowe

w których w oparciu o element listy można uzyskać następny i poprzedni, lub ustalić że jest pierwszym lub ostatnim - umożliwia to przemieszczanie się po elementach listy w obu kierunkach

cykliczne

(zarówno jedno i dwukierunkowe) w których ostatni element wskazuje jako następny pierwszy element, a (w przypadku dwukierunkowych) pierwszy element wskazuje ostatni jako swojego poprzednika

```

/// Przykład C++
#include <iostream>
#include <list>

int main() {
  std::list<int> l;

  // dodanie elementu na końcu
  l.push_back(17);
  l.push_back(13);
  l.push_back(3);
  l.push_back(27);
  l.push_back(21);
  // dodanie elementu na początku
  l.push_front(8);

  // wypisanie liczby elementów
  std::cout << "size=" << l.size() << "\n";

  // wypisanie pierwszego i ostatniego elementu

```

```

std::cout << "first=" << l.front() << " last=" << l.back() << "\n";

// usuniecie ostatniego elementu
l.pop_back();

// posortowanie listy
l.sort();

// odwrócenie kolejności elementów
l.reverse();

// usuniecie pierwszego elementu
l.pop_front();

for (std::list<int>::iterator i = l.begin(); i != l.end(); ++i) {
    // wypisanie wszystkich elementów
    std::cout << *i << "\n";
    // możliwe jest także:
    // - usuwanie elementu wskazanego przez iterator
    // - wstawianie elementu przed wskazanym przez iterator
}
}

```

Przykład Python

```

l = [ 3, 5, 8 ]

# wstawienie elementu na koniec
l.append(1)

# wstawienie elementu na pozycje 2
l.insert(2, 13)

print("liczba elementów =", len(l))
print("pierwszy =", l[0])
print("dwa kolejne =", l[1:3])

# wypisanie wszystkich elementów
for e in l:
    # możemy modyfikować zmienną "e",
    # ale nie będzie miało to wpływu na listę
    print(e)

# alternatywne iterowanie po elementach
# (pozwała na ich modyfikowanie)
for i in range(len(l)):
    l[i] = l[i] + 1
    print(l[i])

# możemy też uzyskać listę w oparciu o wykonanie jakiś
# operacji na danej liście w formie jednolinijkowca:
l = [a * 2 for a in l]
# listę taką możemy przypisać do innej
# lub (jak wyżej) do tej samej zmiennej

# pobranie i usuniecie ostatniego elementu
print("ostatnim był:", l.pop())
print("ostatnim był:", l.pop())

# pobranie i usuniecie elementu na wskazanej pozycji
print("drugim elementem był:", l.pop(1))

# wypisanie całej listy
print(l)

```

Listy umożliwiają łatwą implementację m.in.:

- kolejki typu **FIFO** (First In First Out)

- poprzez dodawanie elementów na koniec listy, a pobieranie z początku listy

- stosu, czyli kolejki typu **LIFO** (Last In First Out)

- poprzez dodawanie elementów na koniec listy i pobieranie także z końca listy

Listy nie muszą zajmować ciągłego obszaru pamięci - każdy z elementów może być alokowany osobno i tylko zawierać wskaźniki na następny i ew. poprzedni element. Utrudnia (wydłuża) to dostęp do n-tego elementu listy, ale usprawnia dodawanie i usuwanie elementów (zarówno z początku, środka jak i końca listy).

Tablice asocjacyjna (mapy)

Tablica asocjacyjna (nazywana także mapą lub słownikiem) podobnie do zwykłej tablicy jest zbiorem par klucz-wartość, z tym że klucze nie muszą tutaj być kolejnymi liczbami całkowitymi zaczynającymi się od zera a mogą być dowolnymi wartościami (napisami, wskaźnikami, etc). Często (np. C++) tablice takie implementowane są jako posortowane drzewa binarne, dzięki czemu dostęp do szukanego elementu realizowany jest w czasie logarytmicznym.

```
/// Przykład C++
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> m;

    m["a"] = 6;
    m["cd"] = 9;
    std::cout << m["a"] << " " << m["ab"] << "\n";

    // wyszukanie elementu po kluczu
    std::map<std::string, int>::iterator iter = m.find("cd");
    // sprawdzenie czy istnieje
    if (iter != m.end()) {
        // wypisanie pary - klucz wartość
        std::cout << iter->first << " => " << iter->second << "\n";
        // usunięcie elementu
        m.erase(iter);
    }

    m["a"] = 45;

    // wypisanie całej mapy
    for (iter = m.begin(); iter != m.end(); ++iter)
        std::cout << iter->first << " => " << iter->second << "\n";
    // jak widać mapa jest wewnętrznie posortowana
}
```

```
### Przykład Python
m = { "ab" : 11, "cd" : "xx" }
x = "e"
m[x] = True;

# pobranie samych kluczy
for k in m:
    print (k, "=>", m[k])

# sprawdzenie istnienia
if "ab" in m:
    print ("jest ab")
    # usunięcie elementu
    del m['ab']

# modyfikacja wartosci
m["cd"] = "oi"

# pobranie par klucz wartosc
for k,v in m.items():
    print (k, "=>", v)
```

```
/// Przykład PHP
<?php

$a = array('ab' => True, 'cd' => 'xx');
$a['efg'] = 15;
$a['cd']="yu";

# sprawdzenie istnienia
if (isset($a['ab'])) {
    // uwaga: gdy $a['ab'] = null
    // to isset zwróci FALSE
    // patrz array_key_exists()
    echo "jest ab";

    # usunięcie elementu
    unset($a['ab']);
}
```

```

while ( list($k, $v) = each($a) ) {
    echo "a[$k] = $v\n";
}

?>

```

```

/// Przykład JavaScript
// pseudo odpowiednik
// tablicy asocjacyjnej

var o = {a: 1, b: 2, c: 3};

// dodanie elementu
o.ab = "pp";

// usunięcie elementu
delete o.b;

for (var k in o) {
    console.log(
        `o.${k} = ${o[k]} \n`
    );
}

```

W standardowej mapie klucz jest unikalny (tzn. nie mogą występować elementy z takim samym kluczem), zapewnia to jednoznaczność odwołań poprzez operator tablicowy []. W C++ jeżeli chcemy mieć kilka jednakowych kluczy z różnymi wartościami można skorzystać `std::multimap<>`.

Strukturami zbliżonymi w działaniu do mapy są *zbiory* (`std::set<>` i `std::multiset<>` z C++ oraz `set([k1, k2, ...])` z Pythona). Różnią się one od map tym że nie przechowują one wartości a jedynie same klucze.

Wskaźniki i referencje

Wskaźnik jest zmienną, która przechowuje adres pamięci, pod którym znajdują się jakieś dane (inna zmienna). Jako że wskaźnik jest zmienną która też jest umieszczona gdzieś w pamięci można utworzyć wskaźnik do wskaźnika itd. Na wskaźnikach można wykonywać operacje arytmetyczne (najczęściej jest to dodawanie offsetu). Na wskaźniku można wykonać operację wyłuskania czyli odwołania się do wartości zmiennej pod adresem na który wskazuje, a nie do zmiennej wskaźnikowej (zawierającej adres).

Wskaźniki pozwalają na operowanie dużymi zbiorami danych (duże struktury, napisy, etc) bez konieczności ich kopiowania przy przekazywaniu do funkcji, umieszczaniu w różnych strukturach danych, sortowaniu, itd (kopiowaniu ulega jedynie wskaźnik czyli adres) oraz na współdzielenie tych samych danych pomiędzy różnymi obiektami.

Wskaźnik może wskazywać na niewłaściwy adres w pamięci (np. na skutek zwolnienia tego fragmentu lub błędu w operacjach matematycznych na wskaźnikach - wyjściu poza dozwolony zakres), typowo wskaźnikowi który nic nie wskazuje przypisuje się wartość NULL (zero). Wyłuskania wskaźników o wartości NULL lub wskazujących niewłaściwy obszar pamięci prowadzą do błędów programu, często do zakończenia programu z powodu naruszenia ochrony pamięci ("Segmentation fault").

Referencja jest zbliżona do wskaźników (w zasadzie jest to wskaźnik trochę inaczej traktowany przez kompilator) - także pozwala na unikanie kopiowania dużych danych. W odróżnieniu od wskaźnika odwołania do referencji zawsze skutkują wyłusaniem, nie jest możliwa arytmetyka wskaźnikowa na referencjach, referencja musi też wskazywać na poprawny obszar pamięci (minimalizacja ryzyka błędu odwołania do niewłaściwego adresu). Możliwa jest referencja na wskaźnik a także wskaźnik do referencji.

```

/// Przykład C++
#include <stdio.h>
#include <iostream>

void f1(int *b) {
    *b = 2 * *b;
}

void f2(int (*f)(const char *s)) {
    f("Uwolnić mrożone truskawki !!!");
}

struct kl {

```

```

// pola składowe
int a;

int getSum(int b) {
    return a + b;
}
};

int main() {
    // zmienna typu int i wskaźnik na zmienną typu int
    int a = 5678;
    int *b = NULL;

    // pobranie adresu zmiennej do wskaźnika
    b = &a;
    std::cout << "a ma adres " << b << "\n";

    // modyfikacja wartości na którą wskazuje wskaźnik
    *b = 3456;
    std::cout << a << " = " << *b << "\n";

    // referencja
    int &c = a;
    c = 6543;
    std::cout << a << " = " << c << "\n";

    // wskaźniki na obiekty
    std::pair<int,int> p = std::make_pair(2, 5);
    std::pair<int,int> *q = &p;

    // dostęp do składowych poprzez wskaźnik na strukturę
    (*q).first = 7;
    q->second = 8;
    std::cout << p.first << " " << p.second << "\n";

    // wskaźnik na składową
    b = &(q->second);
    *b = 3;
    std::cout << p.first << " " << p.second << "\n";

    // przekazywanie wskaźnika do funkcji:
    // 1. (podobnie jak trzymanie wskaźników na obiekty,
    //    zamiast obiektów w listach itp) pozwala na
    //    przekazywanie większych obiektów bez ich kopiowania
    // 2. pozwala na modyfikowanie wartości argumentów:
    f1(b);
    std::cout << p.first << " " << p.second << "\n";

    // wskaźnik "fun" na funkcję przyjmującą
    // wskaźnik const char i zwracającą int
    int (*fun)(const char *s);
    // przypisanie adresu funkcji puts do zmiennej fun
    fun = &puts;
    // użycie wskaźnika na funkcję jako funkcji
    fun("aaa");

    // wskaźnik na funkcję może być przekazywany
    // do innych funkcji jako argument
    f2(fun);

    // wskaźniki a tablice
    // w C tablica to wskaźnik na pierwszy element
    // a t[x] jest równoważne *(t+x)
    short t[4] = { 11, 22, 33, 44 };
    short *tt = t;
    std::cout << "t[2] = " << t[2] << " = " << *(t + 2) << "\n";

    std::cout << "t[0] = " << *tt << " @ " << tt << "\n"; ++tt;
    std::cout << "t[1] = " << *tt << " @ " << tt << "\n";

    // wskaźnik na metodę składową jakiejś klasy
    // (z wyjątkiem metod statycznych)
    // wymaga określenia typu tej klasy,
    // gdyż jest on typem niejawnego argumentu jej metod
    int (kl::*fun2)(int) = &kl::getSum;

    // aby skorzystać trzeba mieć obiekt danej klasy

```

```

// (lub wskaźnik do niego)
kl o1;
o1.a = 2;
kl *o2 = &o1;
std::cout << (o1.*fun2)(3) << " " << (o2->*fun2)(3) << "\n";
}

```

Przykład Python

```
a, b = 5, [1, 2, 3]
```

```

def pinfo(x, xx):
    print(
        "id(x) =", hex(id(x)),
        " == " if id(xb) == id(xx) else " != ",
        "id(xx) =", hex(id(xx))
    )

# utworzenie kopii zmiennej oznacza utworzenie
# nowej referencji wskazującej nadal na ten sam
# obiekt w pamięci
aa = a
pinfo(id(a), id(aa))
bb = b
pinfo(id(b), id(bb))

# przypisanie nowego obiektu pod zmienną
# powoduje zmianę (adresu) obiektu który
# wskazuje:
aa = 5
pinfo(id(a), id(aa))
bb = [9, 11, 13]
pinfo(id(b), id(bb))

# ale modyfikacja obiektów "immutable"
# (takich jak liczby, napisy) nie jest możliwa
# zatem zawsze tworzony jest nowy obiekt
aa=a
aa+=1
pinfo(id(a), id(aa))

# jeżeli do starego obiektu nie ma innych
# referencji może on zostać usunięty
# a nowy może być umieszczony w jego miejscu
# (w takim przypadku wynik id się nie zmieni)

# natomiast modyfikacja obiektów "mutable"
# (takich jak listy i słowniki)
# nie tworzy nowego obiektu tylko modyfikuje
# istniejący (wynik id się nie zmieni) zatem
# wszystkie referencje wskazują na zmodyfikowany
# obiekt:
bb = b
bb[0] = 17
print ("bb =", bb, "b=", b)
pinfo(id(b), id(bb))

# aby uzyskać kopię należy skorzystać
# z odpowiedniej metody
bb = b.copy()

print ("bb =", bb, "b=", b)

bb[0] = 99
print ("bb =", bb, "b=", b)
pinfo(id(b), id(bb))

# uwaga kopiowanie takie jest płytkie: jeżeli w
# liście mamy obiekty "mutable" obie (niezależne
# pod względem zbioru elementów) kopie listy
# będą wskazywać na te same obiekty

# usuwanie zmiennej
b = None
# mapowanie zmiennej "b" zostało zmienione
# nie wskazuje już na listę tylko na
# obiekt typu NoneType

```

```

bbb = bb
del bb
# nazwa "bb" została usunięta
# ale do danych możemy dostawać się przez "bbb"

# dopiero po usunięciu wszystkich referencji
# na dany obiekt to Python może go usunąć (ale
# nie musi wykonać tego natychmiast)

# python także umożliwia przekazywanie funkcji
# jako argumentu do innej funkcji
def a(z):
    print(z*3)

def b(x, y):
    x(y+2)

b(a, 1)

# ponadto funkcja może także zwracać funkcję:
def aa(y):
    def t(x):
        return y*x
    return t

b = aa(3)
b(2)

# można też:
aa(3)(4)

```

Przykład Bash

```

# w Bashu nie ma operacji na wskaźnikach
# ale podobną funkcję w pewnych wypadkach
# może pełnić zmienna zawierająca nazwę
# innej zmiennej

A='tekst do wypisania, $HOME, `ls`';
B="A";

# proste podejście typu echo ${B}
# (działające np. w PHP) nie zadziała,
# ale można to zrobić na kilka innych sposobów:

C=${!B}; # ta metoda nie działa w czystym sh
echo $C

export A
C=$(echo "\${B}" | envsubst)
# ta metoda wymaga zewnętrznego polecenia
# envsubst i wyeksportowania zmiennych
echo $C

C=$( eval "echo \${B}" )
echo $C

# z użyciem większej liczby poleceń eval możemy
# zapewnić także podstawienie kolejnego poziomu
# zmiennych lub wykonanie wpisanych w nich poleceń
C=$( eval eval "echo \${B}" )
echo $C

# zmienna może przechowywać komendę / nazwę
# funkcji do wykonania
x=ls

$x /tmp

```

/// Przykład PHP

```

<?php

// w PHP korzystanie z nazwy zmiennej
// przechowywanej w innej zmiennej
// jest jeszcze prostsze:

```

```

$a="12";
$b="a";
$c="b";

echo "$a $b $c\n"
echo "${b} ${c}\n"
echo "${${c}}\n"

// możliwe jest także używanie zmiennych
// zawierających nazwy funkcji oraz
// przekazywanie ich do innych funkcji
function fa($z) {
    print($z*3);
    print("\n");
}

$c="fa";
$c(2);

function fb($x, $y) {
    $x($y+2);
}

fb("fa", 1);

?>

```

Zasięg zmiennej

Często (np. C / C++) zasięg zmiennych (widoczność i istnienie) jest limitowany do bloku w którym zostały zadeklarowane, zmienne z bloków wewnętrznych mogą przesłaniać zmienne zadeklarowane wcześniej. W niektórych językach (np. Pythonie, Bash'u) mechanizm ten nie jest stosowany (utworzenie zmiennej wewnątrz if'a lub pętli powoduje jej widoczność poza tym blokiem).

Wywołanie funkcji powoduje rozpoczęcie nowego kontekstu w którym zmienne z bloku wywołującego funkcję nie są widoczne (ale nadal istnieją). Typowo argumenty do funkcji przekazywane są przez kopiowanie, więc funkcja nie ma możliwości modyfikacji zmiennych z bloku ją wywołującego nawet do niej przekazanych (wyjątkiem jest przekazanie przez referencję lub wskaźnik).

W przypadku manualnej alokacji pamięci lub tworzenia obiektów poprzez `new` limitowana jest widoczność i istnienie otrzymanego wskaźnika, ale nie zaalokowanego bloku pamięci. Zatem ograniczona jest widoczność takich zmiennych ale nie czas ich istnienia, dlatego też przed utratą wskaźnika na nie należy je usunąć (zwolnić zaalokowaną pamięć).

```

/// Przykład C++
#include <iostream>

int funA(int a) {
    a = a*2;
    return a;
}

int funB(int &a) {
    a = a*2;
    return a;
}

int main() {
    int a = 57, b = 23;
    std::cout << ++a << " " << ++b << "\n";
    // wypisze 58 24
    {
        int a = b;
        std::cout << ++a << " " << ++b << "\n";
        // wypisze 25 25
        // bo a z tego bloku (==24) przesłoniło wcześniejszą zmienną a (==58)
    }
    std::cout << ++a << " " << ++b << "\n";
    // wypisze 59 26
    // bo a z wcześniejszego bloku już nie istnieje i nie przesłania naszego a (==58)

    std::cout << funA(a) << " " << funB(b) << "\n";
    // wypisze wartości zwracane przez funkcje czyli 118 (59*2) i 52 (26*2)
    std::cout << a << " " << b << "\n";
}

```

```

// wypisze aktualne wartości argumentu:
// a=59 bo przekazanie przez wartość i funkcja operowała na własnej kopii
// b=52 bo przekazane przez referencję i funkcja operowała na tej samej kopii
}

```

Przykład Python

```

a, b, d = 5, 12, [1, 2, 3]

def f1(c):
    return c+b

def f2(c):
    b=2
    return c+b

# f1 korzysta z zmiennej globalnej b
# f2 przysłała sobie zmienną globalną b poprzez swoją zmienną lokalną
print("f1(a) =", f1(a), "f2(a) =", f2(a), "ale b nadal wynosi:", b)

def f3(c):
    global b
    b=16
    return c+b

# f3 dzięki temu że jawnie deklaruje iż używa globalnego b
# może modyfikować zmienną globalną
print("f3(a) =", f3(a), "teraz b wynosi:", b)

def f4(c):
    c = 2*c
    return c

# argumenty funkcji są zmiennymi lokalnymi
print("f4(a) =", f4(a), "ale a nadal wynosi:", a)

def f5(c):
    c[0]="xx"
    return c

print("f5(d) =", f5(d), "tym razem d uległo modyfikacji:", d)

if d[1] == 2:
    x = "ABC"
else:
    x = 22

print("zmienna x jest widoczna poza blokiem w którym została utworzona", x)

```

Kolejność bajtów

```

#include <inttypes.h>
#include <stdio.h>
int main() {
    // dane jako tablica liczb 16 bitowych
    uint16_t aa[4] = {0x1234, 0x5678, 0x9abc, 0xdef};

    // wypisujemy ją
    printf("A0: %x %x %x %x\n", aa[0], aa[1], aa[2], aa[3]);
    // chyba nikogo nie zaskoczy wynik powyższego printf: A0: 1234 5678 9abc def

    // wypisujemy dwie pierwsze liczby rozłożone na części 8 bitowe (poszczególne bajty)
    printf("A1: %x %x %x %x\n", (aa[0] >> 8) & 0xff, aa[0] & 0xff, (aa[1] >> 8) & 0xff, aa[1] & 0xff);
    // efekt też jest oczywisty: A1: 12 34 56 78

    // każemy na te same dane patrzeć jako na liczby 8 bitowe (poszczególne bajty)
    uint8_t* bb = (uint8_t*) aa;

    printf("B0: %x %x %x %x\n", bb[0], bb[1], bb[2], bb[3]);
    // czego się teraz spodziewamy?
    // - wypisze nam tylko połowę oryginalnej tablicy
    // - ale dokładny wynik zależy od architektury na której uruchamiamy program:
    //   * na little endian (np. x86) będzie to: B0: 34 12 78 56
    //   * na big endian (np. sparc) będzie to (bardziej naturalne dla człowieka): B0: 12 34 56 78
}

```

Fakt, że różne komputery ten sam ciąg zero-jedynkowy mogą interpretować jako różne liczby (w zależności od architektury *big endian* vs *little endian*), powoduje że przy wymianie danych między systemami konieczne jest ustalenie sposobu tej interpretacji (np. protokoły sieciowe takie jak IP używają *big endian*) lub zawarcie tej informacji w wymienianych danych (kodowania Unicode UTF-16 i UTF-32 zawierają na początku danych znacznik BOM).

Napisy

Napisy są to ciągi liczb, w których kolejne liczby (pojedynczo lub grupami) interpretowane są jako kolejne znaki (litery). W przypadku kodowań o ustalonej długości znaku (np. UTF-32, ASCII) każda n-bitowa liczba odpowiada jednemu znakowi napisu (przy czym i tak mogą się one graficznie nakładać, ale każdy jest integralną całością). W przypadku kodowań o zmiennej długości znaku długość znaku kodowana jest w ramach tego znaku (np. w UTF-8 długość wynosi od 1 do 4 bajtów, informacja o długości zakodowana jest w pierwszym bajcie, a dodatkowe bajty mają ustawione najstarsze bity na 10 co jednoznacznie identyfikuje je jako uzupełniające).

Mogą one być przechowywane jako tablica lub lista tablic (ułatwia to operowanie dużymi zmiennymi napisowymi - unikanie relokacji i przepisywania dużej ilości danych). Możliwe jest kilka sposobów przechowywania napisu w tablicy różniących się metodą uzyskiwania informacji o końcu napisu - może to być osobna zmienna przechowująca długość napisu (lub adres jego końca) albo znacznik końca przechowywany w samym napisie (w tej roli stosowane jest 0 w tzw. NULL-end stringach z języka C).

W przetwarzaniu napisów bardzo często stosowane są wyrażenia regularne służące do dopasowywania napisów do wzorca który opisują, wyszukiwaniu/zastępowaniu tego wzorca. Do typowej, podstawowej składni wyrażeń regularnych zalicza się m.in. następujące operatory:

- .
- [a-z] - znak z zakresu
- [^a-z] - znak z poza zakresu (aby mieć zakres z ^ należy dać go nie na początku)
- ^ - początek napisu/linii
- \$ - koniec napisu/linii
-
- *
- ? - 0 lub jedno powtórzenie
- + - jedno lub więcej powtórzeń
- {n,m} - od n do m powtórzeń
-
- () - pod-wyrażenie (może być używane dla operatorów powtórzeń, a także dla referencji wstecznych)
- | - alternatywa: wystąpienie wyrażenia podanego po lewej stronie albo wyrażenia podanego prawej stronie

```
/// Przykład C++
#include <stdio.h>
#include <iostream>

#include <string>
#include <string.h>
#include <bitset>
#include <regex>
#include <sstream>

int main() {
    // napisy w stylu C
    // czyli tak naprawdę tablice bajtów (znaków)
    const char* x = "abcdefg";

    // wypisanie długości napisu
    printf("%d\n", strlen(x));

    // wypisanie pod-napisu od 2 do końca
    puts(x+2);

    // wyszukiwanie
    // pod-napisu "cd" w x od pozycji 1
    const char* cd = strstr(x+1, "cd");
    printf("%d\n", cd-x);

    // 3 znakovy pod-napis napisu x
    // rozpoczynający się od cd
    char buf[16];
    strncpy(buf, cd, 3);
    buf[3]=0; // NULL end
    puts(buf);
}
```

```

// porównywanie
if (strcmp(x, "a") == 0)
    puts("x == \"a\"");
if (strncmp(x, "a", 1) == 0)
    puts("pierwsze 1 znaków x to \"a\"");

// napisy w stylu C++
std::string xx(x);
std::string y = "aa bb cc bb dd bb ee";

// wypisanie długości napisu
std::cout << xx.size() << "\n";
// .size() to to samo co .length()

// uzyskanie napisu w stylu C
puts(xx.c_str());

// wypisanie pod-napisu od 2 do końca
std::cout << xx.substr(2) << "\n";
std::cout << xx.substr(2, std::string::npos) << "\n";
// i od 0 (początku) do 3
std::cout << xx.substr(0, 3) << "\n";

// wyszukiwanie pod-napisu "bb" w y od pozycji 5
std::cout << y.find("bb", 5) << "\n";

// porównywanie
if (xx == "a")
    std::cout << "x == \"a\"";
if (xx.compare(0, 1, "a") == 0)
    puts("pierwsze 1 znaków x to \"a\"");

if ( std::regex_match(xx, std::regex(".*[dz].*")) )
    puts("x zawiera d lub z");
// regex_match dopasowuje całość napisu do wyrażenia regularnego
// dopasowanie częściowe wraz z opcjonalnym uzyskaniem
// pasującej części umożliwia: std::regex_search()

// modyfikowanie std::string
xx = "Ala ma psa";
// wstawianie - insert(pozycja, co)
xx.insert(6, " kota i");
std::cout << xx << std::endl;

// zastępowanie - replace(pozycja, ile, czym);
xx.replace(4, 2, "miała samochód", 0, 6);
// mogłoby też być xx.replace(4, 2, "miała"); i parę innych wariantów ...
std::cout << xx << std::endl;

// usuwanie - erase(pozycja, ile);
xx.erase(9, 1); // 9 zamiast 8 bo UTF-8 i ł ma dwa znaki
std::cout << xx << std::endl;

// zastępowanie z użyciem wyrażeń regularnych
std::cout << std::regex_replace(y, std::regex("[bc]+"), "XX") << "\n";

// zastępowanie z użyciem podstawienia
// $2 zostanie zastąpione wartością drugie pod-wyrażenia,
// czyli fragmentu ujętego w nawiasach
std::cout << std::regex_replace (
    y, std::regex("([bc]+) ([bc]+)"), "X-$2-X"
) << "\n";

// konwersja liczb na napis w systemach:
// dwójkowym, ósemkowym, dziesiętnym i szesnastkowym

std::cout << std::bitset<8>(7) << " ";
std::cout << std::oct << 0xf << " ";
std::cout << std::dec << 010 << " ";
std::cout << std::hex << 0b11 << "\n";

// liczby podawane do wypisywania są w odpowiednio systemach:
// dziesiętnym, szesnastkowym, ósemkowym i dwójkowym
// wskazane jest to przez brak prefixu i prefixy "0x" "0" "0b"

// alternatywnie w stylu printf, ale bez dwójkowego

```

```

printf("0o%o %d 0x%x\n", 0xf, 010, 0b11);

// wypisywanie znaków unicode
puts("\u21c4 = ⇐");

// strumienie napisowe
std::ostream zzz;
zzz << xx << " cpp\n";
zzz << 34.6 << " " << std::oct << 0xf << " ";

// konwersja do std::string
xx = zzz.str();

std::cout << xx << "\n";
}

```

Przykład Python

```

import re

x = "abcdefg"
y = "aa bb cc bb dd bb ee"
z = "qw=rt"

# wypisanie długości napisu
print(len(x))

# wypisanie pod-napisu od 2 do końca
# i od 0 (początku) do 3
print (x[2:], x[0:3])

# wypisanie ostatniego i 3 ostatnich znaków
print (x[-1], x[-3:])

# wypisanie co 3ciego znaku z napisu oraz napisu od tyłu
print (y[::3], x[::-1])

# wyszukiwanie
# pod-napisu "bb" w y od pozycji 5
print (y.find("bb", 5))

# porównywanie
if x == "a":
    print("x == \"a\"")

if re.search("[dz]", x):
    print(x, "zawiera d lub z")

# sprawdzanie czy jest pod-napisem
if "ab" in x:
    print ("ab jest pod-napisem:", x)

# sprawdzanie czy jest pod-napisem
if "ba" in x:
    print ("ba jest pod-napisem:", x)

# zastępowanie
print (re.sub('[bc]+', "XX", y, 2))
print (re.sub('[bc]+', "XX", y))

# zastępowanie z użyciem podstawienia
# \|2 zostanie zastąpione wartością drugie pod-wyrażenia,
# czyli fragmentu ujętego w nawiasach
print (re.sub('([bc]+) ([bc]+)', "X-\|2-X", y))

# mamy też wpływ na zachłanność wyrażeń regularnych:

print (re.sub('bb (.*) bb', "X \|1 X", y))
# "bb (.*) bb" dopasowało najdłuższy możliwy fragment, czyli: cc bb dd

print (re.sub('.*bb (.*) bb.*', "\|1", y))
# "bb (.*) bb" dopasowało jedynie "dd", bo najdłuższy możliwy
# fragment został dopasowany przez poprzedzające ".*"

print (re.sub('.*?bb (.*) bb.*', "\|1", y))
# "bb (.*) bb" mogło i dopasowało najdłuższy możliwy fragment,
# gdyż było poprzedzone niezachłanną odmianą dopasowania
# dowolnego napisu, czyli: .*?

```

```

# Po każdym z operatorów powtórzeń (. ? + {n,m}) możemy dodać
# pytajnik (.? ?? +? {n,m}?) aby wskazać że ma on dopasowywać
# najmniejszy możliwy fragment, czyli ma działać nie zachłannie.

# nie da się modyfikować napisu z użyciem odwołań x[numer] np.
# x[2]="X"
# nie zadziała

# można (gdy dużo tego typu modyfikacji) przepisać do listy:
l=list(x)
# alternatywnie można manualnie:
# l=[]
# for c in x:
#     l.append(c)
# albo tak:
# l=[c for c in x]

l[1]="X"
l[3]="qqq"
del l[5]
print("".join(l))

# albo tak (gdy mniej modyfikacji)
print(x[:2] + "XXX" + x[3:])

# można także modyfikować po kolei i dodawać do nowego napisu
s = ""
for c in x:
    if c == "a":
        s += "AA"
    else:
        s += c

print(s)

# przy pomocy metody split() napis możemy podzielić
# na listę napisów przy pomocy dowolnego separatora
print(y.split(" "))
print(y.split(" cc "))

# konwersja liczb na napis w systemach:
# dwójkowym, ósemkowym, dziesiętnym i szesnastkowym
print( bin(7), oct(0xf), str(0o10), hex(0b11) )

# liczby podawane do wypisywania są w odpowiednio systemach:
# dziesiętnym, szesnastkowym, ósemkowym i dwójkowym
# wskazane jest to przez brak prefixu i prefixy "0x" "0o" "0b"

# alternatywnie w stylu printf, ale bez dwójkowego
s = "0o%o %d 0x%x" % (0xf, 0o10, 0b11)
print(s)

# wypisywanie znaków z użyciem ich numeru w unikodzie
# - funkcja chr() zwraca napis złożony ze znaku o podanym numerze
# w ramach napisów można też użyć \uNNNN gdzie NNNN jest numerem znaku
# lub po prostu umieścić dany znak w pliku kodowanym UTF8
print(chr(0x21c4) + " == \u21c4 == 🏐")

# funkcja ord() umożliwia konwersję napis złożonego
# z pojedynczego znaku na numer unicolorowy
print(hex(ord("🏐")), hex(ord("\u21c4")), hex(ord(chr(0x21c4))) )

# Python używa Unicode dla obsługi napisów, jednak przed
# przekazaniem napisu do świata zewnętrznego konieczne
# może być zastosowanie konwersji do określonej postaci
# bytowej (zastosowanie odpowiedniego kodowania)
# służy do tego metoda encode() np.
a = "aąbcć ... 🏐"
inUTF7 = a.encode('utf7')
inUTF8 = a.encode() # lub a.encode('utf8')
print("'" + a + "' w UTF7 to: " + str(inUTF7))
print(" i jest typu: " + str(type(inUTF7)))

# obiekty typu 'bytes' mogą zostać zdekodowane do napisu
print("zdekodowany UTF7: " + inUTF7.decode('utf7'))

```

```
# lub zostać poddane dalszej konwersji np. kodowaniu base64:
import codecs
b64 = codecs.encode(inUTF8, 'base64')
print("napis w UTF8 po zakodowaniu base64 to: " + str(b64))
```

/// Przykład PHP

```
<?php

$x = "abcdefg";
$y = "aa bb cc bb dd bb ee";
$z = "qw=rt";

# wypisanie długości napisu
echo strlen($x) . "\n";

# wypisanie pod-napisu od 2 do końca
echo substr($x, 2);
# i od 0 (początku) 3 kolejne znaki
echo substr($x, 0, 3) . "\n";

# wyszukiwanie
# pod-napisu "bb" w $y od pozycji 5
echo strpos($y, "bb", 5) . "\n";

# porównywanie
if ($x == "a")
    echo "$x == a\n";

if (substr_compare($x, "de", 3, 2)) {
    echo "2 znakowy pod-napis od";
    echo "pozycji 3 w $x to \"de\"\n";
}

if (preg_match("/[dz]/", $x)) {
    echo "$x zawiera d lub z\n";
}

# zastępowanie
echo str_replace("bb", "BB", $y) . "\n";
echo preg_replace('/[bc]+/', "XX", $y, 2) . "\n";
echo preg_replace('/[bc]+/', "XX", $y) . "\n";

# zastępowanie z użyciem podstawienia
# $2 zostanie zastąpione wartością pierwszego pod-wyrażenia,
# czyli fragmentu ujętego w nawiasach
echo preg_replace('/^(.*)=.*$/', '$1', $z);
echo " = ";
echo preg_replace('/^(.*)=.*(.)$/', '$1', $z);
echo "\n";

# wypisywanie w różnych systemach liczbowych
printf("0b%b 0o%o %d 0x%x\n", 7, 0xf, 010, 0b11);

?>
```

Przykład Bash

```
# ${zmienna:-"napis"} zwróci napis gdy
# zmienna nie jest zdefiniowana lub jest pusta
# ${zmienna:= "napis"} zwróci napis
# oraz wykona podstawienie zmienna="napis" gdy
# zmienna nie jest zdefiniowana lub jest pusta
# ${zmienna:+ "napis"} zwróci napis gdy
# zmienna jest zdefiniowana i nie pusta
a=""; b=""; c=""
echo ${a:-"aa"} ${b:= "bb"} ${c:+ "cc"}
echo $a $b $c
a="x"; b="y"; c="z"
echo ${a:-"aa"} ${b:= "bb"} ${c:+ "cc"}
echo $a $b $c

# ${#str} zwróci długość napisu w zmiennej str
# ${str:n} zwróci pod-napis $str od n do końca
# ${str:n:m} zwróci pod-napis $str od n o długości m
x=abcdefg
```

```

echo ${#x} ${x:2} ${x:0:3} ${x:0:${(#x)-2}}

# ${str#"ab"} zwróci $str z obcięty "ab" z początku
# ${str%"fg"} zwróci $str z obcięty "fg" z końca
echo ${x#"abc"} ${x%"efg"}
echo ${x#"ac"} ${x%"eg"}
# w napisach do obcięcia możliwe jest stosowanie shellowych
# znaków uogólniających, czyli *, ?, [abc], itd
# operator # i % dopasowują minimalny napis do usunięcia
# operatory ## i %% dopasowują maksymalny napis do usunięcia
x=abcd.e.fg
echo ${x#*.} ${x##*.} ${x%.*} ${x%%.*}

# ${str/"n1"/"n2"} zwróci $str z zastąpionym
# pierwszym wystąpieniem n1 przez n2
# ${str//"n1"/"n2"} zwróci $str z zastąpionymi
# wszystkimi wystąpieniami n1 przez n2
y="aa bb cc bb dd bb ee"
echo ${y/"bb"/"XX"}
echo ${y//"bb"/"XX"}

# polecenie expr match $x 'wr1\(wr2\)wr3'
# zwróci część $x pasującą do wyrażenia regularnego wr2
# wyrażenia regularne wr1 i wr2 pozwalają na
# określanie części napisu do odrzucenia
# alternatywną składnią jest expr $x : 'wr1\(wr2\)wr3'
z="ab=cd"
expr match $z '^([=]*)\]='
expr $z : '^([=]*)=\.*)$'

# możliwe jest też sprawdzanie dopasowań wyrażeń
# regularnych poprzez (uwaga na brak cytowania):
[[ "$z" =~ ^([=]*)= ]] && echo "OK"

# wypisywanie w różnych systemach liczbowych
printf "0o%o %d 0x%x\n" 0xf 010 3

# do bardziej zaawansowanych operacji
# mogą być przydatne także polecenia:
# grep diff sed awk
# join comm paste

# należy też pamiętać o różnicy w działaniu echo "$a" i
# echo $a w przypadku gdy zmienna a zawiera znaki nowej linii:
# - w pierwszym wypadku będą traktowane jako znaki nowej linii
# - w drugim jako spacje

### Przykład Bash

# obsługa napisów w bash'u przy pomocy standardowych komend POSIXa

# jako że większość operacji bashowych wiąże się z
# uruchamianiem zewnętrznych programów to także
# przetwarzanie napisów może być realizowane w ten sposób

a="aąbcć 123"

# obliczanie długości napisu w znakach, w bajtach i ilości słów w napisie
echo -n $a | wc -m
echo -n $a | wc -c
echo -n $a | wc -w

# obliczanie ilości linii (dokładniej ilości znaków nowej linii)
wc -l < /etc/passwd

# wypisanie 5 pola (rozdzielanego :) z pliku /etc/passwd z eliminacją
# pustych linii oraz linii złożonych tylko ze spacji i przecinków
cut -f5 -d: /etc/passwd | grep -v '^[,]*$'
# komenda cut wybiera wskazane pola, opcja -d określa separator

# alternatywne podejście z użyciem AWK
awk -F: '$5 !~ "^[,]*$" {print $5}' /etc/passwd

# awk daje duże możliwości przy przetwarzaniu tego typu tekstowych baz
# danych ... możemy np. wypisywać wypisywać pierwsze pole w oparciu
# o warunki nałożone na inne:
awk -F: '$5 !~ "^[,]*$" && $3 >= 1000 {print $1}' /etc/passwd

```

```

# jak widać w powyższych przykładach do poszczególnych pól odwołujemy
# się poprzez $n, gdzie n jest numerem pola, $0 oznacza cały rekord

# program dla każdego rekordu przetwarza kolejne instrukcje postaci
# "warunek { komenda }", instrukcji takich może być wiele w programie
# (przetwarzane są kolejno) komenda "next" kończy przetwarzanie danego rekordu

# separator pola ustawiamy opcją -F (lub zmienną FS) domyślnym separatorem
# pola jest dowolny ciąg spacji i tabulatorów (w odróżnieniu od cut
# separator może być wieloznakowym napisem lub wyrażeniem regularnym)
# domyślnym separatorem rekordu jest znak nowej linii
# (można go zmienić zmienną RS)

# awk jest prostym językiem programowania obsługującym podstawowe pętle
# i instrukcje warunkowe oraz funkcje wyszukiujące i modyfikujące napisy
echo "aba aab bab baa bba bba" | awk '{
    for (i=1; i<=NF; ++i) {
        if(i%2==0) # dla każdego pola w rekordzie
                    # jeżeli jego numer jest parzysty
        gsub("b+", "B", $i); # zastąp wszystkie ciągi b pojedynczym B
        ii = index($i, "B") # wyszukaj pozycję pod-napisu B
        if (ii) # jeżeli znalazł to wypisz pozycję i pod-napis
            printf("# %d %s\n", ii, substr($i, ii)) # od tej pozycji do końca
    }
    print $0
}'

# AWK obsługuje także tablice asocjacyjne pozwala to np. policzyć
# powtórzenia słów
echo "aa bb aa ee dd aa dd" | awk '
BEGIN {RS="[ \\t\\n]+"; FS=""}
{słowa[$0]++}
{printf("rekord: %d done\n", NR)}
END {for (s in słowa) printf("%s: %s\n", s, słowa[s])}
'

# podobny efekt możemy uzyskać stosując "uniq -c" (który wypisuje unikalne
# wiersze wraz z ich ilością) na odpowiednio przygotowanym napisie
# (spacje zastąpione nową linią, a linie posortowane)
echo "aa bb aa ee dd aa dd" | tr ' ' '\n' | sort | uniq -c
# jednak rozwiązanie awk można łatwo zmodyfikować aby wypisywało pierwsze
# wystąpienie linii bez sortowania pliku

# inną bardzo przydatną komendą jest sed pozwala ona m.in na zastępowanie
# wyszukiwanego na podstawie wyrażenia regularnego tekstu innym
echo "aa bb cc bb dd bb ee" | sed -e 's#[([bc]\\+\\) \\([bc]\\+\\)#X-\\2-X#g'

# sedowe polecenie s przyjmuje 3 argumenty (oddzielane mogą być dowolnym
# znakiem który wystąpi za s), pierwszy to wyszukiwane wyrażenie, drugi
# tekst którym ma zostać zastąpione, a trzeci gdy jest g to powoduje
# zastępowanie wszystkich wystąpień a nie tylko pierwszego

# należy zwrócić uwagę na różnicę w składni wyrażenia regularnego polegającą
# na poprzedzaniu (, ) i + odwrotnym ukośnikiem aby miały znaczenie specjalne

# sed z opcją -i i wskazaniem pliku modyfikuje zawartości tego pliku
# pozwala to na łatwe stworzenie funkcji rekurencyjnego zastępowania:
rreplace() {
    if [ $# -ne 2 ]; then
        echo USAGE: $1 str1 str2
        return
    fi
    grep -R "$1" . | cut -f 1 -d: | uniq | while read f; do
        [ -L $f ] || sed -e "$s#$1#$2#g" -i $f;
    done;
}

```

/// Przykład JavaScript

```

var x = "abcdefg";
var y = "aa bb cc bb dd bb ee";
var z = "qw=rt";

// podstawianie wartości zmiennych w napisie
console.log(`x=${x} y=${y} z=${z}\n`)

function echo(a) {
    // łączenie napisów (lub napisu i czegoś
    // co można skonwertować na napis)
}

```

```

    // z użyciem operatora +
    console.log(a + "\n");
}

// wypisanie długości napisu
echo(x.length);

// wypisanie pod-napisu od 2 do końca
echo(x.substr(2));
// i od 0 (początku) 3 kolejne znaki
echo(x.substr(0, 3));

// wyszukiwanie
// pod-napisu "bb" w $y od pozycji 5
echo(y.indexOf("bb", 5));

// porównywanie
if (x == "abcdefg")
    echo("x == abcdefg");

if (x.search(/[dz]/) > 0) {
    echo("x zawiera d lub z");
}

// zastępowanie
echo(y.replace("bb", "BB"));
echo(y.replace(/[bc]+/g, "XX"));

// zastępowanie z użyciem podstawienia
// $2 zostanie zastąpione wartością
// pierwszego pod-wyrażenia,
// czyli fragmentu ujętego w nawiasach
console.log(
    z.replace(/^(.*)=.*$/, '$1') +
    " = " +
    z.replace(/^(.*)=.*(.*?)$/, '$1') +
    "\n"
);

var a = 7, b = 0xf, c = 010, d = 0b11;

echo("0b" + a.toString(2))
echo("0o" + b.toString(8))
echo(c.toString())
echo("0x" + d.toString(16))

```

XML

Extensible Markup Language (XML) jest tekstowym formatem wymiany danych. W odróżnieniu od formatu klasycznego formatu utożsamiającego linię z rekordem złożonym z pól oddzielanych wskazanym separatorem może on w łatwy sposób opisywać bardziej złożoną (drzewiastą a nie tabelkową) postać danych. Dokument XML składa się z zagnieżdżonych w sobie znaczników, każdy z nich może posiadać atrybuty oraz wartość, którą jest tekst zawierający lub nie kolejne znaczniki. Kolejność występowania elementów w dokumencie jest znacząca. Każdy znacznik otwierający posiada odpowiadający mu znacznik zamykający (np. aa), znaczniki bez wartości mogą być samozamykające (np. <g />). Dokumenty HTML mogą być zgodne z wymogami formalnymi XML tym samym stanowiąc dokumenty XML.

```

/// Przykład C++
// wymaga pobrania biblioteki nagłówkowej rapidxml (http://rapidxml.sourceforge.net/)

#include "rapidxml.hpp"
namespace rapidxml { namespace internal {
    // fix bug in rapidxml https://sourceforge.net/p/rapidxml/bugs/16/
    template<class V, class C> V print_children(V, const xml_node<C>*, int, int);
    template<class V, class C> V print_element_node(V, const xml_node<C>*, int, int);
    template<class V, class C> V print_data_node(V, const xml_node<C>*, int, int);
    template<class V, class C> V print_cdata_node(V, const xml_node<C>*, int, int);
    template<class V, class C> V print_declaration_node(V, const xml_node<C>*, int, int);
    template<class V, class C> V print_comment_node(V, const xml_node<C>*, int, int);
    template<class V, class C> V print_doctype_node(V, const xml_node<C>*, int, int);
    template<class V, class C> V print_pi_node(V, const xml_node<C>*, int, int);
}}

```

```

#include "rapidxml_print.hpp"

#include <iostream>
#include <string.h>

int main() {
    char xmlString[1024];
    strncpy(xmlString,
        "<a>"
        "<b>A<h>qwe ... rty</h></b>"
        "ABCD...HIJ..."
        "<c x=\"q\" w=\"p p\">EE F</c>"
        "<g y=\"zz\" />"
        "<c x=\"pp\">123 <d rr=\"oo\">456</d> 78 90.</c>"
        "</a>",
        1024
    );

    // utworzenie obiektu drzewa XMLowego w oparciu o napis
    rapidxml::xml_document<> xmlDoc;
    xmlDoc.parse<0>(xmlString);

    // pobranie głównego węzła
    rapidxml::xml_node<>* xmlRoot = xmlDoc.first_node();

    std::cout << "nazwa głównego elementu to: " << xmlRoot->name() << "\n";
    std::cout << "jego zawartość tekstowa to: " << xmlRoot->value() << "\n";
    std::cout << "jego wartość to: {{{" << *xmlRoot << "}}}\n";

    std::cout << "jego potomkowie to: \n";
    rapidxml::xml_node<>* xmlNode = xmlRoot->first_node();
    while(xmlNode) {
        std::cout << " " << xmlNode->name() << " : ";
        rapidxml::print( std::cout, *xmlNode, rapidxml::print_no_indenting );
        // rapidxml::print() może zapisywać także do napisu
        std::cout << "\n";
        xmlNode = xmlNode->next_sibling();
    }

    std::cout << "pierwszy węzeł c ma atrybuty:\n";
    xmlNode = xmlRoot->first_node("c");
    if(xmlNode) {
        rapidxml::xml_attribute<>* xmlAtrib = xmlNode->first_attribute();
        while(xmlAtrib) {
            std::cout << xmlAtrib->name() << " = " << xmlAtrib->value() << "\n";
            xmlAtrib = xmlAtrib->next_attribute();
        }
    }

    // modyfikacje dokumentu:

    // zmiana nazwy i zawartości elementu
    xmlNode = xmlRoot->first_node("g");
    if (xmlNode) {
        xmlNode->name("noweGG");
        xmlNode->value("!@#$");
    }

    // zmiana nazwy i wartości atrybutu
    xmlNode = xmlRoot->first_node("c");
    if(xmlNode) {
        rapidxml::xml_attribute<>* xmlAtrib = xmlNode->first_attribute();
        while(xmlAtrib) {
            if (xmlAtrib->name() == std::string("w")) {
                xmlAtrib->name("uu");
                xmlAtrib->value("1 2 3");
                break;
            }
            xmlAtrib = xmlAtrib->next_attribute();
        }
    }

    // usuwanie wszystkich potomków ostatniego <c>
    xmlNode = xmlRoot->last_node("c");
    xmlNode->remove_all_nodes();
    xmlNode->value("");
}

```

```

// usuwanie wszystkich atrybutów ...
xmlNode->remove_all_attributes();
// są też funkcje usuwające wskazanego potomka lub atrybut:
// remove_node() i remove_attribute()

// dodawanie atrybutów
xmlNode->append_attribute(
    xmlDoc.allocate_attribute("abc", "098")
);
xmlNode->append_attribute(
    xmlDoc.allocate_attribute("qwe", "...")
);
// powyższa metoda dodaje na koniec, można także dodawać na początek:
// prepend_attribute() lub na wskazaną pozycję insert_attribute()

// dodawanie potomków
rapidxml::xml_node<*> xmlNode2;
xmlNode2 = xmlDoc.allocate_node(rapidxml::node_data, NULL);
xmlNode2->value("ert");
xmlNode->append_node(xmlNode2);

xmlNode2 = xmlDoc.allocate_node(rapidxml::node_element, "kk");
xmlNode2->value("uio");
xmlNode->append_node(xmlNode2);

xmlNode2 = xmlDoc.allocate_node(rapidxml::node_data, NULL);
xmlNode2->value("bnm");
xmlNode->append_node(xmlNode2);

// podobnie jak atrybuty nody też możemy dodawać na początku
// prepend_node() lub na dowolnej pozycji insert_node()

// wypisanie zmienionego dokumentu
std::cout << xmlDoc;
}

```

Przykład Python

```

import xml.etree.ElementTree as xml
# Python oferuje także inne niż ElementTree moduły do obsługi XML
# także wspierające DOM

d = """<a>
    <b>A<h>qwe ... rty</h></b>
    ABCD... &amp;&apos; HIJ...
    <c x="q" w="p p">EE FÅ</c>
    <g y="zz" />
    <c x="pp">123 <d rr="oo">456</d> 78 90.</c>
</a>"""

rootNode = xml.fromstring(d)

# pobieranie informacji z dokumentu

print("nazwa głównego elementu to:", rootNode.tag)
print("jego zawartość tekstowa to:", rootNode.text)
print("jego pełna zawartość tekstowa to:", "".join(rootNode.itertext()))
print("jego wartość to: {{{", xml.tostring(rootNode, encoding="unicode"), "}}}")

print("jego potomkowie to:")
for c in rootNode:
    print(" ", c.tag, ":", xml.tostring(c, encoding="unicode"))

print("pierwszy węzeł c ma atrybuty:")
try:
    ci = rootNode.iter("c")
    print(next(ci).attrib)
except StopIteration:
    print(" [brak takiego węzła]")

# modyfikacje dokumentu

# zmiana nazwy i zawartości elementu
try:
    ci = rootNode.iter("g")
    cc = next(ci);
    cc.tag = "noweGG"
    cc.text = "!@# $"

```

```

except StopIteration: pass

# zmiana nazwy i wartości atrybutu
try:
    ci = rootNode.iter("c")
    cc = next(ci);
    del cc.attrib["w"]
    cc.attrib["uu"] = "1 2 3"
except StopIteration: pass

try:
    cc = next(ci);
    # usuwanie wszystkich potomków drugiego <c>
    cc.clear()
    # jest też remove() które usuwa wskazany element

    # usuwanie wszystkich atrybutów ...
    cc.attrib.clear()

    se = xml.SubElement(cc, "kk", attrib={"aa":str(45)})
    se.text = "uio"
    se = xml.SubElement(cc, None)
    se.text = "bnm"

    # możliwe jest też wstawienie w podanym miejscu
    se = xml.Element(None)
    se.text = "ert"
    cc.insert(0, se)
except StopIteration: pass

print(xml.tostring(rootNode, encoding="unicode"))

```

/// Przykład XHTML+JavaScript

```

<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="pl">
<head>
  <meta http-equiv="ContentType" content="application/xhtml+xml; charset=utf-8" />
</head><body>

<div id="xyz">
  <p>
    <b id="abc">Lorem <i>ipsum</i></b> dolor sit amet, <i>consectetur</i> adipiscing elit.
    <ul class="typA typX">
      <li>Sed congue, eros quis ultricies ornare,</li>
      <li>massa sem auctor arcu, et semper ex arcu id augue.</li>
      <li>Fusce pretium <i><b>turpis</b></i> massa, maximus <i>dapibus</i> sit amet.</li>
      <li>Nulla fermentum molestie finibus.</li>
    </ul>Etiam accumsan <i>tempus</i> ante at congue.</p>
    <p data-abc="123" data-zz="0" class="az" id="q23y">
      Sed feugiat vestibulum sapien eget iaculis.<br />
      Integer quis magna nec lacus tempor sagittis.
    </p>
  </div>

<script type="text/javascript">
function fff() {
  console.log("wszystkie elementy &lt;i&gt; w dokumencie:");
  var a = document.getElementsByTagName("i");
  for (var j = 0; j &lt; a.length; j++) {
    console.log(a[j].innerHTML + "\n");
  }

  // modyfikacja zawartości elementu o id="abc"
  var b = document.querySelector("#abc");
  b.innerHTML = "ABC &lt;u&gt;ABC&lt;/u&gt; ABC";

  // potomkowie rodzica wskazanego elementu
  var child = b.parentNode.firstChild
  while (child) {
    console.log("potomek jest węzłem: " + child.nodeName + "\n");
    if (child.nodeName == "ul") {
      // jeżeli jest to lista &lt;ul&gt;
      // która posiada (wśród klass podanych w atrybucie class) klasę "typA"
      // to ją usuń i dodaj klasę "typB"
</pre>
</div>
```

```

        if (child.classList.contains('typA')) {
            child.classList.remove('typA');
            child.classList.add('typB');
        }
    }
    child = child.nextElementSibling;
}

var c = document.querySelector("#xyz");
// drugi potomek typu <p> elementu określonego w zmiennej c
var d = c.querySelector("p:nth-of-type(2)");
console.log("ma atrybuty:")
for (var i = 0; i < d.attributes.length; ++i) {
    console.log(" " + d.attributes[i].name + " = " + d.attributes[i].value + "\n");
}

// można też pytać o konkretny atrybut
console.log("abc => " + d.getAttribute("data-abc"));
// oraz ustawić (zarówno dodać nowy jak i zmienić istniejący)
d.setAttribute("data-def", "tyui");
}

addEventListener('DOMContentLoaded', fff, false);
//]]></script>

</body></html>

```

JSON

JavaScript Object Notation (JSON) jest tekstowym formatem wymiany danych. Podobnie jak XML reprezentuje on drzewiastą strukturę, jednak w odróżnieniu od XML kolejność elementów w danych wejściowych nie zawsze przekłada się na kolejność w danych zinterpretowanych. Wiąże się to z tym iż wyróżnia się dwa typy elementów. Pierwszy z nich obejmowany w nawiasy klamrowe i składa się z par klucz-wartość, jest on interpretowany jako słownik / tablica asocjacyjna i w jego wypadku identyfikacja elementu odbywa się wyłącznie po kluczu (który musi być unikalny), a kolejność nie ma znaczenia. Drugi jest obejmowany w nawiasy kwadratowe i składa się wyłącznie z kolejnych wartości, jest on interpretowany jako zwykła tablica lub lista.

```

### Przykład Python
import json
from pprint import pprint

a='''{
    "info": "bbb",
    "ver": 31,
    "d": [
        {"a": 21, "b": {"x": 1, "y": 2}, "c": [9, 8, 7]},
        {"a": 17, "b": {"x": 6, "y": 7}, "c": [6, 5, 4]}
    ]
}'''

# interpretacja napisu jako zbioru danych w formacie json
d = json.loads(a)

# wypisanie zbioru danych
pprint(d) # pprint ładnie formatuje złożone zbiory danych

# jak widać jest to zagnieżdżona struktura list i słowników
# odpowiadająca 1 do 1 temu co było w napisie

# dostęp do poszczególnych elementów: "po pythonowemu"
print(d["d"][1]["b"])
print(d["d"][1]["b"]["x"])
print(d["d"][1]["c"][1])

# przygotowaniem nowego zbioru danych
b={'aa': 'pp', 'b': [3,4,5], 'c':d}
b['e']="test"
b['f']={'a':1, 'b':2}

# wygenerowanie json-owego tekstu w oparciu o niego
c=json.dumps(b,indent=1,ensure_ascii=False)
print(c)

```

```
# dla porównania:
pprint(b)
```

wykonywanie napisu

Często wygodnie jest mieć możliwość wykonania kawałka kodu danego języka, który jest zapisany w zmiennej napisowej. Większość języków skryptowych pozwala to zrobić. Jako że funkcje te pozwalają na wykonywanie dowolnego kodu to szczególną ostrożność należy zachować przekazując do nich dane wprowadzane przez użytkownika.

Przykład Python

```
# dla kodu pythonowego
a="""print("ppp")
b=13-17
"""
exec(a)
print(b)

# dla wyrażeń
a="9+7"
b = eval(a)
print(b)

# lub
exec("c="+a)
print(c)

# oczywiście napis może pochodzić z
# dowolnego źródła, może być też
# treścią całego skryptu pythonowego:
# exec(open("plik.py").read())
```

Przykład Bash

```
# dla kodu bashowego
a='echo "ppp"; b="ooo"'
eval $a
echo $b

# dla obliczeń arytmetycznych
a="13+17"
b=$(( $a ))
echo $b

# alternatywnie przez
# zewnętrzny kalkulator np.:
b=`echo $a | bc`
echo $b
```

/// Przykład PHP

```
<?php
$a='echo "aaa"; $b="cc"';
eval($a);
echo "\n" . $b . "\n";

$a='2+6';
eval('$b=' . $a . ');');
echo $b . "\n";
?>
```

/// Przykład JavaScript

```
// dla kodu javascript
a="console.log('AAA'); b='www';"

eval(a);
console.log("b to: " + b + "\n");

// dla wyrażeń
a="5+7"
b=eval(a)
console.log("b to: " + b + "\n");
```

Iteratory

Iterator jest obiektem pozwalającym na dostęp do elementów jakiegoś innego obiektu (np. kolekcji typu lista, tablica, itd).

/// Przykład C++

```
// Już w powyższych przykładach użycia list i map w C++
// wykorzystane były iteratory, pozwalające na
// pobieranie kolejnych wartości z tych kontenerów:

void wypiszListe1(std::list<int> l) {
    for (std::list<int>::iterator i = l.begin(); i != l.end(); ++i) {
        std::cout << *i << "\n";
    }
}

// Iterator zwracają niektóre z metod tych kontenerów,
// np. .begin() zwraca iterator na pierwszy element.
// Zwiększanie iteratora odbywa się z użyciem operatora ++

// Wyjście poza zakres (zwiększenie iteratora wskazującego na
// ostatni element kolekcji) nie powoduje rzucenia wyjątku,
// za to iterator przyjmuje specjalną wartość oznaczającą koniec.
// Iterator o tej wartości zwracany jest przez metodę .end()
// (lub \cpp{.rend()}) przy iterowaniu w przeciwną stronę).

// Przy używaniu iteratorów w C++ wygodne jest korzystanie
// z typu auto. Typ ten zwalnia programistę z konieczności
// jawnego definiowania typu zmiennej do której przypisywana
// jest od razu jakaś wartość z określonym typem.
// Można napisać np. `auto x = 5;`, ale nie możemy napisać:
// `auto x; x = 5;`

void wypiszListe2(std::list<int> l) {
    for (auto i = l.begin(); i != l.end(); ++i) {
        std::cout << *i << "\n";
    }
}

// C++ udostępnia także inną składnię pętli for pozwalającą
// na iterowanie po wszystkich elementach kolekcji takich jak
// listy, mapy, itp. I upraszczającą powyższy zapis do postaci:

void wypiszListe3(std::list<int> l) {
    for (auto i : l) {
        std::cout << i << "\n";
    }
}

// Zamiast `auto i` można napisać `auto& i` aby otrzymać dostęp
// przez referencję (wtedy wykonanie przypisania wartości do i,
// np. `i = 0`, spowoduje modyfikację elementu listy).

// Warto zauważyć także, że w odróżnieniu od wcześniejszej pętli
// zmienna reprezentuje wyłuskany iterator (jest to wartość /
// referencja do wartości elementu a nie sam iterator).
```

Przykład Python

```
# niekiedy zamiast tworzenia listy lepsze może być
# uzyskiwanie jej kolejnych elementów "na żywo"
# funkcjonalność taką w pythonie zapewniają generatory:

def f(l):
    a, b = 0, 1
    for i in range(l):
        r, a, b = a, b, a + b
        yield r

# użycie generatora w pętli for
for i in f(16):
    print(i)

# można także tworzyć generatory nieskończone
def ff():
    a, b = 0, 1
```

```

while True:
    r, a, b = a, b, a + b
    yield r

# pobieranie kolejnych elementów
a = iter( ff() )
print( next(a) )
print( next(a) )

```

Podstawowe I/O

Standardowe wejście / wyjście

Typowo program posiada trzy strumienie danych: jeden wejściowy (stdin) i dwa wyjściowe (stdout i stderr). Jeżeli nie zostało dokonane przekierowanie strumieni (np. na plik lub strumień innego procesu) to standardowe wejście (stdin) powiązane jest z danymi wprowadzanymi interaktywnie z klawiatury, a standardowe wyjście (stdout) i standardowe wyjście błędu (stderr) z danymi wyświetlanymi na terminalu na którym uruchomiono program (i typowo później wyświetlanymi gdzieś na ekranie).

Standardowe wejście i wyjście może być wykorzystywane do:

- przekazywania między programami kolejnych etapów przetwarzania jakiegoś zbioru danych (zwykle tekstowego): np. `grep in.txt | sort > out.txt` (grep wybiera linie spełniające kryteria, a sort sortuje wynik)
- interaktywnej obsługi programu: program pyta o kolejne parametry i pobiera je z standardowego wyjścia (podejście należy stosować z rozważą gdyż takie programy ciężko używa się w skryptach - lepszym rozwiązaniem jest przyjmowanie parametrów z linii poleceń)

```

/// Przykład C++
#include <stdio.h>

int main() {
    //
    // wyjście / wyjście w stylu C
    //

    // wykorzystywane już wcześniej funkcje
    // puts i printf korzystają z stdout
    puts("Hello world");
    printf("0x%x == %d == %.3f\n", 13, 13, 13.0);

    // jeżeli chcemy korzystać z innego strumienia
    // należy użyć wariantu pozwalającego na podanie
    // pliku do którego ma się odbywać zapis:
    // stdout - standardowe wyjście
    // stderr - standardowe wyjście błędu
    fputs("Hello world 2", stderr);
    fprintf(stderr, "0x%x == %d == %.3f\n", 13, 13, 13.0);

    // wczytanie napisu z standardowego wejścia:
    char napis[10];
    fgets( napis, 10, stdin );
    // wczytany napis będzie miał 9 znaków + NULL-end

    // zapominamy o reszcie inputu jeżeli był dłuższy niż 9 znaków
    if (napis[8] != '\n') {
        int c;
        while((c = getchar()) != '\n' && c != EOF);
    }

    // wczytanie liczby z standardowego wejścia:
    int d;
    fscanf( stdin, "%i", &d);

    fprintf(stdout, "liczba: %d napis: \"%s\"\n", d, napis);
}

```

```

/// Przykład C++
#include <iostream>
#include <limits>
#include <iomanip>

int main() {
    //

```

```

// wyjście / wyjście w stylu C++
//

// standardowe wyjście i standardowe
// wyjście błędu jako strumień C++
std::cout << "Hello C++ world\n";
std::cerr << "Hello C++ world 2\n";

// strumienie pozwalają na proste wypisywanie zmiennych,
// ale jeżeli chcemy formatować wyjście to
// pisanie jest więcej niż w printf
int d = 1363;
std::cout << "d=" << d << " d/3=" << d/3.0 << "\n";
std::cout << std::setprecision(3) << 1.2342;
std::cout << " " << 23.567 << "\n";

// wczytanie napisu z standardowego wejścia:
char napis[10];
std::cin.getline(napis, 10);
if (std::cin.fail()) {
    // zapominamy o reszcie jeżeli była
    std::cin.clear();
    std::cin.ignore(
        std::numeric_limits<std::streamsize>::max(),
        '\n'
    );
}

// wczytanie liczby z standardowego wejścia:
std::cin >> d;

std::cout << "liczba to: " << d;
std::cout << " napis to: \"" << napis << "\"\n";
}

```

Przykład Python

```

import sys

# wypisywanie na standardowe wyjście
print("ABC: " + str(12) + " = " + hex(12))

# bez nowej linii na końcu
print("abc", end='')

# wypisywanie na standardowe wyjście błędu
print("QWE", file=sys.stderr)

# odczyt z wejścia
li = iter(sys.stdin)
liniaA = next(li)
liniaB = next(li)
print("l1: " + liniaA + "l2: " + liniaB)

# można też z użyciem metody czytającej jedną linię
liniaA = sys.stdin.readline()
liniaB = sys.stdin.readline()
print("l1: " + liniaA + "l2: " + liniaB)

# należy zauważyć że dane wczytywane są
# liniami i zawierają znak końca linii

# można także w ramach:
# for l in sys.stdin:
#
# lub skorzystać z metody wczytującej cały plik
# jako listę poszczególnych jego linii:
# ll = sys.stdin.readlines()
# bądź po prostu jako jeden napis
# napis = sys.stdin.read()

# jest też:
info = input("Wpisz coś: ")
print(info)

```

Przykład Bash

```

f1() {
    # wypisywanie na standardowe wyjście
    echo "Hello World"
    printf "%d %.3f\n" 123 13.15686

    # wypisywanie na standardowe wyjście błędu
    echo "Hello World 2" > /dev/stderr
    echo "ABC" > /dev/stderr
}

f2() {
    # odczyt ze standardowego wejścia
    # read wczytuje dane z stdin do podanych zmiennych
    # w taki sposób że do kolejnych zmiennych trafiają
    # kolejne słowa (napisy rozdzielane spacją lub
    # tabulatorem), a do ostatniej zmiennej reszta napisu
    # (do końca linii bez znaku końca linii)
    while read a b; do
        echo "${a}${b}"
    done
}

# przekierowania strumieni standardowych:

# wyjście z echo przekierowywane jest na wejście f2
echo -e "2 3\n1 6" | f2

# wyjścia f1 do odpowiednich plików
f1 > /tmp/out.txt 2> /tmp/err.txt

# użycie >> zamiast > spowoduje dopisywanie do pliku
# zamiast nadpisywania jego zawartości

# połączonych wyjść (normalnego i błędu) f1 do grep
f1 |& grep -v Hello

# standardowe wyjście może zostać przechwycone i
# podstawione w danym miejscu poprzez użycie
# `polecenie` lub ${polecenie}
echo XXX `ls -ld /tmp`
echo XXX ${ls -ld /tmp}

# często też chcemy zignorować standardowe wyjście i/lub
# standardowe wejście - możemy to uzyskać przekierowując
# je do /dev/null np:
grep '^root:' /etc/passwd > /dev/null

# przekierowanie wyjścia cat do pliku /tmp/liczby
# << EOF powoduje że bash podaje na standardowe
# wejście komendy (w tym wypadku "cat") dane czytane
# z skryptu (lub swojego wejścia) dopóki nie wystąpi
# w nowej linii słowo podane po << (w tym
# wypadku EOF), jeżeli słowo to jest ujęte w '' to
# w przekazywanym tekście nie są dokonywane podstawienia
# shellowe (np. rozwijane zmienne)
cat << EOF > /tmp/liczby
1 3
13 9
9 4
7 10
EOF

# przekierowanie pliku /tmp/liczby
# na standardowe wejście f2
f2 < /tmp/liczby

```

/// Przykład PHP

ze względu na specyfikę tego języka
wszystko co jest poza znacznikami php
będzie wypisywane na standardowe wyjście

```

<?php
    echo "są także funkcje ";
    print("realizujące wypisywanie\n");
    printf("%d %.3f\n", 13, 45.33221);
?>

```

Obsługa plików

Zasadniczo obsługa plików pod względem operacji wykonywanych pomiędzy otwarciem a zamknięciem pliku niewiele różni się od obsługi standardowego wejścia i wyjścia. W zdecydowanej większości (jeżeli nawet nie we wszystkich) języków interfejs obsługi standardowego wejścia i wyjścia jest zunifikowany z interfejsem obsługi plików.

```
/// Przykład C++
#include <stdio.h>

int main() {
    //
    // obsługa plików w stylu C
    //

    // otwieramy plik określony w pierwszym argumencie,
    // w trybie określonym w drugim argumencie:
    // r - odczyt, w - zapis, a - dopisywanie,
    // + - dwukierunkowy (używane po r, w albo a)
    FILE *plik = fopen("/tmp/plik1.txt", "w+");

    // zapisujemy do pliku
    fputs("Hello World !!!\n", plik);
    fprintf(plik, "%.3f\n", 13.13131);

    // jako że są to operacje buforowane to aby mieć
    // pewność że to jest już w pliku należy wykonać
    // fflush(), nie jest to konieczne gdy zamykamy
    // plik (wtedy wykonywane jest z automatu)
    fflush(plik);

    int poz = ftell(plik);
    printf("aktualna pozycja w pliku to %d\n", poz);

    // przewijamy do początku
    // jest to równoważne rewind(plik);
    fseek(plik, 0, SEEK_SET);

    // wczytywanie z pliku
    char napis[10];
    fgets(napis, 10, plik);
    // wczytany napis będzie miał 9 znaków + NULL-end

    puts(napis);

    // powrot do poprzedniej pozycji
    fseek(plik, poz, SEEK_SET);

    // operacje binarne - w ten sposób możemy zapisywać
    // i odczytywać całe bufory z pamięci, czyli także napisy
    // zapis do pliku
    double x = 731.54112, y = 12.2;
    fwrite(&x, 1, sizeof(double), plik);
    fflush(plik);

    // przesunięcie do pozycji na której zapisywaliśmy
    fseek(plik, poz, SEEK_SET);
    // i odczyt z pliku ...
    fread(&y, 1, sizeof(double), plik);

    printf("zapisano: %f, odczytano: %f\n", x, y);

    // są także funkcje read() i write() działające w oparciu o
    // numeryczny deskryptor pliku uzyskiwany np. z funkcji open()
    // a nie obiekt FILE uzyskiwany z fopen()

    fclose(plik);
}
```

```
/// Przykład C++
#include <iostream>
#include <fstream>

int main() {
    //
    // obsługa plików w stylu C++

```

```
//
// tworzymy strumień związany z plikiem
std::fstream plik_w;
// otwieramy plik w trybie dopisywania
// ale pozwalającym na modyfikowanie obecnej treści
// aby to działało musi być to strumień in/out
plik_w.open(
    "/tmp/plik2.txt",
    std::fstream::ate|std::fstream::out|std::fstream::in
);

// tworzymy strumień (typu in) związany z plikiem ...
std::ifstream plik_r;
// otwieramy plik do czytania
plik_r.open("/tmp/plik1.txt");

// tryby możemy budować z następujących opcji:
// std::ios::in - odczyt (domyślna dla ifstream)
// std::ios::out - zapis (domyślna dla ofstream)
// std::ios::ate - ustawie pozycji na koniec pliku
// std::ios::app - dopisywanie bez możliwości zmiany zawartości
// std::ios::trunc - nadpisuje plik

std::cout << "Pozycja w pliku OUT: " << plik_w.tellp() << std::endl;
std::cout << "Pozycja w pliku IN: " << plik_r.tellg() << std::endl;

// pisanie i czytanie dokładnie jak dla cin cout ...
plik_w << "Hello World !!!";

// przesunięcie na 16 bajt bliku wejściowego
plik_r.seekg(16);
// i wczytanie czegoś
double x;
plik_r >> x;
std::cout << "wczytano: " << x << "\n";

// przesunięcie o 2 bajty do tyłu i wczytanie
plik_r.seekg(-2, std::ios::cur);
// inne tryby przesunąć to:
// ios::beg - od początku pliku (domyślny)
// ios::end - od końca pliku
plik_r >> x;
std::cout << "wczytano: " << x << "\n";

plik_w.seekp(-3, std::ios::end);
plik_w << 123 << "\n";

// nma koniec zamykamy pliki
plik_w.close();
plik_r.close();
}
```

Przykład Python

```
import os.path

# otwarcie pliku do odczytu
f=open("/etc/passwd", "r")
# funkcja pozwala na określenie kodowania pliku poprzez
# argument nazwany "encoding" (np. encoding='utf8'),
# domyślne kodowanie zależne jest od ustawień systemowych
# można je sprawdzić poprzez locale.getpreferredencoding()
#
# jeżeli plik ma być otwarty w trybie binarnym a nie
# tekstowym konieczne jest podanie flagi b w ramach
# drugiego argumentu

# odczyt po linii
l1 = f.readline()
l2 = f.readline()

# można także czytać z jawnym użyciem iteratorów:
l1 = iter(f)
l3 = next(l1)
l4 = next(l1)

print("l1: " + l1 + "l2: " + l2 + "l3: " + l3 + "l4: " + l4)
```

```

# albo w ramach pętli
i = 5
for l in f:
    print(str(i) + ": " + l)
    i += 1

# powrót na początek pliku
f.seek(0)

# odczyt jako tablica linii
ll = f.readlines()

print(ll)

# i kolejny raz ... jako jednolity tekst
f.seek(0)
print( f.read() )

f.close()

# jeżeli plik istnieje to:
if os.path.isfile("/tmp/plik3.txt"):
    # otwieramy w trybie do zapisu i odczytu
    # i ustawiamy się na końcu pliku celem dopisywania
    f=open("/tmp/plik3.txt", "r+")
    f.seek(0, 2)
else:
    f=open("/tmp/plik3.txt", "w")

# pobieramy aktualną pozycję w pliku
# (która w tym wypadku jest równa długości pliku)
pos = f.tell()

# jeżeli plik ma więcej niż 5 bajtów
if pos > 5:
    # to cofamy się o 3
    f.seek(pos-3)

f.write("0123456789")

f.close()

# obsługa plików binarnych
# wymagane jest dodanie flagi b w flagach funkcji open():
f=open("/tmp/plik1.txt", "rb")

# czytanie baj po bajcie
while True:
    b = f.read(1)
    if b == b"":
        break
    print(b)

f.close()

```

Czekanie na dane

Niekiedy program musi poczekać na jakieś dane (np. wprowadzane z standardowego wejścia przez użytkownika). Typowo funkcje odczytu czekają w nieskończoność na koniec wczytywanych danych (lub na koniec linii). Jeżeli program ma czekać na dane z jednego z kilku źródeł lub czekać z ograniczeniem czasowym przydatna jest funkcja `select`.

Przykład Python

```

import sys, os, select

rdfd, _, _ = select.select([sys.stdin], [], [], 3.0)
# select() przyjmuje 3 listy deskryptorów plików
# (czyli numerycznego identyfikatora otwartego pliku,
# zwracanego np. przez funkcję open()) oraz ilość sekund,
# którą ma czekać na początek danych.

# Pierwsza lista związana jest z plikami z których chcemy
# czytać, druga pisać, a trzecia z plikami na których
# czekamy na wyjątkowe warunki.

```

```

# Funkcja ta kończy działanie gdy pojawią się jakiekolwiek dane
# (nie czeka na koniec danych – EOF) i zwraca również 3 takie listy,
# ale zawierające jedynie deskryptory plików na których pożądana
# operacja jest możliwa (np. są dane do wczytania, można zapisać dane).

if not rdfd:
    print("czas minął")

for fd in rdfd:
    print("czytam z:", fd)
    a = os.read(fd.fileno(), 1024)
    # Do odczytu zastosowana została funkcja os.read() a nie metoda fd.read(),
    # wynika to z faktu, iż fd.read() czeka na EOF lub podaną ilość bajtów,
    # a os.read() wczytuje to co jest dostępne i ogranicza jedynie maksymalną
    # ilość wczytywanych danych (resztę możemy doczytać kolejnym wywołaniem).
    print("wczytałem:", a)

```

Mapowanie plików w pamięci

Możliwe jest także operowanie na plikach zmapowanych do pamięci z użyciem mmap, operacje wykonywane są wtedy na ich zawartości tak jak na tablicach / napisach C.

```

/// Przykład C++
// program korzysta z mechanizmów systemów POSIX
// i może nie działać na niekompatybilnych platformach

#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include <errno.h>
#include <string.h>
#include <stdio.h>

#define _MMAP_MODE_ MAP_PRIVATE // MAP_SHARED

// MAP_PRIVATE powoduje bez widoczności zmian w mapowanym pliku dla innych procesów
// MAP_SHARED pozwala współdzielić zmapowany obszar z innymi procesami,
// jest wymagane aby móc zmodyfikować plik z użyciem msync()

int main() {
    const char* fname = "/tmp/aaa.txt";
    int fsize;

    // pobieramy rozmiar pliku
    struct stat st;
    if (stat(fname, &st)) {
        if (errno == ENOENT) {
            printf("plik \"%s\" nie istnieje\n", fname);
            fsize = 0;
        } else {
            fprintf(stderr, "Error in stat(): %s\n", strerror(errno));
            // powyższe zadziała jak perror()
            return -1;
        }
    } else {
        fsize = st.st_size;
    }

    // otwieramy plik w trybie zapis-odczyt uzyskując deskryptor
    // (jeżeli plik nie istnieje zostanie utworzony z prawami 600)
    int fd = open(fname, O_RDWR|O_CREAT, (mode_t)0600);

    int bufSize = fsize + 20;

    // jeżeli chcemy operować na większym buforze niż rozmiar pliku musimy powiększyć plik
    if (ftruncate(fd, bufSize))
        perror("Error in ftruncate()");
    // operacja ta nie ma wpływu na zajmowane miejsce na dysku:
    // int fd = open("/tmp/x", O_RDWR|O_CREAT); ftruncate(fd, 100000000); close(fd);
    // utworzy plik /tmp/x którego rozmiar wg `ls` będzie 98MB a wg `du` będzie 0
    // alternatywnie można:
    // lseek(fd, bufSize-1, SEEK_SET); write(fd, "", 1); lseek(fd, 0, SEEK_SET)

```

```

// mapujemy plik do pamięci ... w trybie odczytu i zapisu
char *buf = (char*)mmap(NULL, bufSize, PROT_READ | PROT_WRITE, _MMAP_MODE_, fd, 0);
if (buf == MAP_FAILED) {
    perror("Error in mmap()");
    return -1;
}

buf[fsize]=0;
printf("Długość pliku: %d\n", fsize);
printf("Zawartość pliku: %s\n", buf);

// modyfikujemy przedostatni bajt w pliku
// (jeżeli trafimy na fragment wielo-bajtowego znaku to go popsujemy)
if (fsize > 2)
    buf[fsize - 2]='X';

fsize += snprintf(buf+fsize, bufSize-fsize, "Ala ma kota\n\n");
fsize += snprintf(buf+fsize, bufSize-fsize, "Kot ma Alę\n\n");
fsize += snprintf(buf+fsize, bufSize-fsize, "czyżby ...\n\n");

// do pliku chcemy zapisać "fsize" bajtów z "buf", czyli: buf[0] ... buf[fsize-1]
// nie chcemy zapisać kończącego napis znaku NULL (dodawanego przez snprintf),
// jeżeli nie doszło do przepełnienia bufora nie jest on wliczany w długość "fsize"
// i znajduje się w buf[fsize], zatem nie ma konieczności jawnego pomijania go
//
// jeżeli natomiast doszło do przepełnienia bufora jest on ostatnim znakiem w buforze,
// czyli buf[bufSize-1] ... zatem aby go pominąć należy ustawić fsize = bufSize - 1
if (fsize >= bufSize) {
    fprintf(stderr, "Error: przepełnienie bufora\n");
    fsize = bufSize - 1;
}
printf("Nowa długość pliku: %d\n", fsize);
printf("Nowa zawartość pliku: %s\n", buf);

// ustawiamy prawidłowy rozmiar pliku
if (ftruncate(fd, fsize))
    perror("Error in ftruncate()");

#if _MMAP_MODE_ == MAP_SHARED
// synchronizujemy zawartość bufora pamięci do pliku
if (msync(buf, fsize, MS_SYNC))
    perror("Error in msync()");
#else
// zapisujemy zmiany do pliku
if (write(fd, buf, fsize) < 0)
    perror("Error in write()");
#endif

// odmapowanie pliku z pamięci
if (munmap(buf, fsize))
    perror("Error in munmap()");

// zamknięcie pliku
close(fd);
}

```

Operacje na plikach

Często oprócz zapisywania i czytania plików zachodzi potrzeba sprawdzenia czy plik istnieje, wylistowania plików z jakiegoś katalogu, zmiany nazwy lub usunięcia pliku. Stosowne funkcje oferuje biblioteka standardowa C (w oparciu o nie realizowane są komendy shellowe wykonujące takie operacje), a także większość innych języków. W przypadku C++ można korzystać z biblioteki standardowej C lub biblioteki boost (jak w poniższym przykładzie).

```

/// Przykład C++
// przy kompilacji konieczne jest dodanie:
// -lboost_filesystem -lboost_system

#include <iostream>
#include <boost/filesystem.hpp>

int main() {
    // sprawdzenie czy plik istnieje
    boost::filesystem::path pFile( "/tmp/abc.txt" );
}

```

```

if (boost::filesystem::is_regular_file(pFile))
    std::cout << "istnieje\n";
else
    std::cout << "nie istnieje\n";

// listowanie katalogu i sprawdzanie typów
boost::filesystem::path pDir( "/tmp/" );
boost::filesystem::directory_iterator end_iter;
boost::filesystem::directory_iterator iter(pDir);

for(; iter != end_iter; ++iter) {
    if (boost::filesystem::is_regular_file(iter->path())) {
        std::cout
            << iter->path().filename().generic_string()
            << " jest plikiem\n";
        // wypisujemy tylko nazwę pliku
    } else if (boost::filesystem::is_directory(iter->path())) {
        std::cout
            << iter->path().generic_string()
            << " jest katalogiem\n";
        // wypisuje, y pełną ścieżkę
    } else {
        std::cout
            << iter->path().generic_string()
            << " jest czymś innym\n";
    }
}

boost::filesystem::rename("/tmp/a.txt", "/tmp/b.txt");
boost::filesystem::remove("/tmp/b.txt");
}

```

Przykład Python

```

import os

# sprawdzenie czy plik istnieje
if os.path.isfile("/tmp/abc.txt"):
    print("istnieje")
else:
    print("nie istnieje")

# listowanie katalogu i sprawdzanie typów
dl = os.listdir("/tmp/")
print (dl)

for f in dl:
    if os.path.isfile(f):
        print("\n" + f + "\n jest plikiem")
    elif os.path.isdir(f):
        print("\n" + f + "\n jest katalogiem")
    else:
        print("\n" + f + "\n jest czymś innym")

# można także rekurencyjnie wraz z podkatalogami
for currDir, dirs, files in os.walk('/tmp'):
    print("podkatalogi \n" + currDir + "\n:")
    for d in dirs:
        print(" " + os.path.join(currDir, d))

    print("pliki w \n" + currDir + "\n:")
    for f in files:
        print(" " + os.path.join(currDir, f))

# zmiana nazwy
os.rename("/tmp/a.txt" "/tmp/b.txt")

# usuwanie
os.remove("/tmp/b.txt")

```

Przykład Bash

```

# sprawdzenie czy plik istnieje
if [ -f "/tmp/abc.txt" ]; then
    echo "istnieje"
else
    echo "nie istnieje"
fi

```

```

# listowanie katalogu i sprawdzanie typów
for f in /tmp/*; do
    if [ -f "$f" ]; then
        echo "\"$f\" jest plikiem"
    elif [ -d "$f" ]; then
        echo "\"$f\" jest katalogiem"
    else
        echo "\"$f\" jest czymś innym"
        # np. kolejką lub urządzeniem ...
    fi
done

# alternatywne listowanie katalogu
ls /tmp | while read f; do
    echo "/tmp zawiera: $f"
done

# zmiana nazwy
mv /tmp/a.txt /tmp/b.txt

# usuwanie
rm /tmp/b.txt

```

Argumenty linii poleceń

System pozwala na przekazanie do uruchamianego programu zbioru argumentów / opcji, które program może dowolnie zinterpretować i wykorzystać w swoim działaniu.

Od systemu program otrzymuje nie zinterpretowaną tablicę argumentów. Za podział całego zbioru argumentów, na poszczególne elementy tablicy odpowiada program tworzący nowy proces. Typowo napis przekazany w linii poleceń dzielony jest na słowa (ciągi znaków, rozdzielane białymi znakami) i każde słowo stanowi osobny element tablicy, wyjątkiem są napisy ujęte w cudzysłowia które przekazywane są jako pojedynczy argument (element tablicy).

W świecie POSIXowym przyjętym standardem jest podawanie opcji jednoliterowych po pojedynczym myślniku z możliwością ich grupowania (np. `ls -la` to to samo co `ls -l -a`), a opcji długich po dwóch myślnikach. Ewentualne argumenty opcji podawane są zaraz po opcji i w przypadku opcji krótkich oddzielone od niej spacją, a w przypadku opcji długich po znakiem równości (np. `ls --color=auto`). Po opcjach występują argumenty pozycyjne (np. lista plików które ma wyświetlić `ls`).

Obsługę takiego stylu przekazywania opcji do programu (w oparciu o surową tablicę argumentów) zapewniają liczne biblioteki (także standardowe biblioteki różnych języków programowania).

```

/// Przykład C++
// przy kompilacji konieczne jest dodanie:
// -lboost_program_options

#include <boost/program_options.hpp>
#include <iostream>

// pełna deklaracja funkcji main dla systemów POSIX
// funkcja main otrzymuje:
// - liczbę elementów tablicy argumentów
// - tablicę argumentów (jest to tablica napisów typu C)
// - tablicę zmiennych środowiskowych
int main(int argc, char *argv[], char *envp[]) {
    std::cout << "program wywołano z " << argc << " argumentami\n";
    for (int i=0; i<argc; ++i)
        std::cout << "argument " << i << " to: " << argv[i] << "\n";

    // deklaracja opcji linii poleceń
    boost::program_options::options_description desc("Program options");
    desc.add_options()
        ("help,h", "show help message")
        ("load", boost::program_options::value<std::string>(),
            "load from \"arg\" file")
    ;

    // parsowanie opcji linii poleceń wraz z obsługą wyjątków
    boost::program_options::variables_map vm;
    try {
        boost::program_options::store(
            boost::program_options::parse_command_line(argc, argv, desc), vm
        );
    }
}

```

```

        boost::program_options::notify(vm);
    } catch(boost::program_options::error& e) {
        std::cerr << "Cmdline args error: " << e.what() << "\n\n";
        std::cerr << "Use --help to see full options description\n";
        return 2;
    }

    // przetwarzanie otrzymanych opcji
    if (vm.count("help")) {
        std::cout << desc << "\n";
    } else if (vm.count("load")) {
        std::cout << "Load from: " << vm["load"].as<std::string>() << "\n";
    } else {
        std::cout << "Wywołano bez opcji\n";
    }

    // jest też funkcja getopt_long() z biblioteki standardowej C

    // funkcja main powinna zwracać kod powrotu
    // typowo: gdy program zakończy się powodzeniem zero
    //          a gdy zakończy się błędem (nie zerowy) kod błędu
    return 0;
}

```

Przykład Python

```

import argparse

# parser argumentów linii poleceń
# do argumentów można się też dostawać bezpośrednio poprzez sys.argv:
# len(sys.argv) - ilość elementów tej tablicy
# str(sys.argv[0]) - napis odpowiadający nazwie wywołania programu
# str(sys.argv[1]) - napis odpowiadający pierwszemu argumentowi programu

parser = argparse.ArgumentParser(description='Opis programu')
parser.add_argument(
    'ARG', default='ABC', nargs='?',
    help='argument pozycyjny (opcjonalny z wartością domyślną)'
)
parser.add_argument(
    '-v', "--verbose", action="store_true",
    help='opcja typu przełącznik'
)
parser.add_argument(
    "--input", action="store",
    help='opcja z argumentem'
)
parser.add_argument(
    "--vector", action="store", metavar='V', nargs='*',
    help='opcja z wieloma argumentami'
)
args = parser.parse_args()

print(args)

```

Przykład Bash

```

# przetworzenie argumentów określonych w wywołaniu getopt
# dalsze argumenty dostępne są w $1, $2, itd po zakończeniu pętli
# : -> poprzedzająca opcja wymaga argumentu
# :: -> poprzedzająca opcja może mieć argument
# uwaga o ile wymagane argumenty mogą być rozdzielane od opcji spacją
# o tyle opcjonalne nie (krótkie należy podawać zaraz po opcji,
# długie rozdzielając =)
eval set -- "$(getopt -o xy::: -l opcja-x,opcja-y,opcja-z:: -- "$@"
while true; do
    case $1 in
        -x|--opcja-x) echo "X"; shift 1;;
        -y|--opcja-y) echo "Y, argument \"$2\""; shift 2;;
        -z|--opcja-z) echo "Z, argument \"$2\""; shift 2;;
        # tutaj $2 może być pusty (gdy nie podano), ale jest zdefiniowany
        --) shift; break;;
    esac
done

# inną komendą mogącą służyć przetwarzaniu opcji skryptu jest getopt

```

```

/// Przykład PHP
<?php
/*
możliwe jest używanie php w
trybie command-line jednak ze
względu na specyfikę i podstawowe
zastosowania tego języka jest
to przypadek dość egzotyczny

dlatego też zamiast obsługi opcji
przekazywanych z linii poleceń w
przypadku skryptów php większy
sens ma pokazanie obsługi opcji
przekazywanych w żądaniu HTTP
*/

// wysłanie ciasteczka
// musi być przed wysłaniem
// jakiegokolwiek treści
$val = time();
$val = "tmp $val";
setcookie("Ciasteczko", $val);

echo '<pre>';
echo "wysłałem ciasteczko: $val\n";

// tablica zawierająca zmienne
// przekazane w adresie URL
echo '$_GET: ';
print_r($_GET);

// tablica zawierająca zmienne
// przekazane w treści żądania
echo '$_POST: ';
print_r($_POST);

// tablica zawierająca zmienne
// przekazane w ciasteczkach
echo '$_COOKIE: ';
print_r($_COOKIE);

echo '</pre>';

?>

```

Procesy i wątki

Rozgałęzienie procesu - fork()

Aby w systemie mógł działać więcej niż 1 proces konieczna jest możliwość utworzenia nowego procesu (potomka) z poziomu procesu aktualnie działającego (rodzica). Możliwe są dwa podejścia:

- utworzenie "czystego" procesu uruchamiającego podany kod programu z podanymi argumentami (spawn)
- utworzenie kopii aktualnego procesu, która zacznie wykonywać się niezależnie od momentu rozgałęzienia (fork)

W przypadku zastosowania fork proces potomny otrzymuje kopię pamięci rodzica (ma dostęp do wszystkich jego zmiennych oraz zasobów uzyskanych przed fork()); dalsze operacje na zmiennych są niezależne). Po utworzeniu kopii procesu można (ale nie trzeba) zastąpić wykonywany w nim program innym poprzez funkcje z rodziny exec. Cechy te powodują że mechanizm fork jest bardziej elastyczny od spawn.

```

/// Przykład C++
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdio.h>

void getInput(int fd, int pid);

int main() {
    // najprostszą metodą uruchomienia innego programu jest funkcja system

```

```

int retcode = system("ls -l");
printf("Program uruchomiony przez system() zwrócił kod powrotu: %d\n",
      WEXITSTATUS(retcode)
);

// jeżeli chcemy odebrać standardowe wyjście
// (albo zapisać standardowe wejście) można skorzystać z popen
FILE* p = popen("uname -a", "r");
puts("Proces wypisał na swój stdout:");
char buf[256];
size_t s;
while ((s = fread(buf, sizeof(char), sizeof(buf), p))) {
    fwrite(buf, sizeof(char), s, stdout);
}

// tworzymy łącza nie nazwane - rury ...
int toSys[2];
if ( pipe(toSys) == -1 ) {
    perror("Error in pipe(toSys)");
    // perror wypisuje podany komunikat na stderr dodając do niego opis
    // błędu określonego przez errno, można też pobrać identyfikator błędu
    // lub jego opis - szczegóły w man 3 errno
    return 1;
}
// toSys[0] - odbiór z rury
// toSys[1] - wysyłanie przez rurę

int fromSys[2];
if ( pipe(fromSys) == -1 ) {
    perror("Error in pipe(fromSys)");
    return 1;
}

// rozgałęziamy proces
int pid;
switch (pid = fork()) {
    case -1: {
        // funkcja zwróciła -1 co oznacza błąd
        perror("Error in fork");
        return 1;
    }

    case 0: {
        // funkcja zwróciła zero co oznacza że jesteśmy w procesie potomnym

        // podmieniamy stdin i stdout potomka na odpowiednie
        // koniec łącz nienazwanych
        dup2(toSys[0], 0);
        dup2(fromSys[1], 1);

        // zamykamy końce łącz nienazwanych (te które używamy są już dostępne
        // jako stdin, stdout, innych nie potrzebujemy)
        close(toSys[0]); close(toSys[1]);
        close(fromSys[0]); close(fromSys[1]);

        // zastępujemy się innym procesem (w tym wypadku gerp dopasowujący napisy
        // zawierające znak z zakresu A-C) - w taki sposób w jaki robi to system()
        execl("/bin/sh", "sh", "-c", "grep --line-buffered '[A-C]'", (char *) 0);

        // oczywiście zamiast wywoływania innego programu można tutaj użyć własnego
        // kodu pracującego równolegle z programem głównym
    }

    default: {
        // funkcja zwróciła coś innego od 0 i -1 oznacza to że jesteśmy w procesie
        // macierzystym i otrzymaliśmy pid naszego dziecka ...

        // zamykamy niepotrzebne konce rur
        close(toSys[0]); close(fromSys[1]);

        // rury możemy obsługiwać jak pliki
        // i w ten sposób obsługiwana jest końcówka do zapisu
        FILE *pout = fdopen(toSys[1], "w");
        fprintf(pout, "Ała ma kota\n");
        fprintf(pout, "Kot ma coś\n");
        fflush(pout);
    }
}

```

```

// można też je obsługiwać postępując się bezpośrednio deskryptorem
// i tak obsługiwana jest rura do czytania
// rurę do czytania ustawiamy w trybie nie blokującym
fcntl(fromSys[0], F_SETFL, fcntl(fromSys[0], F_GETFL, 0) | O_NONBLOCK);

// czytanie zostało wyniesione jest do osobnej funkcji getInput()
// celem łatwiejszego jego powtarzania
getInput(fromSys[0], pid);

// potomek może potrzebować trochę czasu na przetworzenie danych
sleep(1);
getInput(fromSys[0], pid);

// zamykamy nasz koniec rury do pisanie
// pozwoli to potomkowi czekającemu na input zakończyć się
fclose(pout);

// sprawdzamy czy jest jeszcze jakiś input
// (konieczne np. gdy grep bez --line-buffered)
sleep(1);
getInput(fromSys[0], pid);

// zamykamy rurę do czytania
close(fromSys[0]);

// czekamy na zakończenie potomka
int ppid = wait(&retcode);
printf("potomek o PID=%d zakończył się z kodem %d\n",
      ppid, WEXITSTATUS(retcode)
);
}
}
}

void getInput(int fd, int pid) {
    char* buf[256];
    int len, gLen = 0;

    while ((len = read(fd, buf, 255)) > 0) {
        buf[len] = 0;
        gLen += len;
        printf("Odebrano od potomka %d:\n>>>>\n%s\n<<<<\n", pid, buf);
    }
    printf("Łącznie odebrano: %d\n", gLen);
}

```

Przykład Python

```

# napis który będzie podawany na standardowe wejście uruchamianego polecenia
inStr = "Ala ma kota\nKot ma psa\n..."

# Python pozwala na bezpośrednie stosowanie funkcji znanych z języka C
# takich jak system(), popen(), fork(), execl() itd poprzez moduł os.
# Odbyna się to w sposób analogiczny do użycia w języku C. Na przykład:

import os
os.system('echo -en "' + inStr + '" | grep -v A')

# Python zapewnia jednak także wygodny, zunifikowany sposób uruchamiania innych
# programów / poleceń powłoki poprzez moduł subprocess oraz rozgałęziania
# własnego procesu poprzez moduł multiprocessing.

print("""
#
# podstawy subprocess
#
""")
import subprocess

# uruchamiamy subprocess z grep'em
res = subprocess.run(["grep", "-v", "A"], input=inStr.encode(), stdout=subprocess.PIPE)
print("Kod powrotu to: " + str(res.returncode))
print("Standardowe wyjście z komendy to: " + res.stdout.decode())
# warto zwrócić uwagę na kodowanie i dekodowanie napisów
# (przekazywanych/odbieranych przez stdin/stdout) do / z utf-8

# jeżeli chcemy korzystać np. z znaków uogólniających powłoki lub podać

```

```

# komendę jako pojedynczy napis (a nie listę argumentów) to można użyć
# opcji shell=True:
subprocess.run(["ls -ld /etc/pa*"], shell=True)
# jeżeli potrzebujemy tylko rozbicia napisu na listę argumentów można
# użyć shlex.split()

# run() pozwala także (obok subprocess.PIPE) na przekazywanie
# istniejących deskryptorów (lub subprocess.DEVNULL, co ignoruje wyjście)
# w ramach stdin, stdout, stderr

# moduł subprocess oferuje także funkcję Popen() dającą większą kontrolę
# nad uruchamianiem komendy

print("""
#
# podstawy multiprocessing
#
""")
import multiprocessing, os, time
def fun1(txt, st, cnt):
    for _ in range(cnt):
        print("Proces", os.getpid(), "z st =", st, " i argumentem:", txt)
        time.sleep(st)

def fun2(c, e):
    # obsługa rury i zdarzenia
    c.send("Uwolnić mrożone truskawki")
    time.sleep(1.3)
    c.send([12, 13, { "ab" : 11, "cd" : "xx" }])
    c.close()
    e.clear()

if __name__ == '__main__':
    # przygotowanie potomków
    p1 = multiprocessing.Process(target=fun1, args=('Ala ma psa',1.5,4,))
    p2 = multiprocessing.Process(target=fun1, args=('pies ma kota',1,3,))

    print("mój pid to", os.getpid(), "... uruchamiam procesy")
    # uruchomienie potomków
    p1.start()
    p2.start()

    # czekanie na zakończenie p2
    p2.join()
    print("p2 się zakończył")

    # wymuszenie zakończenia p1 jeżeli nadal żyje
    if p1.is_alive():
        print("p1 żyje")
        p1.terminate()
    p1.join()
    print("p1 się zakończył")

    # rura i event

    # Pipe() tworzy nam parę obiektów multiprocessing.Connection
    # jednego będziemy używać w procesie głównym, a drugiego w potomnym
    pConn, cConn = multiprocessing.Pipe()

    # tworzymy obiekt typu Event ... jest to prosty sygnalizator
    # z flagą binarną zapewniający atomowość wykonywanych operacji
    evt = multiprocessing.Event()
    evt.set()

    # tworzymy i uruchamiamy kolejny podproces
    p1 = multiprocessing.Process(target=fun2, args=(cConn,evt,))
    p1.start()

    # odbieramy dane z połączenia dopóki event jest ustawiony
    try:
        while evt.is_set():
            if pConn.poll(0.1):
                print("otrzymałem: ", pConn.recv())
    except EOFError: pass

```

```

print("""
#
# pool i bariera w multiprocessing
#
""")
import multiprocessing, queue, os, time
from threading import BrokenBarrierError

def fun5Init(b):
    global bariera
    bariera = b

def fun5(arg):
    for i in range(arg['n']):
        print("Wątek", arg['n'], " iter =", i)
        time.sleep(arg['ts'])

    try:
        print("Wątek", arg['n'], " czekam na innych")
        if bariera.wait() == 0:
            print("wątki zsynchronizowane")
            # więcej nie używaj tej bariery:
            bariera.abort()
        except BrokenBarrierError:
            print("czekanie anulowane")

    return arg['n'] + arg['ts']

if __name__ == '__main__':
    # tworzymy barierę i warunek
    b = multiprocessing.Barrier(3)

    # tworzymy pool'a ... w ramach inicjalizacji
    # przekazujemy barierę i warunek
    pool = multiprocessing.Pool(
        processes=3, initializer=fun5Init, initargs=(b,)
    )

    # tworzymy listę argumentów dla funkcji
    # uruchamianej w procesach potomnych
    args = [
        { 'n': 3, 'ts': 0.44 },
        { 'n': 13, 'ts': 0.17 },
        { 'n': 2, 'ts': 0.33 },
        { 'n': 7, 'ts': 0.69 },
        { 'n': 4, 'ts': 0.49 },
    ]

    # tworzymy pulę 3 procesów potomnych i uruchamiamy w nich wskazaną
    # funkcję z kolejnymi zbiorami argumentów z podanej listy
    # uwaga: funkcja wywoływana przez map wraz z ewentualnymi zmiennymi
    # globalnymi musi być zdefiniowana przed wywołaniem Pool()
    results = pool.map(fun5, args)

    pool.close()
    pool.join()

```

Komunikacja międzyprocesowa

W systemie wielopprocesowym konieczne jest zapewnienie mechanizmów komunikacji pomiędzy procesami, zwłaszcza jeżeli grupa procesów ma realizować wspólne zadanie.

Jednym z takich mechanizmów (można powiedzieć że nawet podstawowym) jest pokazane wcześniej łącze nie nazwane (pipe) pozwalające na przekazywanie strumienia danych od jednego do kolejnego procesu. Podobnie działa łącze nazwane z tym że nie jest uzyskiwane w wyniku funkcji `pipe()` a otwarcia specjalnego pliku (utworzonego `mkfifo()`) przez dwa procesy (jeden do czytania drugi do pisania).

Istnieją także inne mechanizmy komunikacji (zarówno prostsze od łącz jak i bardziej od nich zaawansowane), z których część została opisana poniżej.

Sygnały

Najprostszą formą komunikacji międzyprocesowej są sygnały. Podstawowe sygnały niosą tylko i wyłącznie informację o swoim numerze (istnieją także sygnały rozszerzone, tzw. czasu rzeczywistego, które niosą także dodatkową wartość liczbową). Sygnał może zostać wysłany przez dowolny proces do

dowolnego innego procesu, przy czym warunkiem jego dostarczenia są odpowiednie uprawnienia (procesu użytkowników mogą wysyłać tylko do procesów tego samego użytkownika, root może wysyłać do wszystkich).

Każdy sygnał ma zdefiniowane domyślne zachowanie procesu po jego otrzymaniu. Wiele sygnałów związanych jest z standardowymi operacjami (np. wciśnięcie Ctrl+C, Ctrl+Z, Ctrl+/, wykonanie kill \$PID) i ma odpowiednie do tego zachowania domyślne. Proces może zablokować lub zdefiniować własną obsługę większości (ale nie wszystkich) sygnałów.

```
/// Przykład C++
#include <stdio.h>
#include <signal.h>
#include <fcntl.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>

// funkcje obsługi sygnału
void obsługaSygnału(int sig) {
    printf("Otrzymałem sygnał numer: %i\n", sig);
}

void akcjaSygnału(int sig, siginfo_t *info, void *context) {
    printf("Otrzymałem sygnał numer: %i z kodem %d i wartością %d od pid=%d\n",
        sig, info->si_code, info->si_value.sival_int, info->si_pid);
    // si_code określa powód wysłania sygnału
}

int main() {
    int sig, pid;

    pid = getpid();
    printf("Mój PID=%d\n", pid);

    // sygnały są asynchronicznym mechanizmem jednokierunkowej komunikacji
    // systemy POSIXowe oferują dwa rodzaje sygnałów - standardowe i czasu rzeczywistego
    // sygnały czasu rzeczywistego (w odróżnieniu od standardowych przekazujących jedynie
    // numer sygnału) zawierają także kod i wartość oraz są priorytetyzowane
    // (niższy numer => wyższy priorytet => dostarczany w pierwszej kolejności)

    // do zdefiniowania obsługi sygnału standardowego (do jego przechwycenia) służy funkcja:
    signal(SIGINT, &obsługaSygnału); // pojawia się m.in. w efekcie Ctrl+C
    signal(SIGTSTP, &obsługaSygnału); // pojawia się m.in. w efekcie Ctrl+Z
    // oczywiście można definiować różne funkcje dla różnych sygnałów
    // zamiast wskaźnika do funkcji możemy podać także SIG_IGN albo SIG_DFL
    // co oznacza odpowiednio ignorowanie tego sygnału lub jego domyślną obsługę

    // możliwe jest też maskowanie sygnałów
    sigset_t sigSet;

    // inicjujemy maskę sygnałów na pustą, na pełną byłoby sigfillset()
    if (sigemptyset(&sigSet))
        perror("Error in sigemptyset()");

    // dodajemy sygnał do maski
    // można także usuwać - sigdelset() i sprawdzać czy jest obecny - sigismember()
    // zamaskowany SIGTERM jest domyślnym sygnałem wysyłanym przez kill
    if (sigaddset(&sigSet, SIGTERM))
        perror("Error in sigaddset()");

    // ustawiamy maskę sygnałów na przygotowaną
    if (sigprocmask(SIG_SETMASK, &sigSet, NULL))
        perror("Error in sigprocmask()");

    // sygnały nie obsługiwane i nie blokowane przez proces obsługiwane są w sposób domyślny
    // na ogół prowadząc do zakończenia procesu ... sygnałów SIGKILL (wysyłanego przez kill -9)
    // oraz SIGSTOP nie można przechwycić ani zamaskować

    // do zdefiniowania obsługi sygnału czasu rzeczywistego służy funkcja sigaction wraz z strukturą
    // sigaction opisującą obsługę (można tak definiować także obsługę standardowych sygnałów):
    struct sigaction rtSigAct;
    // funkcja obsługi (można także podać SIG_IGN albo SIG_DFL)
    rtSigAct.sa_sigaction = &akcjaSygnału;
    // flagi ... SA_SIGINFO oznacza funkcję przyjmującą 3 argumenty a nie 1 jak w signal()
```

```

rtSigAct.sa_flags = SA_SIGINFO;
// maska sygnałów blokowanych w trakcie obsługi sygnału ... blokujemy wszystko co się da
sigfillset(&rtSigAct.sa_mask);
// ustawienie obsługi
sigaction(SIGRTMIN, &rtSigAct, NULL);

// wysłanie sygnału czasu rzeczywistego
// (funkcja pozwala też na wysyłanie standardowych sygnałów)
sigqueue(pid, SIGRTMIN, (const union sigval)123);

// oczekiwanie na sygnał:
puts("Czekam na dowolny sygnał");
pause();

// dodajemy SIGINT do zbioru
if (sigaddset(&sigSet, SIGINT))
    perror("Error in sigaddset()");

puts ("Czekam na sygnał z podanej maski");
puts (" inne będą obsługiwane ale funkcja nie zakończy się ...");
printf(" aby kontynuować Ctrl+C lub kill %d\n", pid);
if (sigwait(&sigSet, &sig))
    perror("Error in sigwait()");
printf("Czekanie zakończone po otrzymaniu sygnału: %d\n", sig);

puts ("Śpię przez 10s ... dowolny sygnał przerwie");
sleep(10); // jest też usleep() dla mikro sekund

puts ("Czekam przez 10.5s na sygnał z podanej maski");
siginfo_t sigInfo;
struct timespec timeOut;
timeOut.tv_sec = 10;
timeOut.tv_nsec = 500000000;
sig = sigtimedwait(&sigSet, &sigInfo, &timeOut);
if (sig<0) {
    if (errno == EINTR)
        puts("otrzymałem inny sygnał");
    else if (errno == EINVAL)
        puts("timeout");
    else
        perror("Error in sigtimedwait()");
} else {
    printf("Czekanie zakończone po otrzymaniu sygnału %d od pid=%d\n",
        sig, sigInfo.si_pid
    );
}
// jest też sigwaitinfo działająca jak sigtimedwait ale bez timeout'u
}

```

Przykład Python

```

import os, signal

pid = os.getpid()
print("Mój PID:", pid);

# obsługa sygnału
def obsługaSygnału(s, f):
    print("Otrzymałem sygnał numer:", s)

# SIGINT - pojawia się m.in. w efekcie Ctrl+C
signal.signal(signal.SIGINT, obsługaSygnału)

# SIGTSTP - pojawia się m.in. w efekcie Ctrl+Z
signal.signal(signal.SIGTSTP, obsługaSygnału)

print("Czekam na dowolny sygnał")
signal.pause()

print("Czekam na sygnał z podanego zbioru")
# w odróżnieniu od C inne będą obsługiwane dopiero po otrzymaniu
# sygnału ze zbioru (zostaną zakolejkowane)
sig = signal.sigwait([signal.SIGTERM, signal.SIGINT])
print("Czekanie zakończone po otrzymaniu sygnału:", sig)

print("Czekam przez 10.5s na sygnał z podanego zbioru")
# w odróżnieniu od C inny sygnał jest obsługiwany,

```

```
# ale nie przerywa czekania
sig = signal.sigtimedwait([signal.SIGTERM, signal.SIGINT], 10.5)
if sig == None:
    print("timeout");
else:
    print("Czekanie zakończone po otrzymaniu sygnału:",
        sig.si_signo, "od pid:", sig.si_pid
    )
```

Kolejki komunikatów

W przypadku gdy grupa procesów ma ze sobą współpracować potrzebne są bardziej zaawansowane (od sygnałów) mechanizmy komunikacji - pozwalające na przekazywanie jakiś danych między procesami. Jednym z takich mechanizmów są kolejki wiadomości. W pewnym stopniu są one podobne do omówionych wcześniej łącz (pipe-ów), jednak w odróżnieniu od nich zamiast jednolitego strumienia danych pomiędzy dwoma procesami (producentem i konsumentem) służą do przekazywania wiadomości pomiędzy grupą producentów i grupą konsumentów.

```
/// Przykład C++
// przy kompilacji konieczne jest dodanie:
// -lrt -lpthread

#include <stdio.h>
#include <queue.h>
#include <errno.h>
#include <time.h>
#include <string.h>
#include <unistd.h>

void getMsg(union sigval sv) {
    puts("kolejka nie jest już pusta");
    // tu mógłby być realizowany odbiór wiadomości z kolejki
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "%s /nazwa 1|2\n", argv[0]);
        return 1;
    }
    char runType = argv[2][0];

    // kolejki wiadomości POSIX (więcej man 7 mq_overview)
    // kolejka może mieć zarówno wielu producentów jak i odbiorców
    // raz odebrana wiadomość znika z kolejki

    // otwarcie kolejki
    struct mq_attr atrybuty;
    atrybuty.mq_maxmsg=3; // pojemność kolejki - 3 wiadomości
    atrybuty.mq_msgsize=20; // po 20 bajtów każda
    mqd_t msgQ = mq_open(
        argv[1], // nazwa zasobu
        0_RDWR|(runType == '1' ? 0_CREAT|0_EXCL : 0), // flagi
        0770, // prawa dostępu
        &atrybuty
    );
    if (msgQ == (mqd_t) -1) {
        perror("Error in mq_open()");
        return -1;
    }

    char buf[20];
    unsigned priorytet;
    struct timespec timeout;
    clock_gettime(CLOCK_REALTIME, &timeout);
    timeout.tv_sec+=2;

    // odbieramy wiadomość z timeoutem ...
    // wiadomości odbierane są w kolejności priorytetów, a następnie wysłania
    ssize_t size = mq_timedreceive(msgQ, buf, 20, &priorytet, &timeout);
    // mq_receive() - wersja bez timeoutu
    // mamy także możliwość odbioru bez timeoutu bądź zażądania powiadomienia
    // o wpisaniu pierwszej wiadomości do niepustej kolejki - mq_notify()
    if (size > 0){
        printf("odebrałem %d bajtów z priorytetem %d: %s\n", size, priorytet, buf);
    } else {
```

```

        if (errno == ETIMEDOUT)
            puts("timeout");
        else
            perror("Error in mq_timedreceive()");
    }

    // ustawienie powiadamiania o pojawieniu się wiadomości w niepustej kolejce
    struct sigevent sev;
    sev.sigev_notify = SIGEV_THREAD;
    sev.sigev_notify_function = getMsg;
    sev.sigev_notify_attributes = NULL;
    if (mq_notify(msgQ, &sev) == -1) {
        perror("Error in mq_notify()");
    }

    // wysyłamy wiadomość do kolejki
    // funkcja ta zawiesi proces gdy nie ma miejsca w kolejce która nie jest
    // otwarta z O_NONBLOCK na czas określony przez timeout
    // mq_send() - wersja bez timeout'u
    snprintf(buf, 20, "info A od: %d", getpid());
    clock_gettime(CLOCK_REALTIME, &timeout);
    timeout.tv_sec+=2;
    if (mq_timedsend(msgQ, buf, strlen(buf), 9, &timeout))
        perror("Error in mq_timedsend()");
    printf("Wysłano: %s\n", buf);

    if (runType == '1') {
        snprintf(buf, 20, "info B od: %d", getpid());
        if (mq_send(msgQ, buf, strlen(buf), 9))
            perror("Error in mq_send()");
        printf("Wysłano: %s\n", buf);

        sleep(20);
        mq_unlink(argv[1]);
        // w przypadku gdyby program zakończył się bez wykonania mq_unlink()
        // (np. na skutek kill -9) konieczne jest ręczne usunięcie plików z /dev/mqueue
    }
}

```

Przykład Python

```

print("""
#
# kolejki w multiprocessing
#
""")
import multiprocessing, multiprocessing.managers, queue, os, time

def fun3(q):
    # obsługa kolejki ... zakończenie można także realizować
    # np. w oparciu o dane otrzymane w kolejce
    while True:
        if not q.empty():
            el = q.get(block=False)
            # można także uruchamiać blokujący get()
            # z timeoutem lub bez ...
            if el[1] == None:
                break;
            else:
                print("potomek otrzymał", el)
                time.sleep(0.77)

if __name__ == '__main__':
    # tworzymy kolejkę
    # moduł multiprocessing zawiera swoje implementacje kolejki FIFO
    # (Queue, SimpleQueue i JoinableQueue), których można bezpiecznie
    # używać pomiędzy procesami:
    # q = multiprocessing.Queue()
    #
    # chcemy jednak użyć kolejki priorytetowej z modułu queue
    # (oferuje także kolejki FIFO (Queue) i LIFO (LifoQueue))
    # aby móc z niej poprawnie korzystać w multiprocessing
    # musimy użyć jej poprzez SyncManager'a:
    class Manager(multiprocessing.managers.SyncManager):
        pass
    Manager.register("PrioQueue", queue.PriorityQueue)
    m = Manager()
    m.start()

```

```

q = m.PrioQueue()

# dodajemy elementy do kolejki
q.put((100, "delfin ma psa ;-"))
q.put((70, [13, 17]))
q.put((200, [15, { "ab" : 11, "cd" : "xx" }]))

# tworzymy i uruchamiamy kolejny podproces
p1 = multiprocessing.Process(target=fun3, args=(q,))
p1.start()

# wkładanie danych do kolejki
time.sleep(0.404)
print("dodaje nowe wiadomości do kolejki ...")
q.put((13, "ABC ... i co dalej"))
q.put((180, "bla bla bla ..."))

# informacja dla potomka o zakończeniu obsługi kolejki
print("czekam na opróżnienie kolejki ...")
while q.qsize() > 0:
    time.sleep(0.333)

# kończenie
q.put((999, None))
p1.join()
m.shutdown()

```

Pamięć współdzielona

Kolejnym mechanizmem komunikacji międzyprocesowej jest współdzielenie części pamięci, które pozwala na operowanie na tych samych danych przez grupę procesów (a nie tylko przekazywanie danych od procesu do procesu jak wyżej opisane metody). Przy pomocy tego mechanizmu możliwe jest stworzenie własnej implementacji łącz (pipe-ów) czy też kolejek wiadomości.

W przypadku korzystania z wspólnej pamięci konieczne jest mechanizm zabezpieczenia przed możliwością równoczesnej modyfikacji tego samego fragmentu przez kilka procesów. Pierwszym nasuwającym się rozwiązaniem jest zastosowanie zmiennej współdzielonej informującej o tym że ktoś blokuje dany fragment celem jego modyfikacji np.:

```

int blokada=false;
zrownoleglijKod();
if (!blokada) {
    blokada=true;
    sekcjaKrytyczna();
    blokada=false;
}

```

Niestety takie rozwiązanie nie gwarantuje skuteczności blokady, gdyż:

1. może się zdarzyć że na dwóch rdzeniach kod jest wykonywany dokładnie równolegle i dwa procesy sprawdzą wartość zmiennej "blokada" w tej samej chwili a dopiero potem zmienią jej wartość (w efekcie czego oba wejdą do sekcji krytycznej wejdą oba procesy)
2. nastąpi przełączenie procesów wykonywanych na jednym rdzeniu pomiędzy sprawdzeniem warunku a modyfikacją zmiennej, co pozwoli wejść drugiemu procesowi do sekcji krytycznej gdy pierwszy już sprawdził warunek wejścia

Dlatego też korzysta się z mechanizmów systemowych (semaforów, lock'ów, etc) które zapewniają atomowość (nie podzielność) operacji sprawdzenia wartości zmiennej i jej modyfikacji.

```

/// Przykład C++
// przy kompilacji konieczne jest dodanie:
// -lrt -lpthread

#include <stdio.h>
#include <semaphore.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <errno.h>
#include <time.h>

#define BUF_SIZE 128

int main(int argc, char *argv[]) {
    if (argc != 3) {

```

```

    fprintf(stderr, "%s nazwa 1|2\n", argv[0]);
    return 1;
}
int retcode = 1;
char runType = argv[2][0];

// pamięć dzielona przez shm i mmap (więcej `man 7 shm_overview`)
// można by także przez mmap i zwykły plik

// 1) utworzenie zasobu (pliku) SHM
//   jeżeli runType == '1' i zasób istnieje to wywołanie zakończy się błędem
//   ja jeżeli nie istnieje zostanie utworzony
int shmFd = shm_open(
    argv[1], // nazwa zasobu
    O_RDWR|(runType == '1' ? O_CREAT|O_EXCL : 0), // flagi
    0770 // prawa dostępu
);
if (shmFd < 0) {
    perror("Error in shm_open()");
    return 1;
}

// 2) ustalenie wielkości
if (ftruncate(shmFd, BUF_SIZE)) {
    perror("Error in ftruncate()");
    goto END_1;
}

// 3) zamapowanie zasobu SHM w pamięci
void *addr = mmap(0, BUF_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, shmFd, 0);
if (addr == MAP_FAILED) {
    perror("Error in mmap()");
    goto END_1;
}

// semafor POSIX (więcej `man 7 sem_overview`)

// 1) utworzenie semafora
//   jeżeli runType == '1' i semafor istnieje to wywołanie zakończy się błędem
sem_t* sem = sem_open(
    argv[1], // nazwa zasobu
    O_RDWR|O_CREAT|(runType == '1' ? O_EXCL : 0), // flagi
    0770, // prawa dostępu
    1 // wartość semafora
);
if (sem == SEM_FAILED) {
    perror("sem_open");
    goto END_1;
}

int* c = (int*) addr;
if (runType == '1') {
    *c = 0;
}

// próba wejścia w kod zabezpieczony semaforem
if (sem_trywait(sem)) {
    if (errno == EAGAIN)
        puts("semafor nie wpuszcza ... trzeba spróbować później");
    else
        perror("Error in sem_trywait()");
} else {
    // semafor został opuszczony ... jesteśmy w sekcji krytycznej
    *c = *c + 0x01;
    printf("c = 0x%x\n", *c);

    sleep(3); // śpiemy, to nieładnie spać w sekcji krytycznej, ale to tylko demo ...

    // podniesienie semafora
    if (sem_post(sem))
        perror("Error in sem_post()");
}

sleep(5);

// czekanie na możliwość wejścia w kod zabezpieczony semaforem z timeout'em

```

```

struct timespec timeout;
clock_gettime(CLOCK_REALTIME, &timeout);
timeout.tv_sec+=10;
if (sem_timedwait(sem, &timeout)) {
    if (errno == ETIMEDOUT)
        puts("timeout ... trzeba spróbować później");
    else
        perror("Error in sem_timedwait()");
} else {
    // semafor został opuszczony ... jesteśmy w sekcji krytycznej
    *c = *c + 0x0100;
    printf("c = 0x%x\n", *c);

    sleep(5); // śpimy, to nieładnie spać w sekcji krytycznej, ale to tylko demo ...

    // podniesienie semafora
    if (sem_post(sem))
        perror("Error in sem_post()");
}

sleep(3);

// czekamy do skutku na możliwość wejścia w sekcję krytyczną
if (sem_wait(sem)) {
    perror("Error in sem_timedwait()");
} else {
    // semafor został opuszczony ... jesteśmy w sekcji krytycznej
    *c = *c + 0x010000;
    printf("c = 0x%x\n", *c);

    sleep(5); // śpimy, to nieładnie spać w sekcji krytycznej, ale to tylko demo ...

    // podniesienie semafora
    if (sem_post(sem))
        perror("Error in sem_post()");
}

retcode = 0;

// o ile w większości przypadków zwalnianie i zamykanie zasobów w momencie
// kończenia programu nie ma znaczenia (zasoby są automatycznie odbierane przez
// system - pamięć będzie zwolniona, pliki, gniazda sieciowe pozamykane, itd)
// to w przypadku zasobów współdzielonych może mieć istotne znaczenie
// np. jeżeli nie usuniemy zasobu SHM o danej nazwie nie będzie go można później
// otworzyć gdy używamy flag O_CREAT|O_EXCL
//
// dlatego w powyższym kodzie w obsłudze błędów używamy goto do sekcji kończącej:
END_2:
if (runType == '1') {
    // usunięcie semafora (stanie się niedostępny także dla innych procesów)
    sem_unlink(argv[1]);
}
END_1:
// odmapowanie zasobu z pamięci
munmap(addr, BUF_SIZE);
// zamknięcie zasobu SHM
close(shmFd);
if (runType == '1') {
    // usunięcie zasobu SHM (stanie się niedostępny także dla innych procesów)
    shm_unlink(argv[1]);
}
// w przypadku gdyby jednak program zakończył się bez wykonywania tego kodu
// (np. na skutek kill -9) konieczne jest ręczne usunięcie plików z /dev/shm
}

```

Przykład Python

```

print("""
#
# współdzielenie danych w multiprocessing
#
""")
import multiprocessing, queue, os, time

def fun4(lock, l, c):
    # obsługa zmiennej warunkowej
    for i in range(6):
        # blokujemy dostęp do listy

```

```

    c.acquire()
    # modyfikujemy zmienną na której mamy warunek
    l[0] = i
    # powiadamiamy wszystkich o tym
    c.notify_all()
    # zwalniamy dostęp do listy
    c.release()
    time.sleep(0.13)

# obsługa locka i listy
while True:
    if lock.acquire(timeout=0.5):
        # jeżeli udało się założyć lock'a ...
        l[1] += 1
        print("podprocess:", l)
        time.sleep(0.77)
        # zdjęcie lock'a
        lock.release()
        time.sleep(0.33)
    else:
        print("podprocess: can't lock")

if __name__ == '__main__':
    # tworzymy lock'a, czyli blokadę wejścia do sekcji krytycznej
    # (zmieniającej dane współdzielone)
    lock = multiprocessing.Lock()
    # oprócz zwykłych Lock'ów Python oferuje także:
    # * RLock - Lock w którym wątek zakładający blokadę może ją zakładać
    #   kolejne razy bez jej uprzedniego zwalniania, tzn sekwencja:
    #   l = RLock();
    #   l.acquire(); l.acquire();
    #   cos();
    #   l.release(); l.release();
    # jest poprawna (przy zwykłym Lock'u zatrzyma się na drugim
    # l.acquire())
    # * Semaphore - Lock z licznikiem dozwolonych wejść do sekcji
    # krytycznej np. l = Semaphore(value=3) pozwoli na 3 wywołania
    # l.acquire() (bez l.release())

    # tworzymy współdzieloną listę ... na menagerze
    m = multiprocessing.Manager()
    l = m.list([0, 1, 2, 3])

    # tworzymy zmienną warunkową używającą lock'a lock
    c = multiprocessing.Condition(lock)

    # tworzymy i uruchamiamy kolejny podproces
    p1 = multiprocessing.Process(target=fun4, args=(lock,l,c,))
    p1.start()

    print("czekamy na spełnienie warunku ...");
    c.acquire()
    while l[0] < 4:
        print("czekamy ... l[0]=", l[0])
        c.wait()
    c.release()
    print("warunek spełniony ... l[0] = ", l[0]);

    time.sleep(0.17)
    # obsługa listy
    for i in range(3):
        # założenie lock'a
        print("process: czekam na lock ...")
        lock.acquire()

        # dodanie elementu do listy i wypisanie
        l.append(13 + i)
        print("process:", l)
        time.sleep(0.69)

        # zdjęcie lock'a
        lock.release()
        time.sleep(1.0)

    # zakończenie
    p1.terminate()
    p1.join()

```

```
m.shutdown()
```

Wątki

Oprócz możliwości rozgałęzienia procesu (utworzenia potomka) z użyciem np. `fork()` możliwe jest także tworzenie wątków (zwanym też lekkimi procesami) w ramach procesu. Podstawowa różnica między procesem a wątkiem polega na tym iż wszystkie wątki (w ramach danego procesu) współdzielą całość pamięci (przestrzeni adresowej), podczas gdy każdy z procesów posiada niezależną pamięć i co najwyżej jakiś współdzielony z innymi fragment. Każdy z wątków posiada natomiast niezależny stos (umieszczony we wspólnej przestrzeni adresowej), który jest używany m.in. do przechowywania zmiennych lokalnych (w tym argumentów funkcji). Jednak ze względu na umieszczenie tych danych w wspólnej przestrzeni adresowej są one także dostępne dla innych wątków (poprzez wskaźnik lub referencję).

```
/// Przykład C++
// przy kompilacji konieczne jest dodanie:
// -lrt -lpthread

#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

struct watekInfo {
    pthread_mutex_t* mut;
    pthread_cond_t* cond;
    const char* napis;
    int licznik;
};

void wypisz1(struct watekInfo* wi) {
    int i, l = -99, ret;
    for(i=0; i<4; i++) {
        // próbujemy zablokować wejście na ten kawałek kodu
        // można czekać do skutku przy pomocy pthread_mutex_lock()
        if ((ret = pthread_mutex_trylock(wi->mut)) != 0) {
            fprintf(stderr,
                "Wątek: error in pthread_mutex_trylock(): %s\n",
                strerror(ret)
            );
        } else {
            wi->licznik += 2;
            l = wi->licznik;
            pthread_mutex_unlock(wi->mut);
        }

        printf("Wątek: %s %d\n", wi->napis, l);
        usleep(600000);
    }
    sleep(8);
    pthread_exit(0); // kończy działanie wątku
}

void wypisz2(struct watekInfo* wi) {
    int i;
    printf("To watek: %s\n", wi->napis);

    pthread_mutex_lock(wi->mut);
    for(wi->licznik=0; wi->licznik<10; wi->licznik++) {
        printf("."); fflush(stdout);
        if (wi->licznik==5)
            pthread_cond_signal(wi->cond);
        sleep(1);
    }
    pthread_mutex_unlock(wi->mut);

    // zabraniamy przerywania wątku w trakcie poniższego wypisywania
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    for(i=0; i<10; i++) {
        printf("."); fflush(stdout);
        sleep(1);
    }
    printf("\n");
}
```

```

// a teraz już może być anulowany
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

// ale od teraz tylko w wyznaczonych momentach
// na funkcjach takich jak sleep, system, ...
// czy też pthread_testcancel()
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);

sleep(50);

puts("koniec wypisz2");
pthread_exit(0);
}

int main() {
    int i, l, ret;

    // przygotowanie mutex'a
    pthread_mutexattr_t mutAttr;
    pthread_mutexattr_settype(&mutAttr, PTHREAD_MUTEX_ERRORCHECK);
    // możemy ustawić:
    // * PTHREAD_MUTEX_ERRORCHECK aby mechanizm reagował błędem na
    //   unlockowanie załokowanego przez inny wątek mutexu itp,
    // * PTHREAD_MUTEX_RECURSIVE aby blokada nałożona przez wątek A
    //   nie obowiązywała wątku A (umożliwia to korzystanie z
    //   mutexów w funkcjach rekurencyjnych)
    // * PTHREAD_MUTEX_NORMAL lub PTHREAD_MUTEX_DEFAULT
    // flagi te mogą się nieco różnić w zależności od stosowanej
    // biblioteki i użytych #define szczegółów należy szukać
    // w plikach features.h pthread.h oraz dokumentacji man

    pthread_mutexattr_setprotocol(&mutAttr, PTHREAD_PRIO_NONE);
    // poza nic nie zmieniającym PTHREAD_PRIO_NONE mamy także:
    // * PTHREAD_PRIO_INHERIT zabezpieczający wątki przed inwersją
    //   priorytetów (priorytet po wejściu do sekcji krytycznej
    //   jest podwyższany do najwyższego z priorytetów procesów
    //   czekających
    // * PTHREAD_PRIO_PROTECT zabezpieczający również przed
    //   zakleszczeniem dzięki podnoszeniu tego priorytetu powyżej
    //   najwyższego z priorytetów procesów czekających
    //
    // do zapobiegania inwersji priorytetów przydatne jest też:
    // pthread_mutexattr_setprioceiling();

    // inicjalizujemy mutex (dla domyślnych ustawień można podać
    // NULL jako drugi argument)
    pthread_mutex_t mut;
    pthread_mutex_init(&mut, &mutAttr);

    // przygotowanie danych dla wątku
    struct watekInfo wInfo;
    wInfo.napis = "Ala ma kota";
    wInfo.licznik = 10;
    wInfo.mut = &mut;

    // tworzymy nowy wątek uruchamiając w nim funkcję wypisz1
    // i przekazując jako pierwszy argument tej funkcji wInfo
    pthread_t watek;
    if ((ret = pthread_create(
        &watek, 0, (void* (*)(void*))wypisz1, (void*)&wInfo ))
        != 0
    ) {
        fprintf(stderr, "Error in pthread_create(): %s\n",
            strerror(ret));
        return 1;
    }

    for(i=0; i<7; i++) {
        pthread_mutex_lock(wInfo.mut);
        wInfo.licznik -= 1;
        printf("Program główny: %d\n", wInfo.licznik);
        pthread_mutex_unlock(wInfo.mut);
        sleep(1);
    }

#ifdef _GNU_SOURCE
    // pobranie atrybutu wątku

```

```

// i sprawdzenie czy jest wątkiem procesu czy systemu
pthread_attr_t attrbuty;
if ((ret = pthread_getattr_np(watek, &attrbuty)) {
    fprintf(stderr, "Error in pthread_getattr_np(): %s\n",
        strerror(ret));
    return 1;
}
if ((ret = pthread_attr_getscope(&attrbuty, &i)) {
    fprintf(stderr, "Error in pthread_attr_getscope(): %s\n",
        strerror(ret));
    return 1;
}
if (i == PTHREAD_SCOPE_PROCESS) {
    puts("wątek procesu");
} else if (i == PTHREAD_SCOPE_SYSTEM) {
    puts("wątek systemu ... będzie widoczny w ps -m:");
    system("ps -m");
} else {
    puts("inny wątek :-o");
}
#endif

// możemy oczekiwać na zakończenie wątku poprzez wywołanie
// w innym funkcji pthread_join() umożliwia ona także
// uzyskanie kodu z jakim zakończył się wątek
printf("Program główny czeka na koniec wątku\n");
if ((ret = pthread_join(watek, NULL)) {
    fprintf(stderr, "Error in pthread_join(): %s\n",
        strerror(ret));
    return 1;
}
printf("Wątek zakończył się\n");
// jej wywołanie jest konieczne aby wątek po zakończeniu
// był w całości usunięty, chyba że wątek został określony
// jako taki do którego nie można się dołączyć przez wywołanie
// pthread_detach() lub podanie przy tworzeniu flagi
// PTHREAD_CREATE_DETACHED

// przygotowujemy zmienną warunkową
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);

// aktualizujemy strukturę danych dla wątku
wInfo.napis = "drugi";
wInfo.licznik = 0;
wInfo.cond = &cond;

// tworzymy kolejny wątek
if ((ret = pthread_create(
    &watek, 0, (void* (*)(void*))wypisz2, (void*)&wInfo )
    ) != 0
) {
    fprintf(stderr, "Error in pthread_create(): %s\n",
        strerror(ret));
    return 1;
}

// czekamy ...
while (wInfo.licznik < 5)
    pthread_cond_wait(wInfo.cond, wInfo.mut);
// pthread_cond_wait zwalnia tak na prawdę na chwile podany
// do niego mutex a po odebraniu pthread_cond_signal na
// dozorowanym przez niego warunku ponownie go zamyka
printf(" jesteśmy za warunkiem "); fflush(stdout);

sleep(7);

printf(" zabijamy wątek ");
// i go kończymy z zewnątrz
// w zależności od ustawień wątku anulowanie wątku dzieje
// się natychmiastowo albo w jednym z punktów anulowania
// zobacz funkcje to ustawiające w wypisz2()
pthread_cancel(watek);

// wypada też odebrać jego kod powrotu
pthread_join(watek, NULL);

```

```

    puts("a teraz watek już nie żyje ...");

    // usunięcie mutex'a
    pthread_mutex_destroy(wInfo.mut);

    // usunięcie zmiennej warunkowej
    pthread_cond_destroy(wInfo.cond);
}

/// Przykład C++
// przy kompilacji konieczne jest dodanie:
// -lboost_system -lboost_thread -lpthread
#include <boost/thread/mutex.hpp>
#include <boost/thread/thread.hpp>

#include <queue>
#include <string>
#include <iostream>

std::queue<std::string> kolejka;

class Watek1 {
    Watek1(const Watek1&);
public:
    int abc;
    boost::mutex mutex;
    // boost oferuje także mutex'a pozwalającego na tworzenie kolejnych
    // blokad przez wątek który utworzył pierwszą - boost::recursive_mutex

    Watek1() {}

    void operator()() {
        for (int i=0; i<8; i++) {
            if (mutex.try_lock()) {
                abc += 1;
                std::cout << "wątek, abc= " << abc << std::endl;
                mutex.unlock();
            } else {
                std::cout << "wątek nie dostałem mutex'a ..." << std::endl;
            }
            boost::this_thread::sleep(boost::posix_time::millisec(500));
        }

        for (int i=0; i<8; i++) {
            boost::this_thread::sleep(boost::posix_time::millisec(700));
            boost::mutex::scoped_lock scoped_lock(mutex);
            // powyższy lock (na mutexie tego wątku)
            // blokuje od teraz do końca bloku

            if (!kolejka.empty()) {
                std::cout << i << " ) odebrałem z kolejki \
                " << kolejka.front() << std::endl;
                kolejka.pop();
            } else {
                std::cout << i << " ) kolejka pusta" << std::endl;
            }
        }
    }
};

int main() {
    // przygotowanie danych wątku
    Watek1 wf1;
    wf1.abc=56;

    // uruchomienie wątku ...
    // boost::ref() potrzebne bo obiekty klasy Watek1 są niekopiowalne
    boost::thread watek1(boost::ref(wf1));

    boost::this_thread::sleep(boost::posix_time::millisec(1500));

    // manipulacja danymi wątku
    wf1.mutex.lock();
    wf1.abc=13;
    boost::this_thread::sleep(boost::posix_time::seconds(1));
}

```

```

wfl.mutex.unlock();

// obsługa kolejki
wfl.mutex.lock();
kolejka.push("Ala ma kota");
kolejka.push("Kot ma Ale");
wfl.mutex.unlock();

std::cout << "2 pierwsze elementy są w kolejce" << std::endl;
boost::this_thread::sleep(boost::posix_time::seconds(4));

wfl.mutex.lock();
kolejka.push("Abecadlo");
kolejka.push("z pieca spadło");
wfl.mutex.unlock();

std::cout << "2 kolejne elementy są w kolejce" << std::endl;

// czekamy na zakończenie wątku
watek1.join();
}

```

Przykład Python

```

import threading, time
# UWAGA:
# w standardowej implementacji CPython w danym
# czasie tylko jeden wątek może wykonywać
# kod pythonowy
#
# w efekcie stosowanie wątków ma sens tylko dla
# wątków czekających na coś, a nie ma sensu dla
# wątków mających wykonywać się równolegle
# (na różnych rdzeniach)
#
# w takim przypadku należy używać pełnego
# rozgałęziania procesu (moduł multiprocessing)

def fun(txt, st):
    # używamy lokalnej pamięci dla danego wątku
    daneWatku = threading.local()
    for daneWatku.i in range(4):
        print(txt, daneWatku.i)
        time.sleep(st)

# uruchomienie wątku
t = threading.Thread(target=fun, args=("AA",2,))
t.start();

# można także zdefiniować timer który
# uruchomi wątek po zadany czasie
tim = threading.Timer(3, fun, args=("BB",1.2,))
tim.start()

# dla wątków można stosować także mechanizmy
# synchronizacji takie jak dla multiprocessing
#
# ponadto wątki można także obsługiwać przez
# interfejs identyczny z multiprocessing
# poprzez multiprocessing.dummy

```

Komunikacja sieciowa

W systemach POSIXowych komunikacja sieciowa wymaga otwarcia tzw. gniazda (socket), przy otwarciu określone muszą zostać:

1. rodzina protokołów sieciowych która będzie używana (np. AF_INET dla IPv4 lub AF_INET6 dla IPv6)
2. typ komunikacji realizowanej z użyciem gniazda, do istotniejszych typów zaliczyć należy:

SOCK_STREAM

oznaczającego dwukierunkową niezawodną łączność zapewniającą zachowanie kolejności danych (w przypadku rodzin IPv4 i IPv6 będzie to oznaczało użycie TCP)

SOCK_DGRAM

oznaczającego bezpołączeniowe przesyłanie wiadomości o limitowanej długości maksymalnej (w przypadku rodzin IPv4 i IPv6 będzie to oznaczało użycie UDP)

SOCK_RAW

oznaczającego odbieranie i wysyłanie surowych pakietów warstwy sieciowej (pozwala m.in. na obsługę pakietów ICMP, swobodne manipulowanie adresami w pakietach IP, itd)

Opcjonalnie może zostać określony konkretny protokół jeżeli w ramach wskazanej rodziny i wskazanego typu występuje ich kilka.

Otwarcie gniazda skutkuje uzyskaniem deskryptora plikowego, na którym zależnie od parametrów przekazanych przy tworzeniu gniazda mogą być wykonywane operacje takie jak wysyłanie / odbieranie danych, ustawiany adres nasłuchiwania, zestawione połączenie, itd.

UDP

User Datagram Protocol (UDP) jest protokołem działającym powyżej protokołu IP umożliwiającym proste przesyłanie danych przez sieć. Pakiet UDP oprócz źródłowego i docelowego adresu IP, zawiera też źródłowy i docelowy numer portu. Numer portu określa usługę / proces działający w ramach hosta (określanego numerem IP) który wygenerował / ma otrzymać dany pakiet.

Po otwarciu gniazda obsługującego UDP możliwe jest:

- wysyłanie danych pod wskazany adres IP i numer portu
- oczekiwanie na odbiór danych i ich odbiór
- wskazanie adresu i numeru portu na którym oczekujemy na dane (jeżeli nie zostanie wykonane, co jest typowe w klientach UDP, będzie użyty losowy numer portu)

W UDP nie ma zestawionego połączenia zatem nie ma silnego rozróżnienia pomiędzy klientem (nawiązującym połączenie) a serwerem (oczekującym na połączenie / odbierającym je). Jedyna różnica polega na tym że: klient (typowo) nie wywołuje funkcji bind() służącej do określenia adresu wraz z numerem portu używanego do odbioru danych, a zamiast tego korzysta z automatycznie przydzielonego przez system operacyjny (losowego) numeru portu.

```
/// Przykład C++
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUF1_SIZE 256
#define BUF2_SIZE 64

const char* getAddrInfo(
    struct sockaddr_storage *srcAddr, socklen_t srcAddrLen, char* buf, int bufLen
);

int main(int argc, char *argv[]) {
    char buf1[BUF1_SIZE], buf2[BUF2_SIZE];
    int ret, pid;

    if (argc != 3 && argc != 4) {
        fprintf(stderr, "USAGE: %s dstHost dstPort [listenPort]\n", argv[0]);
        return EXIT_FAILURE;
    }

    // struktura zawierająca adres na który wysyłamy
    // użycie funkcji getaddrinfo pozwala na zapewnienie obsługi zarówno IPv4 jak i IPv6,
    // a także podawanie adresów w postaci numerycznej bądź przy użyciu nazw
    // alternatywnie można ręcznie przygotowywać struktury sockaddr_in / sockaddr_in6
    // z użyciem np. htons(), inet_aton(), ...
    struct addrinfo *dstAddrInfo;
    if((ret = getaddrinfo(argv[1], argv[2], NULL, &dstAddrInfo)) != 0) {
        fprintf(stderr, "Error in getaddrinfo(): %s\n", gai_strerror(ret));
        return EXIT_FAILURE;
    }

    // tutaj mogliśmy uzyskać kilka adresów,
    // ale optymistycznie zakładamy że pierwszy będzie OK

    // utworzenie gniazda sieciowego ... SOCK_DGRAM oznacza UDP
```

```

// (na gnieździe z SOCK_DGRAM standardowo nie będzie widać błędów ICMP
// związanych np. z niedostępnością docelowego hosta lub portu)
int sfd = socket(dstAddrInfo->ai_family, SOCK_DGRAM, 0);
if (sfd < 0) {
    perror("Error in socket()");
    return EXIT_FAILURE;
}

if (argc == 4) {
    // jeżeli podane opcjonalne argumentu to określamy na jakim porcie słuchamy
    if (dstAddrInfo->ai_family == AF_INET6) {
        struct sockaddr_in6 listenAddr;
        listenAddr.sin6_family=AF_INET6;
        listenAddr.sin6_port=htons(atoi(argv[3]));
        // host to network short (porządek bajtów), są też dla long i network to host
        listenAddr.sin6_addr=in6addr_any;
        // in6addr_any oznacza że nasłuchujemy na każdym adresie IP danego hosta
        // (zamiast tego można określić konkretny adres)

        if(bind(sfd, (struct sockaddr *) &listenAddr, sizeof(struct sockaddr_in6))) {
            perror("Error in bind()");
            return EXIT_FAILURE;
        }
    }
    else if (dstAddrInfo->ai_family == AF_INET) {
        struct sockaddr_in listenAddr;
        listenAddr.sin_family=AF_INET;
        listenAddr.sin_port=htons(atoi(argv[3]));
        // host to network short (porządek bajtów), są też dla long i network to host
        listenAddr.sin_addr.s_addr=INADDR_ANY;
        // INADDR_ANY oznacza że nasłuchujemy na każdym adresie IP danego hosta
        // (zamiast tego można określić konkretny adres)

        if(bind(sfd, (struct sockaddr *) &listenAddr, sizeof(struct sockaddr_in))) {
            perror("Error in bind()");
            return EXIT_FAILURE;
        }
    }
    // określenie tego wraz z uruchomieniem recvfrom() pozwala na oczekiwanie na
    // pakiety UDP na wskazanym adresie i porcie, gdyby w ramach obsługi odebranej
    // wiadomości odsyłana byłaby odpowiedź do nadawcy to byłoby to "serwer UDP"
}

// aby móc niezależnie wysyłać i odbierać rozgałęziamy proces
// alternatywnie można użyć select() do czekania na dane z sieci lub standardowego wejścia
switch (pid = fork()) {
    case -1: {
        // funkcja zwróciła -1 co oznacza błąd
        perror("Error in fork");
        return 1;
    }
    case 0: {
        // potomek odbiera odpowiedzi w nieskończonej pętli
        // zostanie zakończony sygnałem od rodzica

        while(1) {
            // struktura do której zapisany zostanie adres nadawcy
            struct sockaddr_storage srcAddr;
            socklen_t srcAddrLen = sizeof(struct sockaddr_storage);
            // informację tą można wykorzystać do odsyłania odpowiedzi do nadawcy

            // odbiór wiadomości ... blokuje do momentu otrzymania wiadomości
            ret = recvfrom(
                sfd, buf1, BUF1_SIZE, 0, (struct sockaddr *) &srcAddr, &srcAddrLen
            );

            // wypisanie wiadomości z informacją o nadawcy
            buf1[ret] = '\0';
            printf("odebrano %d bajtów od %s: %s\n",
                ret, getAddrInfo(&srcAddr, srcAddrLen, buf2, BUF2_SIZE), buf1
            );
        }
    }
    default: {
        // rodzic wysyła wiadomości
        while (fgets( buf1, BUF1_SIZE, stdin )) {
            ret = sendto(
                sfd, buf1, strlen(buf1), 0,

```

```

        dstAddrInfo->ai_addr, dstAddrInfo->ai_addrlen
    );
    printf("wysłano: %d bajtów: %s\n", ret, buf1);
}
// kończymy potomka gdy skończyliśmy nadawanie
kill(pid, SIGTERM);
wait(0);
}
}

// zamknięcie gniazda
puts("KONIEC");
freeaddrinfo(dstAddrInfo);
close(sfd);
}

// konwersja adresów IPv4 i IPv6 wraz z numerem portu na napis
const char* getAddrInfo(
    struct sockaddr_storage *srcAddr, socklen_t srcAddrLen,
    char* buf, int bufLen
) {
    if (srcAddrLen > sizeof(struct sockaddr_storage))
        return NULL;

    const char* addrStr = NULL;
    uint16_t port;
    if (srcAddr->ss_family == AF_INET) {
        struct sockaddr_in* srcAddrPtr = (struct sockaddr_in*) srcAddr;
        addrStr = inet_ntop(AF_INET, &(srcAddrPtr->sin_addr), buf, bufLen);
        port = ntohs(srcAddrPtr->sin_port);
    } else if (srcAddr->ss_family == AF_INET6) {
        struct sockaddr_in6* srcAddrPtr = (struct sockaddr_in6*) srcAddr;
        addrStr = inet_ntop(AF_INET6, &(srcAddrPtr->sin6_addr), buf, bufLen);
        port = ntohs(srcAddrPtr->sin6_port);
    }

    if (addrStr) {
        int len = strlen(buf);
        snprintf(buf+len, bufLen-len, ":%d", port);
    }

    return addrStr;
}

```

Przykład Python

```

import socket, sys, os, signal

if len(sys.argv) != 3 and len(sys.argv) != 4:
    print("USAGE: " + sys.argv[0] + " dstHost dstPort [listenPort]", file=sys.stderr)
    exit(1);

# struktura zawierająca adres na który wysyłamy
dstAddrInfo = socket.getaddrinfo(sys.argv[1], sys.argv[2])

# tutaj mogliśmy uzyskać kilka adresów,
# ale optymistycznie zakładamy że pierwszy będzie OK
dstAddrInfo = dstAddrInfo[0]

# utworzenie gniazda sieciowego ... SOCK_DGRAM oznacza UDP
# (na gnieździe z SOCK_DGRAM standardowo nie będzie widać błędów ICMP
# związanych np. z niedostępnością docelowego hosta lub portu ...)
sfd = socket.socket(dstAddrInfo[0], socket.SOCK_DGRAM)

if len(sys.argv) == 4:
    # jeżeli podane opcjonalne argumentu to określamy na jakim porcie słuchamy
    if dstAddrInfo[0] == socket.AF_INET6:
        sfd.bind([':::', int(sys.argv[3])])
        # :: oznacza dowolny adres IPv6
    elif dstAddrInfo[0] == socket.AF_INET:
        sfd.bind(['0.0.0.0', int(sys.argv[3])])
        # 0.0.0.0 oznacza dowolny adres
    # określenie tego wraz z uruchomieniem recvfrom() pozwala na oczekiwanie na
    # pakiety UDP na wskazanym adresie i porcie, gdyby w ramach obsługi odebranej
    # wiadomości odsyłana byłaby odpowiedź do nadawcy to byłoby to "serwer UDP"

while True:
    rfd, _, _ = select.select([sfd, sys.stdin], [], [])

```

```

if sfd in rdfd:
    # odbieramy i wypisujemy dane z UDP
    data, addr = sfd.recvfrom(4096)
    print("odebrano od", addr, ":", data.decode());
if sys.stdin in rdfd:
    # wczytujemy stdin
    data = sys.stdin.readline()
    if not data:
        # kończymy
        sfd.close()
        sys.exit()
    else:
        # wysyłamy dane z stdin
        sfd.sendto(data.encode(), dstAddrInfo[4])

# alternatywnie można użyć fork():
# pid = os.fork()
# if pid == 0:
#     # potomek odbiera odpowiedzi w nieskończonej pętli
#     # zostanie zakończony sygnałem od rodzica
#     while True:
#         data, addr = sfd.recvfrom(4096)
#         print("odebrano od", addr, ":", data.decode());
# else:
#     # rodzic wysyła wiadomości
#     while True:
#         data = sys.stdin.readline()
#         if not data:
#             break
#         else:
#             sfd.sendto(data.encode(), dstAddrInfo[4])
# os.kill(pid, signal.SIGTERM)
# sfd.close()

```

TCP - klient

Transmission Control Protocol (TCP) jest protokołem działającym powyżej protokołu IP gwarantującym niezawodność i kolejność dostarczania danych. Podobnie jak w przypadku UDP pakiety TCP oprócz numerów IP także zawierają docelowy i źródłowy numer portu, służący do identyfikacji procesu otrzymującego i nadającego.

Działanie TCP opiera się na zestawianiu (i kontrolowaniu) połączenia, w związku z tym klient po otwarciu gniazda musi nawiązać połączenie z wskazanym adresem IP i numerem portu, a dopiero potem może wysyłać lub odbierać dane (nie określając już adresów, gdyż odbywa się to w ramach nawiązanego połączenia) nawet poprzez zwykłe funkcje write() i read(). Połączenie należy zamknąć korzystając z funkcji close(), co informuje zarówno nasz system jak i serwer o zakończeniu połączenia.

```

/// Przykład C++
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUF_SIZE 256

int main(int argc, char *argv[]) {
    int ret, sfd;
    char buf[BUF_SIZE];

    if (argc != 3) {
        fprintf(stderr, "%s dstHost dstPort\n", argv[0]);
        return 1;
    }

    // struktura zawierająca adres na który wysyłamy
    // użycie funkcji getaddrinfo pozwala na zapewnienie obsługi zarówno IPv4 jak i IPv6,
    // a także podawanie adresów w postaci numerycznej bądź przy użyciu nazw
    // alternatywnie można ręcznie przygotowywać struktury sockaddr_in / sockaddr_in6
    // z użyciem np. htons(), inet_aton(), ...

```

```

struct addrinfo *dstAddrInfo;
if((ret = getaddrinfo(argv[1], argv[2], NULL, &dstAddrInfo)) != 0) {
    fprintf(stderr, "Error in getaddrinfo(): %s\n", gai_strerror(ret));
    return 1;
}

// mogliśmy uzyskać kilka adresów, więc próbujemy używać kolejnych do skutku
struct addrinfo *aiIter;
for (aiIter = dstAddrInfo; aiIter != NULL; aiIter = aiIter->ai_next) {
    // utworzenie gniazda sieciowego ... SOCK_STREAM oznacza TCP
    sfd = socket(aiIter->ai_family, SOCK_STREAM, 0);
    if (sfd < 0) {
        perror("Error in socket()");
        continue;
    }

    // łączymy z serwerem
    if (connect(sfd, aiIter->ai_addr, aiIter->ai_addrlen)) {
        perror("Error in connect()");
        close(sfd);
        sfd = -1;
        continue;
    }

    // udało się połączyć ... przerywamy sprawdzanie kolejnych adresów
    break;
}
freeaddrinfo(dstAddrInfo);
if (sfd < 0) {
    fprintf(stderr, "Can't connect\n");
    return 1;
}

const char* msg = "Ala ma Kota\n";
if (send(sfd, msg, strlen(msg), 0) < 0) {
    // takie wywołanie send() jest równoważne wywołaniu write()
    perror("Error in send()");
    return 1;
}

// ustalamy limit czasu oczekiwania na dane 13s
struct timeval timeout;
timeout.tv_sec = 13;
timeout.tv_usec = 0;

// tworzymy zestaw deskryptorów dla funkcji select()
// i umieszczamy w nim deskryptor gniazda sieciowego
fd_set fdSet;
FD_ZERO(&fdSet);
FD_SET(sfd, &fdSet);

// czekamy na dane w którymś z deskryptorów wchodzących
// w skład zestawu lub timeout
select(FD_SETSIZE, &fdSet, 0, 0, &timeout);

// select zmienia wartość fdSet i timeout ...
// będą one zawierały odpowiednio zbiór deskryptorów gotowych
// do czytania i czas jaki pozostał do "wytimeoutowania"

if (FD_ISSET (sfd, &fdSet)) {
    // jeżeli sfd jest w zbiorze gotowych do czytania to czytamy
    ret = read(sfd, buf, BUF_SIZE);
    if (ret < 0) {
        perror("Error in read()");
    } else if (ret == 0) {
        puts("connection close by peer");
    } else {
        buf[ret]='\0';
        printf("odebrano %d bajtów: %s\n", ret, buf);
    }
} else {
    puts("timeout");
}

// w zależności od wymagań implementowanego protokołu można kontynuować
// czekanie na kolejne dane z TCP i/lub jakiegoś innego wejścia (np. stdin),
// można wysłać kolejne dane (żądanie) do serwera, itd ...
//

```

```

// w przypadku oczekiwania na dalsze dane (ponownego wywołania select) należy
// pamiętać o ponownym ustawieniu fdSet i timeout

// zamkniecie gniazda
close(sfd);
}

### Przykład Python
import socket, select, sys

if len(sys.argv) != 3:
    print("USAGE: " + sys.argv[0] + " dstHost dstPort", file=sys.stderr)
    exit(1);

# struktura zawierająca adres na który wysyłamy
dstAddrInfo = socket.getaddrinfo(sys.argv[1], sys.argv[2], proto=socket.IPPROTO_TCP)

# mogliśmy uzyskać kilka adresów, więc próbujemy używać kolejnych do skutku
for aiIter in dstAddrInfo:
    try:
        print("try connect to:", aiIter[4])
        # utworzenie gniazda sieciowego ... SOCK_STREAM oznacza TCP
        sfd = socket.socket(aiIter[0], socket.SOCK_STREAM)
        # połączenie ze wskazanym adresem
        sfd.connect(aiIter[4])
    except:
        # jeżeli się nie udało ... zamykamy gniazdo
        if sfd:
            sfd.close()
        sfd = None
        # i próbujemy następny adres
        continue
    break;

if sfd == None:
    print("Can't connect", file=sys.stderr)
    exit(1);

# wysyłanie
sfd.sendall("Ala ma Kota\n".encode())

# czekanie na dane i odbiór danych
while True:
    rfd, _, _ = select.select([sfd], [], [], 13.0)
    if sfd in rfd:
        d = sfd.recv(4096)
        d = d.decode()
        print(d, end="")

        # odbiór pustego pakietu lub pakietu zawierającego
        # jedynie pustą linię kończy działanie
        if d == "" or d == "\n" or d == "\r\n":
            break
    else:
        # timeout kończy działanie
        break

# zamykanie połączenia
sfd.shutdown(socket.SHUT_RDWR)
sfd.close()

```

TCP - serwer

Ze względu na nawiązywanie połączenia w przypadku TCP serwer wygląda i działa odmiennie od klienta. Po otwarciu gniazda serwer musi:

1. określić adresu i numeru portu na którym oczekuje na przychodzące połączenia (funkcja bind)
2. określić że gniazdo to używane jest do nasłuchiwanie połączeń przychodzących (funkcja listen)
3. rozpocząć oczekiwanie na odbiór połączenia (funkcja accept)

Po nawiązaniu połączenia TCP (realizowanym na poziomie systemu operacyjnego) funkcja accept() zwraca nowe gniazdo związane z nawiązanym połączeniem TCP. Ma ono ustawiony adresem IP i portem używany przez klienta oraz inny (niż używany do nasłuchiwanie) numer portu po stronie serwera.

/// Przykład C++

```

/// Przykład C++
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <sys/wait.h>

#define BUF_SIZE 256
#define QUERY_SIZE 3
#define MAX_CHILD 5

int childNum = 0;
void onChildEnd(int sig) {
    puts("odebrano sygnał o śmierci potomka");
    --childNum;
    waitpid(-1, NULL, WNOHANG);
}

int main(int argc, char *argv[]) {
    int res;
    char buf[BUF_SIZE];

    if (argc != 2) {
        fprintf(stderr, "USAGE: %s listenPort\n", argv[0]);
        return EXIT_FAILURE;
    }

    // obsługa sygnału o zakończeniu potomka
    signal(SIGCHLD, &onChildEnd);

    // utworzenie gniazda sieciowego ... SOCK_STREAM oznacza TCP
    int sfd = socket(PF_INET6, SOCK_STREAM, 0);
    if (sfd < 0) {
        perror("Error in socket()");
        return EXIT_FAILURE;
    }

    // ustawienie opcji gniazda ... IPV6_V6ONLY=0 umożliwia korzystanie z tego
    // samego socketu dla IPv4 i IPv6 (rozszerzenie Linux-owe - na niektórych
    // systemach nie jest możliwe dualne słuchanie przy pomocy socketu IPv6)
    int opt = 0;
    if (setsockopt(sfd, IPPROTO_IPV6, IPV6_V6ONLY, &opt, sizeof(opt)) < 0) {
        perror("setsockopt(IPV6_V6ONLY=0)");
        return EXIT_FAILURE;
    }

    // utworzenie struktury opisującej adres
    struct sockaddr_in6 server;
    server.sin6_family = AF_INET6;
    server.sin6_port = htons(atoi(argv[1]));
    // host to network short (porządek bajtów), są też dla long i network to host
    server.sin6_addr = in6addr_any;
    // in6addr_any oznacza że nasłuchujemy na każdym adresie IP danego hosta
    // (zamiast tego można określić konkretny adres)

    // przypisanie adresu ...
    if (bind(sfd, (struct sockaddr *) &server, sizeof(struct sockaddr_in6)) < 0) {
        perror("Error in bind()");
        return EXIT_FAILURE;
    }

    // określenie gniazda jako używanego do nasłuchiwania połączeń przychodzących
    if (listen(sfd, QUERY_SIZE) < 0) {
        perror("Error in listen()");
        return EXIT_FAILURE;
    }

    while(1) {
        // odebranie połączenia
        struct sockaddr_in6 from;
        socklen_t fromLen = sizeof(struct sockaddr_in6);
        int sfd2 = accept(sfd, (struct sockaddr *) &from, &fromLen);

```

```

// weryfikacja ilości potomków
if (childNum >= MAX_CHILD) {
    printf("za dużo potomków\n");
    // można tutaj także wysłać informacje o błędzie serwera do klienta
    close(sfd2);
    continue;
}

// aby móc obsługiwać wiele połączeń rozgałęziamy proces
int pid = fork();
if (pid == 0) {
    // proces potomny ... zajmuje się tylko nowym połączeniem,
    // a nie nasłuchaniem więc zamykamy sfd
    close(sfd);

    // konwersja adresu na postać napisową
    char fromStr[64];
    snprintf(fromStr, 64, "%s:%d",
        inet_ntop(AF_INET6, &(from.sin6_addr), buf, BUF_SIZE),
        ntohs(from.sin6_port)
    );

    // obsługa połączenia
    printf("połączenie od: %s\n", fromStr);
    FILE * net;
    net=fdopen(sfd2, "r+");
    // należałoby tutaj używać czekania z timeout'em ...
    // inaczej usługa jest podatna na ataki DoS
    while(fgets(buf, BUF_SIZE, net)) {
        printf("odebrano od %s: %s\n", fromStr, buf);
        fputs(buf, net);
    }
    fclose(net);
    printf("koniec połączenia od: %s\n", fromStr);

    // zakończenie połączenia i potomka
    close(sfd2);
    return 0;
}

if (pid == -1)
    perror("Error in fork()");

++childNum;
close(sfd2);
}

// zamknięcie gniazda
close(sfd);
}

```

Przykład Python

```

import socket, select, signal, sys, os

MAX_CHILD = 5
QUERY_SIZE = 3
TIMEOUT = 13
BUF_SIZE = 4096

if len(sys.argv) != 2:
    print("USAGE: " + sys.argv[0] + " listenPort", file=sys.stderr)
    exit(1);

# obsługa sygnału o zakończeniu potomka
childNum = 0
def onChildEnd(s, f):
    print("odebrano sygnał o śmierci potomka")
    global childNum
    childNum -= 1
    os.waitpid(-1, os.WNOHANG);
signal.signal(signal.SIGCHLD, onChildEnd)

# utworzenie gniazd sieciowych ... SOCK_STREAM oznacza TCP
sfd_v4 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sfd_v6 = socket.socket(socket.AF_INET6, socket.SOCK_STREAM)

```

```

# ustawienie opcji gniazda ... IPV6_V6ONLY=1 wyłącza korzystanie
# z tego samego socketu dla IPv4 i IPv6
sfd_v6.setsockopt(socket.IPPROTO_IPV6, socket.IPV6_V6ONLY, 1)

# przypisanie adresów ...
# '0.0.0.0' oznacza dowolny adres IPv4 (czyli to samo co INADDR_ANY)
# ':::' oznacza dowolny adres IPv6 (czyli to samo co in6addr_any)
sfd_v4.bind(('0.0.0.0', int(sys.argv[1])))
sfd_v6.bind(':::', int(sys.argv[1]))

# określenie gniazd jako używanych do odbioru połączeń przychodzących
# (długość kolejki połączeń ustawiona na wartość QUERY_SIZE)
sfd_v4.listen(QUERY_SIZE)
sfd_v6.listen(QUERY_SIZE)

# czekanie na połączenia z użyciem select() w nieskończonej pętli
while True:
    sfd, _, _ = select.select([sfd_v4, sfd_v6], [], [])
    for fd in sfd:
        # odebranie połączenia
        sfd_c, sAddr = fd.accept()

        # weryfikacja ilości potomków
        if childNum >= MAX_CHILD:
            print("za dużo potomków - odrzucam połączenie od:", sAddr);
            sfd_c.send("Internal Server Error\r\n".encode())
            sfd_c.close()
            break

        # aby móc obsługiwać wiele połączeń rozgałęziamy proces
        pid = os.fork()
        if pid > 0:
            # rodzic - zwiększamy licznik potomków
            childNum += 1
        else:
            # potomek - obsługa danego połączenia
            print("połączenie od:", sAddr)
            while True:
                # czekanie na dane z timeout'em
                # aby zabezpieczyć się przed atakiem DoS
                rd, _, _ = select.select([sfd_c], [], [], TIMEOUT)
                if sfd_c in rd:
                    data = sfd_c.recv(BUF_SIZE)
                    if data:
                        print("odebrano od", sAddr, ":", data.decode());
                        sfd_c.send(data)
                    else:
                        print("koniec połączenia od:", sAddr)
                        break
                else:
                    print("timeout połączenia od:", sAddr)
                    break
            # zamykanie połączenia
            sfd_c.shutdown(socket.SHUT_RDWR)
            sfd_c.close()
            sys.exit()

```

Inne funkcje systemowe

Alokacja pamięci

Zadeklarowanie zmiennej w kodzie programu (prawie zawsze) wiąże się z przydzieleniem jej jakiegoś obszaru pamięci. Kompilator typowo zmienne lokalne umieszcza w obszarze pamięci nazywanym stosem (znajdują się tam też m.in. argumenty funkcji, stany rejestrów procesora w momencie wywołania funkcji i adresy powrotu funkcji).

Stos posiada ograniczony rozmiar, którego przekroczenie (np. na skutek zbyt długiego ciągu wywołań funkcji lub zdefiniowania zbyt dużych zmiennych lokalnych) kończy się błędem przepełnienia stosu. Rozmiar stosu może jednak być zmieniony zarówno przed uruchomieniem procesu (przy pomocy `ulimit -s`), lub w trakcie jego pracy (przy pomocy `setrlimit(RLIMIT_STACK, ...)`), pod warunkiem że mieści się w ramach ogólnosystemowego limitu. Zwiększenie rozmiaru stosu wpływa na rozmiar pamięci wirtualnej przydzielonej procesowi, jednak (samo w sobie) nie wpływa na rzeczywiste zużycie pamięci przez proces.

Często nawet samo zadeklarowanie dużej tablicy nie wpływa na rzeczywistą zajętość pamięci, wpływ

ma dopiero używanie takiej tablicy:

```
/// Przykład C++
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/resource.h>

char buf[64], *bufS;

int fun() {
    int i, j, x[10000000];

    for (i=0; i<17; i+=2) {
        for (j=0; j<20000; ++j) x[i*10000+j] = j;
        sleep(1);
        system(buf);
    }
}

int main() {
    snprintf(buf, 64, "ps -o 'vsize,size,rss,cmd' -p %d", getpid());
    system(buf);

    puts("zwiększam rozmiar stosu");
    struct rlimit rl;
    rl.rlim_cur = 2000000000;
    rl.rlim_max = rl.rlim_cur;
    if(setrlimit(RLIMIT_STACK, &rl))
        perror("Error in setrlimit()");

    system(buf);
    bufS = buf + strlen(buf);
    snprintf(bufS, 64 - (bufS-buf), " --no-headers");

    fun();
    *bufS = '\0'; system(buf); *bufS = ' ';
    fun();
}
```

Mimo możliwości zwiększenia rozmiaru stosu typowo większe bufor czy też obiekty alokowane są poza stosem. W C++ stosowane są dwie techniki takie alokacji: operator `new` i funkcje z rodziny `malloc`. Alokacja pamięci poza stosem powoduje konieczność jawnego zwalniania pamięci (nie jest ona zwalniana wraz z zniknięciem zmiennej) przy pomocy odpowiednio operatora `delete` lub funkcji `free`.

```
/// Przykład C++
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <string>
#include <inttypes.h>

struct Abc {
    std::string a;
    int b;
private:
    short c;
public:
    Abc() {
        puts("konstruktor ABC");
        c = 123;
    }
    ~Abc() {
        puts("destruktor ABC");
    }
    void print() {
        printf("a: %s, b: %d, c:%d\n", a.c_str(), b, c);
    }
};

struct Xxx {
    Abc abc1;
```

```

    Abc abc2;
    int tab[9];
    int xx1;
    int xx2;
};

int main(int argc, char *argv[]) {
    int a[5], aa;

    // alokacja pamięci
    int *bb = (int*)malloc(sizeof(int));
    int *cc = new int;
    int *c = new int[20]; // tablica alokowana dynamicznie w stylu C++
    Abc *dd = new Abc;

    printf("na stosie:  %p %p %p %p\n", &argv, &argc, &aa, a);
    printf("poza stosem: %p %p %p %p\n", bb, cc, c, dd);

    // zwalnianie pamięci
    free(bb);
    delete cc;
    delete c;
    delete dd;

    // alokujemy większy obszar pamięci przy pomocy malloc'a
    char* x = (char*)malloc(1024);
    // jest też realloc() umożliwiający zmianę rozmiaru zaalokowanego obszaru

    // zerujemy zaalokowaną pamięć ... na ogół nie jest to potrzebne,
    // ale w tym przykładzie może się przydać
    memset(x, 0, 1024);
    // często przydatna jest też funkcja memcpy() kopiująca wskazany
    // fragment pamięci w inne miejsce

    // pamięć zaalokowaną z użyciem malloc'a możemy dzięki
    // artemetyce wskaźnikowej oraz rzutowaniu typów wskaźnikowych
    // interpretować w dowolny sposób ...
    Abc *d1, *d2;
    int *b1, *b2;
    d1 = (Abc*)x;
    d2 = (Abc*)(x) + 1;
    b1 = (int*)(x + 2 * sizeof(Abc));
    b2 = b1 + 10;

    printf("rozmiar Abc: %ld == %ld == %ld\n",
           sizeof(Abc), (uint64_t)d2-(uint64_t)d1, (uint64_t)b1-(uint64_t)d2
    );

    d1->b = 15;
    d1->print();
    // jak widać powstał obiekt typu Abc, ale nie wykonał się konstruktor ...
    // ani konstruktory składowych (jak std::string), więc wywołanie:
    // d1->a = "aa bb cc";
    // może się nie udać (zakończyć się np. Segmentation fault)
    //
    // aby uniknąć związanych z tym problemów możemy skorzystać z wariantu
    // operatora new który utworzy obiekt w zaalokowanej pamięci:
    new(d2) Abc;
    d2->a = "aa bb cc";
    d2->print();

    for (aa=0; aa<10; ++aa)
        b1[aa] = aa + 16;
    *b2 = 136917;

    // możemy też na taki obszar pamięci (nawet już zapełniony)
    // patrzeć np. jak na strukturę
    Xxx* z = (Xxx*)x;
    printf("%d == %d  %d == %d  %d == %d\n",
           z->tab[2], b1[2], z->xx1, b1[9], z->xx2, *b2
    );

    free(x); // takie zwolnienie nie wywoła destruktorów dla d1 i d2
}

```

Czas

Bieżący czas w systemach komputerowych może być reprezentowany na różne sposoby. W przypadku systemów POSIXowych jest to ilość sekund od początku epoki czyli 1970-01-01 00:00:00 UTC. Na podstawie tej wartości może zostać obliczona przez funkcje biblioteczne aktualna data i godzina w ustawionej (poprzez zmienną środowiskową TZ) strefie czasowej.

```
/// Przykład C++
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

size_t time2str(
    const time_t* time, const char* timeZone, const char *format,
    char *buf, size_t bufLen, struct tm *tmPtr
);

int main() {
    time_t czas1 = time(0);
    printf("Od początku epoki upłynęło %d sekund\n", czas1);
    // początek epoki to: 1970-01-01 00:00:00 UTC
    // wynik jest niezależny od lokalnej strefy czasowej
    // (zawsze wyrażany jest w UTC)

    // jeżeli potrzebna jest większa precyzja to można użyć np.:
    struct timespec czas2;
    clock_gettime(CLOCK_REALTIME, &czas2);
    printf("Od początku epoki upłynęło %d sekund i %d ns\n",
        czas2.tv_sec, czas2.tv_nsec);
    // na niektórych platformach (np. Linux) w clock_gettime()
    // wspierane są także inne zegary niż czasu rzeczywistego
    // np. podające czas monotoniczny lub od uruchomienia systemu

    char buf[128];

    // korzystamy z własnej funkcji tworzącej napis w oparciu o
    // format zmiennej czasowej ... co prawda do UTC jest gmtime(),
    // ale w połączeniu z strftime potrafi dawać złe wyniki
    // (np. dla "%s") ponadto funkcja taka pozwala także na
    // obsługę innych stref czasowych
    time2str(
        &(czas2.tv_sec), "UTC", "%Y-%m-%d %H:%M:%S %Z (%s)",
        buf, sizeof(buf), NULL
    );
    printf("Mamy teraz: %s\n", buf);

    time2str(
        &(czas2.tv_sec), NULL, "%Y-%m-%d %H:%M:%S %Z (%s)",
        buf, sizeof(buf), NULL
    );
    printf("Mamy teraz: %s\n", buf);
}

size_t time2str(
    const time_t* time, const char* timeZone, const char *format,
    char *buf, size_t bufLen, struct tm *tmPtr
) {
    // jeżeli podano tmPtr != NULL użyj podanej struktury,
    // w przeciwnym razie użyj lokalnej
    struct tm tmTmp;
    if (!tmPtr) {
        tmPtr = &tmTmp;
    }

    // jeżeli podano strefę czasową to
    // zapamiętaj oryginalną strefę i ustaw nową
    char oldTZ[128];
    if (timeZone) {
        strncpy(oldTZ, getenv("TZ"), 128);
        setenv("TZ", timeZone, 1);
        tzset();
    }

    // pobierz rozłożony czas w ustawionej strefie czasowej
    localtime_r(time, tmPtr);
}
```

```

// zapisz sformatowany napis do bufora
bufLen = strftime(buf, bufLen, format, tmPtr);

// przywróć oryginalną strefę czasową
if (timeZone) {
    setenv("TZ", oldTZ, 1);
    tzset();
}

return bufLen;
}

```

Przykład Python

```

import time
import os

def time2str(t, tz=None, fmt="%Y-%m-%d %H:%M:%S %Z (%s)"):
    if tz:
        oldTZ = os.environ['TZ']
        os.environ['TZ'] = tz
        time.tzset()

    tm = time.localtime(t)
    s = time.strftime(fmt, tm)

    if tz:
        os.environ['TZ'] = oldTZ
        time.tzset()

    return [s, tz]

czas = time.time()
print("Od początku epoki upłynęło " + str(czas) + " sekund")
# początek epoki to: 1970-01-01 00:00:00 UTC
# wynik jest niezależny od lokalnej strefy czasowej
# (zawsze wyrażany jest w UTC)

tt = time2str(czas, tz="UTC")
print("Mamy teraz: " + tt[0]);

tt = time2str(czas)
print("Mamy teraz: " + tt[0]);

from datetime import datetime, timezone

dt = datetime.fromtimestamp(czas, tz=timezone.utc)
print("Mamy teraz: " + dt.strftime("%Y-%m-%d %H:%M:%S %Z (%s)"))

# w celu pobrania aktualnego czasu zamiast fromtimestamp()
# można skorzystać z metody now()

```

Timery

Jeżeli program ma wykonywać jakąś czynność co określony czas może korzystać z funkcji `sleep` lub podobnych celem przeczekania podanego czasu lub skorzystać z timerów celem otrzymania sygnału po zadanym czasie.

```

/// Przykład C++
// przy kompilacji konieczne jest dodanie:
// -lrt -lpthread

#include <time.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

void onTimer(int sig) {
    puts("timer !!!");

    // drzemka 0.35s
    // do spania można wykorzystać usleep(), sleep() lub nanosleep()
    // usleep i sleep działają bardzo podobnie

```

```

// (z tym że pierwszy przyjmuje czas w sekundach, a drugi w mikrosekundach)
usleep(350000);
}

int main() {
    // obsługujemy sygnał SIGALRM
    signal(SIGALRM, &onTimer);

    // najprostrzy timer ...
    alarm(4); pause();

    // timery POSIX (więcej man timer_settime)

    // 1) tworzymy timer
    timer_t timer;
    timer_create(CLOCK_REALTIME, NULL, &timer);
    // podanie wskaźnika na strukturę sigevent zamiast NULL pozwala na określenie
    // innego sygnału jaki ma wygenerować timer lub obsługę tak jak wątku
    // (więcej w `man 7 sigevent`)

    // 2) konfiguryujemy timer
    struct itimerspec tSpec;
    // pierwsze wykonanie timera po 100ms
    tSpec.it_value.tv_sec=0;
    tSpec.it_value.tv_nsec=100000000;
    // kolejne co 1s
    // (jeżeli it_interval będzie wyzerowane to timer wykona się tylko raz)
    tSpec.it_interval.tv_sec=1;
    tSpec.it_interval.tv_nsec=0;

    // 3) uruchamiamy timer
    // drugim argumentem są flagi - flaga TIMER_ABSTIME pozwala na
    // ustawienie timera na czas absolutny (a nie okres czasu)
    timer_settime(timer, 0, &tSpec, NULL);
    char i=0;
    do {
        struct itimerspec tRest;
        timer_gettime(timer, &tRest);
        printf("do timera pozostało: %dns\n", tRest.it_value.tv_nsec);
        pause(); // czekamy na timer
        printf("aktualna liczba utraconych tyknięć timera wynosi %d\n",
            timer_getoverrun(timer));
    } while(i++<3);
    // powyższa metoda zapewnia rozpoczynanie w równych odstępach czasu ...
    // w przypadku zastosowania sleep() należałoby obliczać czas przez który
    // wykonywał się kod i odejmować go od czasu który podajemy do sleep()

    // drzemka

    // nanosleep pozwala na określenie sekund i milisekund oraz zwraca czas
    // różnicę pomiędzy planowanym czasem spania a rzeczywistym czasem
    // od wejścia do powrotu z funkcji nanosleep() - uwzględniając np. czas
    // obsługi sygnału który przerwał działanie funkcji nanosleep()
    struct timespec drzemka, pobudka;
    drzemka.tv_sec=2;
    drzemka.tv_nsec=100000000;
    // chcemy spać 2.1s ale obudzi nas timer
    // to ile zesmy niedospali zostanie zapisane w pobudka
    if (nanosleep(&drzemka, &pobudka)<0 && errno == EINTR)
        printf("niedospaliliśmy: %ds i %dns\n", pobudka.tv_sec, pobudka.tv_nsec);

    // usunięcie timera
    timer_delete(timer);
}

```

Biblioteki

Celem uniknięcia konieczności wielokrotnego tworzenia funkcji realizujących te same zadania w programowaniu powszechnie wykorzystuje się różnego rodzaju biblioteki programistyczne. Biblioteka jest zbiorem funkcji i powiązanych z nimi typów danych oraz danych. Praktycznie każdy z języków oferuje zbiór podstawowych funkcji zgromadzony w tzw. bibliotece standardowej, oprócz biblioteki standardowej danego języka można korzystać także z zewnętrznych (w tym własnych) bibliotek.

Biblioteka posiada określony interfejs programistyczny (API), definiowany na poziomie kodu źródłowego, który w przypadku programów w C/C++ jest zazwyczaj opisany w plikach nagłówkowych związanych z daną biblioteką (najczęściej z rozszerzeniem .h). Zmiany API biblioteki wraz z kolejnymi jej wersjami mogą (ale nie muszą) - zależeć od zachowywania tzw. kompatybilności wstecznej przez twórców biblioteki) wymagać zmian w programach z niej korzystających. Interfejs biblioteki typowo dostępny jest natywnie tylko dla języka w którym została stworzona. Dla innych języków wymaga on obudowania w taki sposób aby był zrozumiały dla kompilatora / interpretera danego języka, zadanie to realizują tzw. wrappery.

Oprócz API biblioteki posiadają także określony interfejs binarny (ABI), który odpowiada za możliwość wywoływania odpowiednich procedur z biblioteki przez skompilowany i zlinkowany program. Typowo ABI jest mniej stabilne od API i zmiany wymagające skompilowania i zlinkowania programu z nową wersją biblioteki zachodzą częściej niż zmiany API nie zachowujące kompatybilności wstecznej.

Kod biblioteki może być włączony na stałe w kod programu (biblioteka statyczna) - zapewnia to brak problemów z zgodnością ABI biblioteki pomiędzy systemem na którym program został skompilowany i zlinkowany, a systemem na którym jest uruchamiany. Skutkuje to jednak wzrostem rozmiaru pliku wykonywalnego oraz zapotrzebowaniem na pamięć uruchomionego programu. Alternatywą są biblioteki ładowane dynamicznie, które dostarczane są jako osobne (w stosunku co do binarki programu) pliki i ładowane do pamięci w momencie startu programu lub podczas jego działania. Kilka różnych programów używających tej samej biblioteki dynamicznej będzie używało jednej jej kopii umieszczonej w pamięci RAM.

API, skrypty, ...

API pythonowe biblioteki C++

Podstawowy interpreter Pythona stworzony jest w C, podobnie wiele z popularnych modułów (bibliotek) pythonowych jest tworzony w C lub C++. Python potrafi korzystać z zwykłych bibliotek C (poprzez moduł ctypes), jednak aby z biblioteki korzystało się wygodnie powinna ona wystawiać bardziej natywny interfejs pythonowy w postaci modułu. Na utworzenie takiego interfejsu z poziomu C pozwala biblioteka programistyczna dostarczana wraz z Pythonem, w przypadku C++ można także korzystać z obudowującego wywołania tej biblioteki boost::python.

```
/// Przykład C++
/* Plik: py_api.cpp      kompilacja:
g++ --std=c++11 -shared -fPIC -I/usr/include/python3.7m/ py_api.cpp \
-o MyPyAPI.so -lpython3.7m -lboost_python37
uwaga: - kolejność argumentów może być istotna
        - plik wynikowy powinien mieć taką samą nazwę jak moduł
        zadeklarowany przy pomocy BOOST_PYTHON_MODULE()
*/

#include <iostream>
#include <string>
#include <list>

std::string f1(int a) {
    for (int i=0; i<a; ++i)
        std::cout << "Ala ma kota\n";
    return "Kot ma Alę";
}

struct K1 {
    int a;
    static K1* obj;
    int f1(int b) { return a + b; }
    static K1* get() { return obj; }
};

void f2(K1& k, int n) {
    std::cout << "run f2(): " << k.f1(2*n) << "\n";
    k.a += 1;
}

K1* K1::obj = NULL;

#include <boost/python.hpp>

BOOST_PYTHON_MODULE(MyPyAPI) {
    boost::python::def("f1", f1);

    boost::python::class_<K1>("Klasa")
        .def_readwrite("a", &K1::a)
        .def("f1", &K1::f1)
}
```

```

// w pythonie w odróżnieniu od C++ referencja do klasy jest jawnym
// argumentem metod nie statycznych zatem f2 które przyjmuje jako
// pierwszy argument referencję do obiektu klasy K1 możemy użyć
// jako metody klasy pythonowej
.def("f2", f2)

// typ z funkcji do pythona zwracana jest wartość zmiennej (a nie
// referencja do zmiennej C++) jednak dla funkcji zwracających
// wskaźnik lub referencję na ogół chcemy aby zwracana była właśnie
// referencja do istniejącej zmiennej C++
.def("get", &K1::get,
    boost::python::return_value_policy<
        boost::python::reference_existing_object
    >()
)
;

// f2 możemy też użyć jako niezależnej funkcji
boost::python::def("f22", f2);

// podobnie jak f1
boost::python::def("f11", &K1::f1);
}

```

Przykład Python

```

# skrypt wykorzystujący moduł
# MyPyAPI stworzony w C++

# dodajemy bieżący katalog do ścieżki w której python szuka bibliotek
import sys
sys.path.append('./')

# importujemy naszą bibliotekę (z pliku MyPyAPI.so)
import MyPyAPI

# uruchomienie funkcji f1 i odebranie wyniku
ret = MyPyAPI.f1(2)
print(ret)

# utworzenie obiektu klasy K1 (Klasa)
kk = MyPyAPI.Klasa()

# oraz jego używanie ...
kk.a = 3
print(kk.f1(2))

kk.f2(1)
MyPyAPI.f22(kk, 1)
print("kk.a =", kk.a)

print(MyPyAPI.f11(kk, 1))

```

Przykład Python

```

# skrypt używający biblioteki C
# bez API pythonowego

# skrypt powoduje włączenie NumLock
# przy pracy w trybie graficznym
# zgodnym z X11

from ctypes import *
X11 = cdll.LoadLibrary("libX11.so.6")

X11.XOpenDisplay.restype = c_void_p
display = X11.XOpenDisplay(None);
X11.XkbLockModifiers(
    c_void_p(display), c_uint(0x0100),
    c_uint(16), c_uint(16)
)
X11.XCloseDisplay(c_void_p(display))

```

interfejs skryptowy programu C++

W wielu przypadkach zachodzi potrzeba połączenia zalet programów kompilowanych (szybkość działania) i interpretowanych (elastyczność, łatwość modyfikacji). Jednym z rozwiązań to umożliwiających jest utworzenie w ramach kodu kompilowanego C++ interfejsu modułu

pythonowego, przy jednoczesnej możliwości zapewnienia wywoływania skryptów pythonowych z poziomu kodu C++ korzystających do komunikacji z tym kodem z tego interfejsu.

/// Przykład C++

```
/* PLIK: py_run.cpp      kompilacja:
   g++ -I/usr/include/python3.7m/ py_run.cpp -lpython3.7m -lboost_python37
   uwaga: - kolejność argumentów może być istotna
           - plik używa "py_api.cpp" z poprzedniego przykładu
           - plik wywołuje skrypt "py_run.script.py" z bieżącego katalogu
*/

// korzystamy z wcześniej przygotowanego (w "API pythonowe biblioteki C++") interfejsu pythonowego
#include "py_api.cpp"

int main(int, char **) {
    K1 *o1 = new K1();
    o1->a = 1;

    K1 *o2 = new K1();
    o2->a = 2;

    K1::obj = o1;

    std::cout << "o1->a = " << o1->a << "    o2->a = " << o2->a << "\n";

    // initialise python
    Py_Initialize();

    try {
        // initialise and import MyPyAPI module
        PyObject* module = PyInit_MyPyAPI();
        PyDict_SetItemString(PyImport_GetModuleDict(), "MyPyAPI", module);
        Py_DECREF(module);
        PyRun_SimpleString("import MyPyAPI" );
        // poprzez PyRun_SimpleString można też uruchamiać inne fragmenty kodu pythonowego

        // prepare to run scripts
        boost::python::object main = boost::python::import("__main__");
        boost::python::object global(main.attr("__dict__"));

        // export object to python
        global["o1"] = boost::python::ptr(o1);

        // run file
        boost::python::object result = boost::python::exec_file(
            "./py_run.script.py", global, global
        );

        // import object from python
        boost::python::object script = global["script1"];

        // run python function with args from C++
        if(!script.is_none()) {
            // run scripts
            std::cout << "RUN\n";
            std::cout << "return = " << boost::python::extract<int>(
                script(boost::python::ptr(o2))
            ) << "\n";
        }
    } catch(boost::python::error_already_set &) {
        PyErr_Print();
        exit(-1);
    }

    std::cout << "o1->a = " << o1->a << "    o2->a = " << o2->a << "\n";

    delete o1;
    delete o2;

    return 0;
}
```

Przykład Python

```
# PLIK: py_run.script.py
# uruchamiany przez kod C++ z py_api.cpp
```

```

print("początek pliku .py")

ck0 = MyPyAPI.Klasa.get()
print("ck0:", ck0.f1(0))
print("ck1:", ck1.f1(0))

ck0.a=4
print("ck0:", ck0.f1(0))
print("ck1:", ck1.f1(0))

def script1(arg):
    print("run script1")

    print("ck0:", ck0.f1(0))
    print("ck1:", ck1.f1(0))
    print("arg:", arg.f1(0))

    arg.a=13
    print("ck0:", ck0.f1(0))
    print("ck1:", ck1.f1(0))
    print("arg:", arg.f1(0))

    print("end script1")

    return arg.a + ck0.a;

print("koniec pliku .py")

```

Bazy danych

Standardowym językiem używanym do komunikacji z systemami bazodanowymi jest SQL. Pomimo jego standaryzacji istnieją różnice w składni zapytań dla poszczególnych silników bazodanowych (takich jak: MariaDB, PostgreSQL, SQLite, ...).

Typowo komunikacja z bazą danych odbywa się za pośrednictwem biblioteki odpowiedzialnej za nawiązanie połączenia z serwerem i przekazywanie do niego zapytań SQL. Wymaga to działania osobnego procesu (często nawet na innej maszynie) obsługującego silnik bazodanowy, co jest pożądanym rozwiązaniem dla baz danych z których równocześnie może korzystać wielu klientów. Typowym przykładem może być komunikacja skryptów jakiegoś serwisu internetowego z bazą danych. Jednak takie podejście nie jest wygodne w rozwiązaniach nie wymagających współdzielenia bazy danych. Do zastosowań takich można użyć biblioteki SQLite, która pozwala na łatwe stosowanie bazy SQLowej do wewnętrznych potrzeb aplikacji, bez konieczności uruchamiania osobnego systemu bazodanowego.

```

/// Przykład C++
// przy kompilacji konieczne jest dodanie:
// -lsqlite3

#include <sqlite3.h>
#include <stdio.h>

/* bazę można utworzyć zarówno w poniższym kodzie poleceniami SQL,
 * jak też z poziomu powłoki:
cat << EOF | sqlite3 example.db
CREATE TABLE posts (pid INT PRIMARY KEY, uid INT, text TEXT);
INSERT INTO posts VALUES (1, 21, 'abc ..');
INSERT INTO posts VALUES (3, 2671, 'test');
EOF
*/

int main() {
    sqlite3 *db;
    if ( sqlite3_open("example.db", &db) ) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return 1;
    }

    sqlite3_stmt *stmt;
    int ret = sqlite3_prepare_v2(db,
        "SELECT uid, text FROM posts",
        -1, &stmt, NULL
    );
    while ( ( ret = sqlite3_step(stmt) ) == SQLITE_ROW ) {

```

```

        int uid = sqlite3_column_int(stmt, 0);
        const char* text = (const char*) sqlite3_column_text(stmt, 1);
        printf("uid = %d, text: %s\n", uid, text);
    }
}

```

Przykład Python

```

import sqlite3
import os.path

if os.path.isfile('example.db'):
    create = False
else:
    create = True
conn = sqlite3.connect('example.db')
c = conn.cursor()

if create:
    print("create new db")
    c.execute("CREATE TABLE users (uid INT PRIMARY KEY, name TEXT);")
    c.execute("CREATE TABLE posts (pid INT PRIMARY KEY, uid INT, text TEXT);")

    c.execute("INSERT INTO users VALUES (21, 'user A');")
    c.execute("INSERT INTO users VALUES (2671, 'user B');")

    c.execute("INSERT INTO posts VALUES (1, 21, 'abc ..');")
    c.execute("INSERT INTO posts VALUES (2, 21, 'qwe xyz');")
    c.execute("INSERT INTO posts VALUES (3, 2671, 'test');")

    conn.commit()

maxUid = 100
for r in c.execute("SELECT * FROM users WHERE uid < ?;", (maxUid,)):
    print(r)

for r in c.execute("""SELECT u.name, p.text FROM
                    users AS u JOIN posts AS p ON (u.uid = p.uid);"""):
    print(r)

```

/// Przykład PHP

```

<?php
$sql_conn = mysqli_connect('server', 'user', 'password', 'database');
if (!$sql_conn) {
    echo "ERROR: " . mysqli_connect_error();
    exit;
} else {
    echo "Connected to MySQL: " . mysqli_get_host_info($sql_conn);
}

$query = $db->query("SELECT u.name, p.text FROM users AS u JOIN posts AS p ON (u.uid = p.uid);");
while ($row = $query->fetch_array()) {
    print($row["name"] . ": " . $row["text"] . "\n");
}
?>

```

(Graficzny) interfejs użytkownika

Jeżeli zachodzi potrzeba interakcji z użytkownikiem programu, zamiast wspomnianego wcześniej prostego pytania o kolejne parametry z użyciem standardowego wyjścia i wejścia, często stosuje się pseudo-graficzny lub graficzny interfejs użytkownika. Interfejs pseudo-graficzny jest zasadniczo bardziej wyrafinowaną formą interakcji przez standardowe wyjście i wejście programu i wykorzystuje standardowe wyjście do rysowania różnego typu okienek dialogowych, itp. w trybie tekstowym.

Przykład Python

```

# interfejs pseudo graficzny z użyciem biblioteki curses ...
# a tak naprawdę programu dialog z niej korzystającego

from dialog import Dialog
from time import sleep

d = Dialog(dialog="dialog")
# tak naprawdę jest to wrapper

```

```

# wykorzystujący program o nazwie "dialog"
#
# jest w stanie (w ograniczonym zakresie)
# współpracować z innymi tego typu narzędziami
# (także działającymi w trybie graficznym)
# takimi jak: whiptail, kdialog, ...
# d = Dialog(dialog="kdialog", compat="kdialog")

d.set_background_title("ABC ...")

res = d.yesno("Tak czy NIE?")

if res == d.OK:
    d.msgbox("TAK !!!")
else:
    d.msgbox("NIE !!!")

d.gauge_start("proszę czekać ... będzie rozmowa ...")
for i in range(0, 101, 5):
    d.gauge_update(i)
    sleep(1)

```

Przykład Python

interfejs pseudo graficzny z użyciem Qt4

```

import sys
from PyQt4 import QtGui, QtCore

app = QtGui.QApplication(sys.argv)

win = QtGui.QWidget()
win.resize(250, 150)
win.move(300, 300)
win.setWindowTitle('ABC ...')

def showDialog():
    reply = QtGui.QMessageBox.question(None, 'Message',
        "Tak czy nie?", QtGui.QMessageBox.Yes |
        QtGui.QMessageBox.No, QtGui.QMessageBox.No)

    if reply == QtGui.QMessageBox.Yes:
        print("YES")
    elif reply == QtGui.QMessageBox.No:
        print("NO")
    else:
        print("cos innego?!")

button = QtGui.QPushButton('Nacisnij mnie', win)
button.clicked.connect(showDialog)
button.resize(button.sizeHint())
button.move(
    250/2-button.sizeHint().width()/2,
    150/2-button.sizeHint().height()/2
)

win.show()

```

Przykład Python

interfejs pseudo graficzny z użyciem Gtk 3

```

import gi
gi.require_version("Gtk", "3.0")
from gi.repository import Gtk

window = Gtk.Window(title="Hello World")

def showDialog(caller):
    dialog = Gtk.MessageDialog(window, 0,
        Gtk.MessageType.QUESTION, Gtk.ButtonsType.YES_NO,
        "Ważne pytanie !!!")
    dialog.format_secondary_text("Tak czy nie?")

    response = dialog.run()
    if response == Gtk.ResponseType.YES:
        print("TAK !!!")
    elif response == Gtk.ResponseType.NO:
        print("NIE !!!")

```

```

else:
    print("bez odpowiedzi!?!")

dialog.destroy()

button1 = Gtk.Button("Nacisnij mnie")
button1.connect("clicked", showDialog)

window.add(button1)

window.show_all()
window.connect("delete-event", Gtk.main_quit)
Gtk.main()

```

Programowanie sprzętu

Typowo programy komputerowe uruchamiane są pod kontrolą jakiegoś systemu operacyjnego, który odpowiada za realizację sporej części funkcji określonych przez bibliotekę standardową (np. operacji plikowych, standardowego wejścia i wyjścia) oraz zapewnia możliwość równoczesnego działania wielu procesów (ich przełączania). Jednak niekiedy zachodzi potrzeba stworzenia programu który będzie działał bezpośrednio na sprzęcie bez pośrednictwa systemu operacyjnego. Może to mieć miejsce np. wtedy gdy zasoby danej platformy sprzętowej są zbyt ograniczone aby uruchomić na niej system operacyjny i naszą aplikację lub tworzymy system operacyjny.

Przerwania

Przerwanie (żądanie przerwania, IRQ) jest sygnałem powodującym wstrzymanie wykonywania aktualnego kodu i wykonanie zdefiniowanej (dla danego przerwania) procedury jego obsługi. Przerwania mogą być związane z:

- sygnałem otrzymanym od zewnętrznego układu (tzw. przerwania sprzętowe) - mające na celu np. poinformowanie o zakończeniu jakiejś operacji wejścia wyjścia, otrzymaniu danych na jakimś interfejsie,
- wykonywanym kodem (tzw. przerwania programowe) - zarówno wywołane celowo (np. celem komunikacji z systemem operacyjnym), jak też pojawiające się na skutek nietypowych / błędnych działań - tzw. "wyjątki" (np. dzielenia przez zero).

Na wielu architekturach program może wygenerować dowolne z przerw (także tych związanych z sprzętem, czy też wyjątków) poprzez wywołanie stosownej instrukcji procesora której argumentem jest numer przerwania. Generowanie przerw instrukcją procesora jest wykorzystywane m.in. celem realizacji wywołań systemowych (czyli wywołań funkcji jądra z poziomu programu użytkownika). W przypadku Linuxa na architekturze x86 za wywołania systemowe odpowiada przerwanie o numerze 0x80, numer funkcji systemowej musi być uprzednio umieszczony w rejestrze eax, a jej kolejne argumenty w rejestrach ebx, ecx, edx, edi, esi, ebp; podobnie status operacji i wartość zwracana jest umieszczana w odpowiednich rejestrach procesora.

Istnieje możliwość zablokowania obsługi wszystkich przerw pochodzących z zewnątrz (z wyjątkiem przerw nie maskowanych) stosowaną instrukcją procesora. Obsługa przerw przez procesor polega na zidentyfikowaniu (w oparciu o numer/źródło przerwania i dane zawarte w obszarze pamięci często nazywanym tablicą wektorów przerw) adresu procedury obsługi przerwania i wykonaniu jej w sposób podobny do wykonania funkcji, czyli z odłożeniem aktualnego stanu procesora na stos (celem umożliwienia powrotu do niego po zakończeniu procedury obsługi przerwania w celu kontynuowania przerwanego kodu). Jeżeli procedura obsługi przerwania ma inny poziom uprzywilejowania (np. działa na prawach jądra systemu operacyjnego) to procesor może także dokonać przełączenia stosu.

Z punktu widzenia wielozadaniowego systemu operacyjnego szczególnie istotne jest przerwanie sprzętowe związane z odmierzeniem ustalonego czasu (zwane przerwaniem zegarowym), pozwala ono na odebranie kontroli aktualnie wykonywanemu procesowi, przekazanie jej systemowi operacyjnemu i umożliwienia przełączenia na inny proces.

Mikrokontrolery

Programowanie mikrokontrolerów, a zwłaszcza z wykorzystaniem peryferiów w które są wyposażone, dość znacznie różni się od tworzenia aplikacji działających pod kontrolą systemu operacyjnego (nawet pomimo użycia tego samego języka programowania - najczęściej C lub C++), a także pomiędzy poszczególnymi typami mikrokontrolerów. Korzystanie z peryferiów oferowanych przez mikrokontroler polega głównie na wykonywaniu odczytu i zapisów do rejestrów zgodnie z dokumentacją danego układu oraz obsłudze odpowiednich przerw. Niekiedy takie surowe operowanie na rejestrach obudowane jest przez funkcji biblioteki wspomagającej programowanie danej platformy, z których programista może korzystać aby w czytelniejszy sposób zapisać wykonywany algorytm.

Także proces kompilacji kodu dla takich platform przebiega odmiennie niż przy „normalnym”

programowaniu – bardzo często wykorzystywana jest cross kompilacja, czyli kompilator działa pod kontrolą innej architektury niż ta dla której tworzy kod. W związku z tym wykorzystywane są inne kompilatory wraz z własnymi zestawami bibliotek standardowych (stanowiące tzw. toolchain dla danej platformy). Bardzo istotne jest też często określenie w opcjach takiego kompilatora szczegółów mikrokontrolera dla którego chcemy zbudować dany kod (typowo toolchain dotyczy całej rodziny, a konkretny model mikrokontrolera, posiadający konkretny zbiór peryferiów, itd określany jest w opcjach).

Przykładowe kody wraz z opisem kompilacji lub plikiem Makefile automatyzującym kompilację dla dwóch chyba najpopularniejszych rodzin mikrokontrolerów udostępniony jest w postaci repozytoriów git:

- [AVR-examples](#) - dla 8 bitowych mikrokontrolerów AVR
- [STM32-examples](#) - dla 32 bitowych mikrokontrolerów STM32 (rdzeń ARM)

obsługa sprzętu w ramach OS

Obsługa sprzętu nie zawsze wiąże się z tworzeniem kodu działającego bez systemu operacyjnego, na większych układach może działać normalny system operacyjny (np. Linux na płytkach typu Raspberry Pi czy Orange Pi). W takiej sytuacji obsługę sprzętu można zrealizować poprzez własny moduł jądra za nią odpowiedzialny lub (gdy to jest możliwe) poprzez funkcje udostępnione do przestrzeni użytkownika przez system operacyjny i jakąś bibliotekę. Przykładem drugiego podejścia może być wykorzystywanie plików funkcji operujących na przestrzeni I/O (plik nagłówkowy asm/io.h), funkcji realizujących komunikację I2C (plik nagłówkowy linux/i2c-dev.h), czy też biblioteki libusb pozwalającej na obsługę komunikacji USB.

komunikacja I2C

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#include <linux/i2c-dev.h>
#ifndef I2C_FUNC_I2C
#include <i2c/smbus.h>
#endif

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

// wymagany pakiet libi2c-dev, kompilacja:
// dla libi2c-dev < 4:
// gcc      komunikacja_I2C.c
// dla libi2c-dev >= 4:
// gcc -li2c komunikacja_I2C.c

void help(const char *prog_name) {
    fprintf(stderr, "USAGE (without on chip register addresses):\n");
    fprintf(stderr, " - read:  %s i2c_dev addr r -\n", prog_name);
    fprintf(stderr, " - write: %s i2c_dev addr w - data\n", prog_name);
    fprintf(stderr, "\n");
    fprintf(stderr, "USAGE (with on chip register addresses):\n");
    fprintf(stderr, " - read:  %s i2c_dev addr r reg_addr\n", prog_name);
    fprintf(stderr, " - write: %s i2c_dev addr w reg_addr data\n", prog_name);
    fprintf(stderr, "\n");
    fprintf(stderr, "i2c_dev  = /dev/i2c-... device path\n");
    fprintf(stderr, "addr     = I2C_devide_address\n");
    fprintf(stderr, "reg_addr = register (data) address in I2C chip\n");
    fprintf(stderr, "data     = data to write to device\n");
}

int main(int argc, char *argv[]) {
    int fd, i2c_addr, reg_addr, res, d;

    if (argc < 4) {
        help(argv[0]);
        return 1;
    }

    // otwarcie urządzenia
    fd = open(argv[1], O_RDWR);
    if (fd < 0) {
        fprintf(stderr, "Error open I2C device file (%s): %s\n", argv[1], strerror(errno));
    }
}
```

```

    return 2;
}

// ustawienie adresu slave
i2c_addr = strtol(argv[2], NULL, 0);
res = ioctl(fd, I2C_SLAVE, i2c_addr);
if (res < 0) {
    fprintf(stderr, "ERROR set device address to 0x%02x: %s\n", i2c_addr, strerror(errno));
    return 3;
}

if (argv[3][0] == 'r' && argc == 5) { // odczyt

    if (argv[4][0] != '-') {
        // jeżeli jest podany to ustawiamy adres rejestru do odczytu
        // warto zauważyć że jest to robione jako WRITE do urządzenia I2C
        reg_addr = strtol(argv[4], NULL, 0);
        res = i2c_smbus_write_byte(fd, reg_addr);
        if (res < 0) {
            fprintf(stderr, "ERROR write (register address) %d to i2c device 0x%02x on %s: %s\n", reg_addr, i2c_addr, argv[1], strerror(errno));
            return 4;
        }
    }

    // czytamy dane z urządzenia
    res = i2c_smbus_read_byte(fd);
    if (res < 0) {
        fprintf(stderr, "ERROR read from i2c device 0x%02x on %s: %s\n", i2c_addr, argv[1], strerror(errno));
        return 5;
    }
    printf("0x%02x\n", res);
} else if (argv[3][0] == 'w' && argc == 6) { // zapis
    d = strtol(argv[5], NULL, 0);

    if (argv[4][0] != '-') {
        // jeżeli jest podany adres rejestru do zapisu to go używamy i wykonujemy zapis
        reg_addr = strtol(argv[4], NULL, 0);
        res = i2c_smbus_write_byte_data(fd, reg_addr, d);
        if (res < 0) {
            fprintf(stderr, "ERROR write %d to i2c device 0x%02x, register 0x%02x on %s: %s\n", d, i2c_addr, reg_addr, argv[1], strerror(errno));
            return 5;
        }
    } else {
        // w przeciwnym razie po prostu zapisujemy dane do urządzenia
        res = i2c_smbus_write_byte(fd, d);
        if (res < 0) {
            fprintf(stderr, "ERROR write %d to i2c device 0x%02x on %s: %s\n", d, i2c_addr, argv[1], strerror(errno));
            return 5;
        }
    }
} else if (argv[3][0] == 'R' && argc == 5 && argv[4][0] != '-') { // odczyt z rejestru jedną funkcją

    reg_addr = strtol(argv[4], NULL, 0);
    res = i2c_smbus_read_byte_data(fd, reg_addr); // odczyt z rejestru o danym adresie może być wykonany
    if (res < 0) {
        fprintf(stderr, "ERROR read from i2c device 0x%02x, register 0x%02x on %s: %s\n", i2c_addr, reg_addr, argv[1], strerror(errno));
        return 5;
    }
    printf("0x%02x\n", res);
} else {
    help(argv[0]);
    return 1;
}

close(fd);
}

```

Wzorce projektowe, algorytmy, itp

Obiektowość

Obiektowość jest podejściem do programowania polegającym na łączeniu danych i powiązanych z nimi

procedur w obiekty. W rezultacie zamiast grupy funkcji operujących na jakiejś strukturze danych i osobnych instancjach tej struktury (np. napis typu `char*` i funkcje z rodziny `string.h`) operuje się obiektami zawierającymi w sobie dane i posiadającym metody na nich operujące (np. napis typu `std::string`). Z obiektowością związane są zagadnienia takie jak: określanie interfejsów klas odnoszących się jedynie do kluczowych dla danego zagadnienia elementów i ukrywaniu pozostałych aspektów wewnątrz implementacji klasy, wydzielanie wspólnych cech obiektów różnego typu do klas bazowych po których dziedziczą.

Dziedziczenie, polimorfizm, itp

Dziedziczenie jest jednym z mechanizmów pozwalających na konstrukcję klas opisujących bardziej złożone obiekty z klas prostszych.

```
/// Przykład C++

// klasa abstrakcyjna - definiująca interfejs dla jakiejś grupy obiektów
// fakt bycia klasą abstrakcyjną wynika z niezdefiniowanej metody wirtualnej
// z faktu tego wynika niemożność utworzenia obiektów będących bezpośrednimi
// instancjami takiej klasy
class Kształt {
public:
    virtual float objetosc() = 0;
    // użycie virtual wymusza wywoływanie metody z klasy potomnej
    // kiedy odwołujemy się przy użyciu klasy bazowej
};

// klasy dziedzicząca po klasie Kształt
class Kula : public Kształt {
    float promien;
public:
    Kula(float r) {
        promien = r;
    }
    float objetosc() {
        return 1.25 * 3.14 * promien * promien * promien;
    }
};

class Szescian : public Kształt {
    float bok;
public:
    Szescian(float a) {
        bok = a;
    }
    float objetosc() {
        return bok*bok*bok;
    }
};

struct Kolor {
    char r, g, b;
};

class Cena {
public:
    Cena(float c);
};

class Material {
};

// klasa dziedzicząca po kilku klasach bazowych
struct Opakowanie : public Kształt, public Kolor, public Cena {
    Kształt *ksztalt;
    Material *material;
public:
    Opakowanie(Kształt *k, const Kolor& kolor, float c) :
        // wywołanie konstruktorów klasy bazowej
        // jawnego dla Cena i kopiującego dla Kolor
        Cena(c), Kolor(kolor)
    {
        // inicjalizacja zmiennej wskaźnikowej ksztalt
        // dzięki rozwiązaniu z zmienną przechowującą informację o kształcie
        // nie ma potrzeby tworzenia różnych klas Opakowanie dla różnych kształtów
        ksztalt = k;
        material = 0;
    }
};
```

```

Opakowanie(Kształt *k, float c) :
    // wywołanie konstruktora klasy bazowej Cena
    // oraz inicjalizacja zmiennych kształt i material
    Cena(c), kształt(k), material(0)
{
    // inicjalizacja pól odziedziczonych po strukturze Kolor
    r = 0x96; g = 0xFF; b = 0x03;
}

float objetosc() {
    return kształt->objetosc();
}

// jest to alternatywna w stosunku co do dziedziczenia
// metoda rozszerzania interfejsu jakiejś klasy

// dziedziczenie (w przypadku dziedziczenia public) dodaje metody
// klasy bazowej bezpośrednio do interfejsu klasy dziedziczącej
// natomiast to rozwiązanie pozwala na pobranie poszczególnych klas
// opisujących złożony obiekt niezależnie i operowaniu na ich interfejsie
Material* getMaterial() {
    return material;
}

void setMaterial(Material *m) {
    material = m;
}
};

```

Szablony

Niektóre z języków z silnym typowaniem pozwalają na definiowanie szablonów funkcji i klas, stanowiące ich uogólnienie dla różnych typów.

```

/// Przykład C++

// Szablon funkcji wypisującej na standardowe wyjście
// listę dowolnego typu. Wymagane jest tylko aby dla tego
// typu był zdefiniowany operator << z std::ostream.

template <typename T> void wypiszListe(std::list<T>& l) {
    for (auto i : l) {
        std::cout << i << "\n";
    }
}

// przykład użycia
void wypisz() {
    std::list<int> x={1, 3, 7, 2, 3};
    wypiszListe(x);

    std::list<float> z={2.7, 5.0, 3.1, 3.9};
    wypiszListe(z);
}

```

Tworzenie i rozbudowa obiektów

metoda wytwórcza

jest to kreatywny wzorzec projektowy (inaczej nazywany *wirtualny konstruktor*) realizowany poprzez utworzenie klasy fabryki z metodą statyczną odpowiedzialną za tworzenie różnych produktów (dziedziczących po wspólnej klasie bazowej).

Efekt zastosowania jest ukrycie szczegółów implementacji poszczególnych produktów przez użytkowników (korzysta on tylko z klasy bazowej produktu i fabryki).

```

/// Przykład C++
#include <iostream>

struct Product {
    virtual void info() = 0;
    virtual ~Product() {}
};

```

```

struct ProductA : public Product {
    virtual void info() {
        std::cout << "ProductA: a, b, c\n";
    }
};

struct ProductB : public Product {
    virtual void info() {
        std::cout << "ProductB: a, f, g\n";
    }
};

struct ProductFactory {
    static Product* getProduct(const std::string& type) {
        if (type == "A")
            return new ProductA();
        else if (type == "B")
            return new ProductB();
        else
            return NULL;
    }
};

int main(int argc, char *argv[]) {
    std::string id = "A";
    if (argc > 1 && argv[1][0] == 'B')
        id = "B";

    Product* p = ProductFactory::getProduct(id);
    p->info();
    delete p;
}

```

fabryka abstrakcyjna

jest to kreatywny wzorec projektowy realizowany poprzez utworzenie klasy fabryki z metodą statyczną odpowiedzialną za tworzenie różnych typów fabryk dziedziczących po wspólnej klasie bazowej. Klasa bazowa fabryki określa gamę produktów z użyciem klas bazowych produktów, w efekcie każda z fabryk produkuje taką samą gamę produktów, ale do ich tworzenia może korzystać z specyficznych dla danej implementacji fabryki klas dziedziczących po produktach bazowych.

Efekt zastosowania jest ukrycie szczegółów implementacji poszczególnych produktów przez użytkownika (korzysta on tylko z klasy bazowej produktu i fabryki). W odróżnieniu od metody wytwórczej stosowany jest do produkcji spójnych gam produktów dzięki jednokrotnemu określaniu typu produktów przy tworzeniu fabryki (zamiast niezależnego określania typu każdego produktu). Pozwala tylko raz określić używany typ fabryki i wytwarzać produkty uzyskaną fabryką konkretnego typu, jest to przydatne wtedy gdy konkretna implementacja fabryki jest zależna np. od stosowanej biblioteki lub systemu.

```

/// Przykład C++
#include <iostream>

struct Product {
    virtual void info() = 0;
    virtual ~Product() {}
};

struct ProductA : public Product {
    virtual void info() {
        std::cout << "ProductA: a, b, c\n";
    }
};

struct ProductB : public Product {
    virtual void info() {
        std::cout << "ProductB: a, f, g\n";
    }
};

struct Factory {
    static Factory* getFactory(const std::string& type);
    virtual Product* getProduct() = 0;
    // fabryka taka może z łatwością produkować kilka

```

```

// różnych produktów (całą rodzinę produktów)
virtual ~Factory() {}
};

struct FactoryA: public Factory {
    virtual Product* getProduct() {
        return new ProductA();
    }
};

struct FactoryB: public Factory {
    virtual Product* getProduct() {
        return new ProductB();
    }
};

Factory* Factory::getFactory(const std::string& type) {
    if (type == "A")
        return new FactoryA();
    else if (type == "B")
        return new FactoryB();
    else
        return NULL;
}

int main(int argc, char *argv[]) {
    std::string id = "A";
    if (argc > 1 && argv[1][0] == 'B')
        id = "B";

    Factory* f = Factory::getFactory(id);
    Product* p = f->getProduct();
    p->info();
    delete p;
    delete f;
}

```

budowniczy

jest to kreatywny wzorzec projektowy realizowany poprzez utworzenie klasy kierownika korzystającego z bazowego interfejsu klasy budowniczego do realizacji jakiegoś algorytmu (często konwersji formatów). Użytkownik odpowiedzialny jest za utworzenie konkretnego budowniczego, przekazanie wskaźnika na niego kierownikowi (ewentualnie wraz z danymi potrzebnymi do budowy, np. danymi które mają zostać poddane konwertowaniu) oraz odebraniu wyniku od budowniczego.

Efektom jest pełne ukrycie szczegółów implementacji przed klasą implementującą kierownika (nie wie nic o produkcie, używa tylko bazowego budowniczego).

```

/// Przykład C++
#include <iostream>

struct Product {
    void info() {
        std::cout << "Product:\n";
        std::cout << "  " << el1 << "\n";
        std::cout << "  " << el2 << "\n";
    }
    void setElement1(const std::string& e) {
        el1 = e;
    }
    void setElement2(const std::string& e) {
        el2 = e;
    }
private:
    std::string el1, el2;
};

struct ProductBuilder {
    virtual ~ProductBuilder() {}

    void createProduct() {
        p = new Product();
    }
    Product* getProduct() {
        return p;
    }
};

```

```

    }
    // w takiej implementacji nie odebranie lub
    // nie skasowanie produktu przez klienta będzie
    // prowadziło do wycieków pamięci

    virtual void buildElement1(const std::string& d) = 0;
    virtual void buildElement2() = 0;

protected:
    Product* p;
};

struct ProductDirector {
    void setData(const std::string& d) {
        this->d = d;
    }
    void setBuilder(ProductBuilder* b) {
        this->b = b;
    }
    void createProduct() {
        b->createProduct();
        b->buildElement1(d);
        b->buildElement2();
    }
private:
    ProductBuilder* b;
    std::string d;
};

struct ProductBuilderA : public ProductBuilder {
    void buildElement1(const std::string& d) {
        p->setElement1(d + d);
    };
    void buildElement2() {
        p->setElement2("ab");
    };
};

struct ProductBuilderB : public ProductBuilder {
    void buildElement1(const std::string& d) {
        p->setElement1("b" + d);
    };
    void buildElement2() {
        p->setElement2("bb");
    };
};

int main(int argc, char *argv[]) {
    ProductBuilder* b;
    if (argc > 1 && argv[1][0] == 'B')
        b = new ProductBuilderB();
    else
        b = new ProductBuilderA();

    ProductDirector* d = new ProductDirector();
    d->setData("x");
    d->setBuilder(b);
    d->createProduct();

    Product* p = b->getProduct();
    p->info();
    delete p;
    delete d;
    delete b;
}

```

singleton

Singleton jest kreatywnym wzorcem projektowym zapewniającym istnienie tylko jednej instancji (jednego obiektu) klasy go implementującej. Uzyskiwane jest to poprzez ukrycie konstruktorów i zastąpienie ich metodą statyczną zwracającą zawsze ten sam obiekt danej klasy.

```

/// Przykład C++
#include <iostream>

```

```

class Singleton {
private:
    // konstruktor, konstruktor kopiujący oraz operator
    // przypisania są prywatne aby uniemożliwić tworzenie
    // lub kopiowanie obiektów tej klasy z zewnątrz
    Singleton();
    Singleton(const Singleton&) {}
    Singleton& operator=(const Singleton&);

    // destruktor też prywatny ... dla zabezpieczenia przed
    // możliwością zrobienia delete na uzyskanym wskaźniku
    ~Singleton() {}

    // klasa zawiera prywatny statyczny wskaźnik na siebie
    // (na swoją jedyną instancję)
    static Singleton* objPtr;

public:
    // a także metodę służącą pobraniu tego wskaźnika
    // (i utworzeniu obiektu gdy nie istnieje)
    static Singleton* getPtr() {
        // alternatywnie zamiast wskaźnika można użyć
        // statycznego obiektu tej klasy zadeklarowanego
        // wewnątrz tej funkcji
        if (!objPtr)
            objPtr = new Singleton();

        return objPtr;
    }
};

Singleton* Singleton::objPtr = 0;

Singleton::Singleton() {
    std::cout << "konstruktor Singleton\n";
}

int main() {
    Singleton* a = Singleton::getPtr();
    Singleton* b = Singleton::getPtr();
    std::cout << a << " == " << b << "\n";
}

```

dekorator

jest strukturalnym wzorcem projektowym pozwalającym na rozbudowę funkcjonalności jakiejś klasy bez ingerencji w nią, realizowany jest poprzez dziedziczenie po wspólnym z obudowywaną klasą interfejsie i zawieranie w sobie wskaźnika na obudowywaną klasę. Dekorator reimplementuje wybrane funkcje klasy oryginalnej jednak wywołując w nich funkcje klasy oryginalnej, co pozwala na wielokrotne obudowywanie pierwotnego obiektu.

```

/// Przykład C++
#include <iostream>

struct Product {
    virtual ~Product() {}
    virtual void info() = 0;
};

struct ProductA : public Product {
    void info() {
        std::cout << "Product:\n";
        std::cout << "  ab\n";
    }
};

struct DecoratorX : public Product {
    DecoratorX(Product* p) {
        this->p = p;
    }

    void info() {
        p->info();
        std::cout << "  xx\n";
    }
}

```

```

protected:
    Product* p;
};

struct DecoratorY : public Product {
    DecoratorY(Product* p) {
        this->p = p;
    }

    void info() {
        p->info();
        std::cout << " yy\n";
    }
protected:
    Product* p;
};

int main(int argc, char *argv[]) {
    Product* p = new ProductA();
    Product* p1 = new DecoratorX(p);
    Product* p2 = new DecoratorY(p1);

    p1->info();
    p2->info();

    delete p2;
    delete p1;
    delete p;
}

```

kompozyt

jest strukturalnym wzorcem projektowym pozwalającym na zapewnienie interfejsu zgodnego z interfejsem obiektu dla grupy obiektów.

```

/// Przykład C++
#include <iostream>
#include <list>

struct Product {
    virtual void info() = 0;
    virtual ~Product() {};
};

struct ProductA : public Product {
    virtual void info() {
        std::cout << "ProductA: a, b, c\n";
    }
};

struct ProductB : public Product {
    virtual void info() {
        std::cout << "ProductB: a, f, g\n";
    }
};

class ProductComposite : public Product {
public:
    std::list<Product*> elements;

    virtual void info() {
        for (auto e : elements)
            e->info();
    }

    ~ProductComposite() {
        for (auto e : elements)
            delete e;
    }
};

int main(int argc, char *argv[]) {
    ProductComposite* pc = new ProductComposite();

    pc->elements.push_back(new ProductA());
}

```

```

pc->elements.push_back(new ProductB());
pc->elements.push_back(new ProductA());

pc->info();
delete pc;
}

```

Komunikacja między obiektami

listener

Obserwator (określany też jako listener) jest czynnościowym wzorcem projektowym pozwalającym jednemu obiektowi powiadomić wiele innych obiektów o jakimś zdarzeniu, obiekty te powinny być uprzednio zarejestrowane w klasie powiadamiającej jako zainteresowane danym zdarzeniem). Może być też wykorzystywany do realizacji wzorca *łańcuch zobowiązań* wtedy gdy powiadamiane obiekty mają / mogą wykonać jakieś operacje i od kodu powrotu funkcji powiadamiającej zależy powiadamianie dalszych obiektów.

```

/// Przykład C++
#include <iostream>
#include <map>

struct Obserwator {
    virtual bool onUpdate(int eventID) = 0;
};

struct Obserwowany {
    void addListener(Obserwator* l, int key) {
        // sprawdź czy jest już zarejestrowany
        auto subset = listeners.equal_range(key);
        for (auto iter=subset.first; iter!=subset.second; ++iter)
            if (iter->second == l) return;

        // zarejestruj
        listeners.insert(std::pair<int, Obserwator*>(key, l));
    }

    void remListener(Obserwator* l) {
        for (auto iter=listeners.begin(); iter!=listeners.end(); ++iter) {
            if (iter->second == l) {
                listeners.erase(iter);
                return;
            }
        }
    }

    // klasyczny wzorec listenera
    void callListenersAll(int eventID) {
        // mapa zapewnia wywoływanie listenerów wg kolejności kluczy
        for (auto& l : listeners) {
            l.second->onUpdate(eventID);
        }
    }

    // wzorec listenera połączony z łańcuchem zobowiązań
    void callListeners(int eventID) {
        // mapa zapewnia wywoływanie listenerów wg kolejności kluczy
        for (auto& l : listeners) {
            if (l.second->onUpdate(eventID))
                break;
        }
    }
};

protected:
    std::multimap<int, Obserwator*> listeners;
};

struct ObserwatorA : public Obserwator {
    virtual bool onUpdate(int eventID) {
        std::cout << " ObserwatorA z id=" << eventID << "\n";
        if (eventID == 4)
            return true;
        return false;
    }
};

```

```

struct ObserwatorB : public Obserwator {
    virtual bool onUpdate(int eventID) {
        std::cout << " ObserwatorB z id=" << eventID << "\n";
        if (eventID == 7)
            return true;
        return false;
    };
};

int main(int argc, char *argv[]) {
    Obserwowany* o = new Obserwowany();
    Obserwator* o1 = new ObserwatorA();
    Obserwator* o2 = new ObserwatorA();
    Obserwator* o3 = new ObserwatorB();

    o->addListener(o1, 30);
    o->addListener(o2, 10);
    o->addListener(o3, 20);

    std::cout << "callAll:\n";
    o->callListenersAll(7);

    std::cout << "call:\n";
    o->callListeners(7);

    delete o;
    delete o1;
    delete o2;
    delete o3;
}

```

Mediator

jest czynnościowym wzorcem projektowy realizującym przekazywanie wiadomości pomiędzy obiektami poprzez pośrednika, w przypadku nie natychmiastowego wysyłania wiadomości przez mediatora pozwala także na realizację *kolejki wiadomości* / *pętli komunikatów*.

```

/// Przykład C++
#include <iostream>
#include <map>

struct Receiver {
    virtual void receive(std::string msg) = 0;
};

struct Mediator {
    void addReceiver(Receiver* l, int key) {
        // sprawdź czy jest już zarejestrowany
        auto subset = receivers.equal_range(key);
        for (auto iter=subset.first; iter!=subset.second; ++iter)
            if (iter->second == l) return;

        // zarejestruj
        receivers.insert(std::pair<int, Receiver*>(key, l));
    }

    void remReceiver(Receiver* l) {
        for (auto iter=receivers.begin(); iter!=receivers.end(); ++iter) {
            if (iter->second == l) {
                receivers.erase(iter);
                return;
            }
        }
    }

    bool send(std::string msg, int dstID) {
        // gdyby zamiast natychmiastowego wywołania receive() u adresatów
        // zaimplementować gromadzenie wiadomości w jakiejś strukturze
        // i osobną funkcję wysyłającą kolejne wiadomości z tej struktury
        // to wzorec ten pozwoliłby na realizację kolejki wiadomości
        // (pętli komunikatów)
        //
        // struktura przechowująca zakolejkowane do wysłania wiadomości może
        // uwzględniać kolejność ich odebrania przez Mediatora lub wartość
    }
}

```

```

        // priorytetu zawartego w wiadomości (kolejka priorytetowa)
        auto subset = receivers.equal_range(dstID);
        for (auto& iter=subset.first; iter!=subset.second; ++iter) {
            iter->second->receive(msg);
        }
    }

protected:
    std::multimap<int, Receiver*> receivers;
};

```

Regulator PID

Regulator PID jest to algorytm regulacji parametru procesu działający w pętli sprzężenia zwrotnego posiadający członki: proporcjonalny (P), całkujący (I) i różniczkujący (D).

Wyjściem algorytmu jest wartość zmiany jakiegoś sygnału sterującego - może być wykorzystana bezpośrednio przy sterowaniu krokowym lub akumulowana celem uzyskania stałej wartości sygnału sterującego.

Wejściem algorytmu jest wartość mierzona (bądź od razu różnica wartości zadanej i mierzonej). Jeżeli kierunek zmiany sterowania jest zgodny ze zmianą wartości mierzonej (zwiększenie wartości sygnału sterującego powoduje zwiększenie wartości mierzonej) to należy odejmować wartość mierzona od zadanej, w przeciwnym razie od zadanej mierzoną.

```

/// Przykład C++
class PID {
    // poprzednia różnica między wartością zadaną a otrzymaną
    // (poprzedni błąd regulacji / uchyb)
    float lastDiff;

    // poprzednia wartość otrzymana (zmienna procesu)
    float lastVal;

    // część całkująca, akumulowana pomiędzy krokami
    float integral;

public:
    // nastawa - wartość zadana
    float setPoint;

    // wartość wyjścia dla sterowania krokowego
    // (akumulacja w układzie realizującym)
    float outStep;

    // wartość wyjścia dla sterowania sygnałem
    float outValue;

    // parametry regulatora PID
    float Kp, Ki, Kd;

    // limity wartości sterowanej
    float outValueMin, outValueMax;

    // konstruktor - inicjalizacja parametrów
    PID (
        float initVal, float sp,
        float kp, float ki, float kd, float min, float max
    ) :
        lastDiff(0), lastVal(initVal), integral(0),
        setPoint(sp), outStep(0), outValue(0),
        Kp(kp), Ki(ki), Kd(kd),
        outValueMin(min), outValueMax(max)
    {}

    int doStep(float inputVal, float stepTime) {
        // obliczymy aktualny błąd regulacji
        // (na podstawie odczytanej wartości wejściowej)
        float diff = setPoint - inputVal;

        // wyłączamy regulację gdy prowadziłby do przesterowania
        if (outValue > outValueMax && diff > 0)
            return 1;
        if (outValue < outValueMin && diff < 0)

```

```

        return -1;

        // całkowanie przybliżamy jako suma pól trapezów
        integral += (diff + lastDiff) / 2 * stepTime;

        // różniczkowanie przybliżamy jako tangens nachylenia
        // prostej pomiędzy poprzednim krokiem a obecnym
        // celem złagodzenia odpowiedzi na zmiany wartości zadanej
        // różniczkujemy sygnał wejściowy
        float derivative = -(inputVal - lastVal) / stepTime;
        // a nie błąd regulacji: derivative = (diff - lastDiff) / stepTime;

        // obliczenie wartości sygnału sterującego na podstawie tego kroku
        outStep = Kp*diff + Ki*integral + Kd*derivative;

        // akumulacja sygnału sterującego
        outValue += outStep;

        // zapamiętanie aktualnego błędu regulacji
        // jako poprzedni dla następnego kroku
        lastDiff = diff;

        // zapamiętanie aktualnej wartości wejściowej
        // jako poprzedniej dla następnego kroku
        lastVal = inputVal;

        return 0;
    }
};

int main() {
    PID pid(readInput(), 21.0, 1.0, 0.0, 0.0, -100.0, 100.0);

    // okres działania algorytmu [us]
    unsigned int stepTime = 250000;

    // pętla algorytmu
    while(1) {
        // odczyt wejścia
        float curVal = readInput();

        // obliczenie wartości sterującej z użyciem PID
        pid.doStep( curVal, stepTime );

        // wystawienie wartości sterującej
        setOutput(pid.outValue);

        // odczekanie do następnego kroku
        usleep(stepTime);
    }
}

```

Sieci komputerowe

Sieć komputerowa jest to zbiór hostów (urządzeń wykorzystujących sieć do komunikacji, takich jak komputery, drukarki sieciowe, ...) oraz infrastruktury sieciowej (okablowanie, urządzenia pośredniczące w komunikacji). W przypadku bardziej abstrakcyjnego patrzenia na zagadnienia sieciowe (właśnie od strony protokołów wyższych warstw niż sprzętowa - takich jak IP) na sieć można patrzeć tylko jako na zbiór hostów (posiadających identyfikujące je numery). Dla funkcjonowania sieci konieczne jest zapewnienie dobrze określonych zasad jej działania - właśnie zbiorami takich zasad wymiany informacji są protokoły sieciowe.

Sieci komputerowe działają na zasadzie przesyłania informacji w postaci porcji, z których każda posiada co najmniej informację o adresie odbiorcy (zwykle też nadawcy), nazywanych ramkami lub pakietami. Kierowanie pakietów w odpowiednie miejsce odbywa się na podstawie adresu pakietu i nie jest związane z fizycznym zestawianiem łącza pomiędzy nadawcą a odbiorcą - każdy pakiet jest kierowany niezależnie, a w ramach pojedynczego łącza (kanału transmisji) mogą być przekazywane pakiety adresowane do różnych odbiorców. Nazywane jest to komutacją pakietów, w odróżnieniu od komutacji łącza (która występowała np. w klasycznej, analogowej telefonii, gdzie przełączniki w centralach dokonywały zestawienia połączeń elektrycznych między dwoma aparatami telefonicznymi).

To właśnie umieszczanie w nagłówku każdego z pakietów adresów umożliwia kierowanie ruchem takich pakietów bez wcześniejszego fizycznego zestawiania łącz. W oparciu o ten adres host jest w stanie rozpoznać czy pakiet jest przeznaczony dla niego czy nie, a przełączniki i routery mogą kierować pakiety

do odpowiednich fragmentów sieci.

Jeżeli w sieci komputerowej host A chce się komunikować z hostem B, a host C z hostem D nie musimy zapewnić im osobnych łączy do przeprowadzania takiej komunikacji, tak jak to było na przykład w klasycznej telefonii analogowej, gdzie dwie takie "rozmowy" wymagałyby zestawienia osobnych fizycznych kabli od jednego abonenta do drugiego (odbywało się to za pomocą przekazników w centralach). W sieci komputerowej hosty te będą wytwarzały odpowiednio zaadresowane pakiety i reagowały tylko na pakiety zaadresowane do nich. Dzięki temu pakiety, stanowiące osobne strumienie komunikacji, mogą być z łatwością przesyłane tym samym fizycznym łączem. Host C może słyszeć lub nie komunikację pomiędzy hostami A i B (zależy to od różnych czynników, na przykład od wykorzystywanej sieci i względnego położenia tych hostów), natomiast wie że to "nie do niego".

Jeżeli w ramach sieci mamy jakieś urządzenie dzielące ją na mniejsze kawałki, podział ten będzie się odbywał w oparciu o adresy pakietów – urządzenie takie do danej sieci będzie przekazywało tylko pakiety adresowane do hostów w tej sieci. Mamy do czynienia z przełączaniem, czyli komutacją pakietów – w odróżnieniu od (omówionej pokrótce na przykładzie telefonii) komutacji łączy, która polegała na zestawianiu fizycznego łącza pomiędzy komunikującymi się hostami.

Protokoły sieciowe na ogół tworzą tak zwane stosy protokołów, gdzie dane opakowywane są kolejno w nagłówki kolejnych protokołów. Każdy z protokołów w takim stosie pełni dedykowane mu funkcje i na ogół nie ingeruje ani w protokoły warstwy niższej, ani w przenoszoną przez niego zawartość, czyli protokoły warstwy wyższej z danymi.

Komunikacja sieciowa typowo posiada strukturę warstwową. W modelu OSI wyróżnia się 7 warstw:

- fizyczną (pierwszą) definiującą aspekty związane z fizycznym przesyłem sygnału takie jak częstotliwości radiowe, poziomy napięcie, etc.; określa sposób transmisji kolejnych bajtów
- łącza danych (drugą) definiującą aspekty związane z formatem ramki, protokoły ustalania zasad dostępu do medium transmisyjnego, itd.; określa sposób transmisji porcji danych pomiędzy hostami w jednej sieci
- sieciową (trzecią) definiującą aspekty związane z formatem pakietu, adresacją i zasady routingu umożliwiające zapewnienie łączności pomiędzy różnymi sieciami; określa sposoby transmisji porcji danych pomiędzy sieciami
- transportową (czwartą) odpowiedzialną za podział strumienia na porcje informacji, kontrolę nad poprawnością transmisji, adresację usług w ramach hosta
- sesji (piątą)
- prezentacji (szóstą)
- aplikacji (siódmą)

W modelu TCP/IP wyróżnia się 4 warstwy:

- Dostępu do sieci - obejmującą warstwy 1 i 2 modelu OSI
- Internetu - obejmującą warstwę 3 modelu OSI
- Transportową - obejmującą warstwę 4 modelu OSI
- Aplikacji - obejmującą warstwy 5, 6 i 7 modelu OSI

Z punktu widzenia modelu TCP/IP można powiedzieć o enkapsulacji danych kolejnych warstw w ramach warstwy niższej, czyli "surowe" dane (np. strona HTML) obudowywane są strukturą opisywaną przez warstwę aplikacji (np. nagłówkami HTTP), następnie całość ta umieszczana jest w polu danych pakietu warstwy transportowej (np. TCP), ten z kolei w polu danych pakietu IP (warstwy sieciowej), na koniec pakiet IP jest umieszczany w polu danych ramki warstwy dostępu do sieci (np. ramki ethernetowej). W ramach podróży przez kolejne sieci pakiet IP jest wyjmowany i wkładany w kolejne ramki warstwy dostępu do sieci, na ogół tylko z niewielkimi ingerencjami w zawartość tego pakietu (prawie zawsze nie dochodzącymi do pola danych pakietu TCP lub datagramu UDP, czyli nie wykraczającymi poza warstwę 4 OSI).

Warstwa sprzętowa

Od strony sprzętowej sieć składa z:

- hostów stanowiących nadawców i odbiorców informacji
- urządzeń sieciowych pośredniczących w ich przekazywaniu, takich jak nadajniki, switchy, mediakonwertery
- okablowania miedzianego bądź światłowodowego (jeżeli nie jest siecią bezprzewodową)

Ethernet

W przypadku sieci w standardzie Ethernet stosowane są 48 bitowe adresy MAC (pierwsza część identyfikuje producenta karty) oraz wspólny dla wszystkich odmian (przewodowych i bezprzewodowych) format ramki (określający położenie w ramce adresów, informacji dodatkowych

oraz danych). Pakiety protokołu warstwy wyższej (np. pakiety IP wraz ich strukturą zawierającą adresy itd) z punktu widzenia ramki ethernetowej są danymi, w które ta warstwa nie wnika. Do mapowania adresów IP na adresy MAC wykorzystywany jest protokół ARP (dla IPv4) lub Neighbor Discovery (dla IPv6) - odbywa się to poprzez wysłanie ramki ethernetowej na adres rozgłoszeniowy (odbierany przez wszystkie hosty) z pytaniem o to jaki MAC adres ma host o podanym numerze IP.

Sieć ethernetowa typowo posiada strukturę wielokrotnej gwiazdy (drzewa), w węzłach której stosowane są switchy. Kierują one ramki do odpowiednich gałęzi na podstawie adresu docelowego i wpisów w tablicy adresów MAC, utworzonej w oparciu o źródłowe nadawców przechodzących przez dany switch ramek. W przypadku gdy adresu docelowego nie ma w tablicy ramka kierowana jest na wszystkie porty switcha z wyjątkiem tego na którym została odebrana. W taki sposób zawsze są też przesyłane ramki wysyłane na adres rozgłoszeniowy (broadcast).

Ethernet pozwala na virtualne podzielenie pojedynczej sieci lokalnej na wiele niezależnych (nie komunikujących się ze sobą w warstwie ethernetu) sieci, nazywanych VLAN. Działanie tego mechanizmu opiera się na zastosowaniu zarządzalnych switchy, które programowo mogą być dzielone na części zapewniające separację ruchu poszczególnych VLANów. Ponadto wybrane porty takiego switcha mogą być przypisane do różnych części (celem udostępnienia do innego switcha lub hosta kilku sieci wirtualnych), w takim przypadku do ramek ethernetowych wysyłanych tym portem dodawana jest informacja do którego VLANu należą (dwu bajtowy numer), a w przypadku ramek otrzymywanych na podstawie tego numeru odbywa się ich kierowanie do odpowiedniej "części" przełącznika (mówimy o VLANach tagowanych). Możliwe jest aby jeden wybrany VLAN na takim porcie był nie tagowany (do jego ramek nie będzie dodawany numer, a otrzymywane pakiety bez numeru będą kierowane do niego).

Ethernet pozwala również na grupowanie kilku portów w jeden port wirtualny (tzw port trunking / bonding) celem zwiększenia przepustowości lub niezawodności łącza. A dzięki zastosowaniu w różnych typach sieci ethernet tego samego formatu ramki możliwe jest też stosunkowo proste zmienianie medium transmisyjnego (np. z kabla miedzianego na światłowód) z użyciem media-konwerterów.

W przewodowych sieciach Ethernet wykrywaniem zajętości medium transmisyjnego oraz wykrywaniem kolizji zajmuje się protokół CSMA/CD (Carrier Sense Multiple Access with Collision Detection - wielodostęp z rozpoznawaniem stanu kanału oraz wykrywaniem kolizji). Przed rozpoczęciem nadawania stacja musi sprawdzić czy medium jest wolne, jeżeli tak może zacząć nadawać, jeżeli dwie stacje zaczną nadawać równocześnie zostaje to wykryte, obie przerywają nadawanie i wznowiają po losowym czasie. Jednak ze względu na stosowanie głównie połączeń punkt-punkt full-duplex (osobne przewody do nadawania i osobne do odbioru), co ogranicza tzw. domenę kolizji do pojedynczego hosta, protokół ten nie odgrywa tutaj szczególnie istotnej roli.

VLANy, bonding, ...

Ethernet pozwala na virtualne podzielenie pojedynczej sieci lokalnej na wiele niezależnych (nie komunikujących się ze sobą w warstwie ethernetu) sieci, nazywanych VLAN. Działanie tego mechanizmu opiera się na zastosowaniu zarządzalnych switchy, które programowo mogą być dzielone na części zapewniające separację ruchu poszczególnych VLANów. Ponadto wybrane porty takiego switcha mogą być przypisane do różnych części (celem udostępnienia do innego switcha lub hosta kilku sieci wirtualnych), w takim przypadku do ramek ethernetowych wysyłanych tym portem dodawana jest informacja do którego VLANu należą (dwu bajtowy numer), a w przypadku ramek otrzymywanych na podstawie tego numeru odbywa się ich kierowanie do odpowiedniej "części" przełącznika (mówimy o VLANach tagowanych). Możliwe jest aby jeden wybrany VLAN na takim porcie był nie tagowany (do jego ramek nie będzie dodawany numer, a otrzymywane pakiety bez numeru będą kierowane do niego).

Ethernet pozwala również na grupowanie kilku portów w jeden port wirtualny (tzw port trunking / bonding) celem zwiększenia przepustowości lub niezawodności łącza. A dzięki zastosowaniu w różnych typach sieci ethernet tego samego formatu ramki możliwe jest też stosunkowo proste zmienianie medium transmisyjnego (np. z kabla miedzianego na światłowód) z użyciem media-konwerterów.

Kable

Sieć ethernetowa wykorzystuje 8 żyłowe kable złożone z 4 par. Najpopularniejszym przewodem jest kabel UTP kategorii 5e, czyli nieekranowana skrętka pozwalająca na pracę z częstotliwością 100 MHz. W przypadku instalacji okablowania strukturalnego często stosowane są wyższe kategorie okablowania a także kable dodatkowo ekranowane. Ekran może obejmować osobno każdą parę, jak też może być wspólny dla całego przewodu, może być wykonany z folii lub siatki. Np. SF/FTP oznacza kabel z ekranem z siatki i folii (SF/), gdzie dodatkowo każda para jest ekranowana folią (FTP).

W ramach poszczególnych par realizowana jest transmisja różnicowa, czyli istotne jest napięcie pomiędzy przewodami w parze, a nie napięcie na danym przewodzie (w odniesieniu do jakiegoś zewnętrznego poziomu odniesienia). Standard 100Mb/s (dokładniej 100BASE-TX) wykorzystuje jedynie dwie pary przewodów, standard 1Gb/s (1000BASE-T) wykorzystuje wszystkie 4 pary przewodów. Długość kabla pomiędzy dwoma urządzeniami nie powinna przekraczać 100 m.

Wykorzystywanie skręconych par przewodów (jeden skręt na 6-10 cm kabla) ma na celu eliminację zakłóceń transmisji - (w uproszczeniu) zakłócenia wchodzą tak samo na oba przewody i różnica między nimi nie zmienia się.

Kable zakańczane są gniazdami bądź wtykami typu RJ-45 montowanymi według jednego z dwóch schematów kolorystycznych: EIA/TIA 568A lub 568B. Pierwotnie (dla sieci 100Mb/s lub starszych) użycie różnych standardów na obu końcach kabla służyło stworzeniu kabla skrosowanego. Połączenie takie przy jednakowych urządzeniach, gdzie nadajnik i odbiornik trafia zawsze na te same piny, zamieniało na kablu nadajnik z odbiornikiem, umożliwiając transmisje między nimi. Aktualnie zdecydowana większość urządzeń obsługuje protokół Auto MDI-X, który umożliwia automatyczne ustalenie na których pinach odbywa się nadawanie, a na których odbiór. W rzadkich przypadkach konieczne może być jednak zastosowanie kabla skrosowanego

Sieci IP

Protokół IP (Internet Protocol) odpowiedzialny jest przede wszystkim za sposób adresacji hostów oraz reguły komutacji pakietów (routing). Jest on wspomagany przez kolejny protokół z tej rodziny - ICMP (Internet Control Message Protocol), którego zadaniem jest przekazywanie informacji kontrolnych np. o nieosiągalności hosta docelowego, odrzuceniu przetwarzania pakietu ze względu na zbyt dużą liczbę skoków (gdy wartość pola TTL z nagłówka IP wyniesie zero) a także pingi (zarówno żądanie jak i odpowiedź).

Jako że IP stworzony jest do łączenia różnych fizycznych sieci w jedną sieć logiczną (internet), to pozwala on na wydzielanie w oparciu o zasady adresacji mniejszych fragmentów sieci, nazywanych niekiedy podsieciami. Oczywiście zapewnia też mechanizmy przekazywania pakietów pomiędzy takimi podsieciami.

Adresacja i routing

Adresy hostów (nazywane adresami IP) są to 32-bitowe (w IPv4) lub 128-bitowe (w IPv6) liczby.

Adresy IPv4 zapisywane są najczęściej w notacji kropkowo-dziesiętnej, gdzie każdy bajt (ciąg 8 bitów) zapisywany jest jako liczba dziesiętna rozdzielana kropką od pozostałych. Adresy IPv6 zapisywane są zazwyczaj w notacji dwukropkowej, polegającej na zapisywaniu 16 bitowych części adresu liczbami szesnastkowymi oddzielanymi dwukropkiem, dodatkowo jeden ciąg zer (o długości będącej wielokrotnością 16 bitów) może być skompresowany (pominięty) co daje w zapisie dwa dwukropki ::.

Długość prefixu i maska

Adresy hostów grupuje się w adresy sieci, bazując na jednakowym (bitowo) początku takiego adresu (zwanym adresem sieci lub prefixem). Ilość bitów stanowiących adres sieci w danym adresie IP nazywana jest długością prefixu i zapisywana jest zazwyczaj po ukośniku. Np. zapis 2001:db8::a17/48 oznacza że pierwsze 48 bity stanowią adres sieci a kolejne 128-48 = 80 bitów stanowi adres hosta w tej sieci.

Długość prefixu jednoznacznie określa maskę danej podsieci, czyli liczbę odpowiadającą długości adresu (32 bity lub 128 bitów), złożoną z ciągu jedynek o długości prefixu oraz ciągu zer (o długości adresu hosta). W przypadku IPv4 spotykane jest także podawanie maski sieci w notacji kropkowo-dziesiętnej zamiast długości prefixu.

Sieć może zostać podzielona na mniejsze sieci (z większą wartością prefixu), jak też grupa sieci może zostać zagregowana w jedną większą (2^n raza) sieć (z prefixem mniejszym o n). Agregacja hostów i sieci w większe całości jest wykorzystywana w mechanizmach routingu, co pozwala na redukcję wielkości tablic routingu.

Przynależność do sieci

Adres sieci zapisuje się typowo z wyzerowanymi bitami stanowiącymi adres hosta (czyli po dokonaniu bitowego *and* z maską danej sieci) oraz podaną informacją o długości prefixu, dla powyższego przykładu będzie to 2001:db8::/48. Informacja taka jest wystarczająca do sprawdzenia czy dowolny inny adres IP należy do tej sieci czy nie.

```
### Przykład Python
# adresacja IPv6

from ipaddress import *

a1 = IPv6Address("2001:0db8::17:15")
aa1 = int(a1)
print("adres IPv6 jest 128 bitową liczbą całkowitą np.: " + str(a1) + " == " + hex(aa1))

n0 = IPv6Network(":::/112");
m1 = n0.netmask
```

```

mm1 = int(m1)
p1 = n0.prefixlen

print("Maska podsieci IPv6 jest 128 bitową liczbą całkowitą np.: " + str(m1) + " == " + hex(mm1))
print("Jako że maska jest liczbą, która zapisana binarnie, zawsze zawiera ciągły ciąg bitów")
print("o wartości 1, a po nim ciągły ciąg bitów o wartości 0 (mogą być zerowej długości), to")
print("często stosowany jest zapis polegający na podawaniu długości prefiksu: /" + str(p1))
print("jest to ilość bitów o wartości 1 w masce, czyli im większy prefix tym mniejsza sieć.")

n1 = IPv6Network("2001:0db8::17:15/112", strict=False);
nn1 = int(n1.network_address)

print("Aby obliczyć adres sieci (czyli wspólną dla wszystkich hostów w danej sieci część")
print("adresu IP) należy wykonać binarny AND pomiędzy adresem IP hosta a maską podsieci.")
print("Dla powyższego przykładu:")
print(hex(mm1 & aal) + " == " + str(n1) + " == " + hex(nn1))

# aby sprawdzić czy adres IP należy do danej sieci trzeba obliczyć adres sieci tego hosta
# w oparciu o maskę sieci którą sprawdzamy
def sprawdzSiec(n, a):
    nn = int(a) & int(n.netmask)
    if nn == int(n.network_address):
        print(str(a) + " należy do sieci " + str(n))
    else:
        print(str(a) + " NIE należy do sieci " + str(n))

sprawdzSiec(n1, IPv6Address("2001:0db8::17:ab13"))
sprawdzSiec(n1, IPv6Address("2001:0db8::13:a"))

### Przykład Python
# adresacja IPv4

from ipaddress import *

a1 = IPv4Address("192.168.34.17")
aa1 = int(a1)
print("adres IPv4 jest 32 bitową liczbą całkowitą np.: " + str(a1) + " == " + hex(aa1))

n0 = IPv4Network("0.0.0.0/25");
m1 = n0.netmask
mm1 = int(m1)
p1 = n0.prefixlen

print("Maska podsieci IPv4 jest 32 bitową liczbą całkowitą np.: " + str(m1) + " == " + hex(mm1))
print("Jako że maska jest liczbą, która zapisana binarnie, zawsze zawiera ciągły ciąg bitów")
print("o wartości 1, a po nim ciągły ciąg bitów o wartości 0 (mogą być zerowej długości), to")
print("często stosowany jest zapis polegający na podawaniu długości prefiksu: /" + str(p1))
print("jest to ilość bitów o wartości 1 w masce, czyli im większy prefix tym mniejsza sieć.")

n1 = IPv4Network("192.168.34.17/25", strict=False);
nn1 = int(n1.network_address)

print("Aby obliczyć adres sieci (czyli wspólną dla wszystkich hostów w danej sieci część")
print("adresu IP) należy wykonać binarny AND pomiędzy adresem IP hosta a maską podsieci.")
print("Dla powyższego przykładu:")
print(hex(mm1 & aa1) + " == " + str(n1) + " == " + hex(nn1))

# aby sprawdzić czy adres IP należy do danej sieci trzeba obliczyć adres sieci tego hosta
# w oparciu o maskę sieci którą sprawdzamy
def sprawdzSiec(n, a):
    nn = int(a) & int(n.netmask)
    if nn == int(n.network_address):
        print(str(a) + " należy do sieci " + str(n))
    else:
        print(str(a) + " NIE należy do sieci " + str(n))

sprawdzSiec(n1, IPv4Address("192.168.34.13"))
sprawdzSiec(n1, IPv4Address("192.168.34.199"))

```

Routing

Router kieruje każdy z pakietów do kolejnego routera lub bezpośrednio do hosta docelowego na podstawie jego adresu docelowego i tablicy routingu. Tablica taka zawiera adresy sieci wraz z adresami następnych routerów do nich prowadzących bądź wskazaniem lokalnego interfejsu sieciowego poprzez który powinny być osiągalne hosty z danej sieci. W tym celu korzysta z sprawdzania przynależności adresu do sieci, w celu ustalenia adresu następnego routera i/lub

interfejsu sieciowego na który ma zostać przekazany pakiet.

Tablica przeglądana jest od wpisów najbardziej precyzyjnych, czyli z największym prefixem do wpisów najbardziej ogólnych (ostatnim wpisem jest na ogół trasa domyślna czyli sieć $::0$ dla IPv6 lub $0.0.0.0/0$ dla IPv4). Dzięki czemu jeżeli kilka wpisów (sieci) z tablicy routingu pasuje do adresu docelowego z nagłówka pakietu, wybierany jest wpis najbardziej precyzyjny (o najdłuższym prefixie), a pasująca do każdego adresu trasa domyślna wybierana jest tylko gdy nie ma żadnej lepszej. Może się zdarzyć że kilka wpisów (nawet z tą samą maską) pasuje do adresu docelowego hosta, w takiej sytuacji do wyboru ścieżki używane są inne dane z tablicy routingu (takie jak metryka).

Tablice routingu mogą zawierać wpisy dodawane statycznie (wpisane do konfiguracji danego urządzenia), jak też wpisy dodawane dynamicznie w oparciu o protokoły wymiany informacji routingowych (protokoły routingu) takie jak: IGRP, OSPF, BGP. Protokoły routingu dynamicznego mogą być wykorzystywane m.in. do rozkładania obciążenia na różne łącza, zapewnienia redundancji łącz, blokowania ataków (D)DoS.

Także każdy z hostów ma tablice routingu, typowo składa się z dwóch pozycji – trasy do sieci lokalnej (tej sieci z której adres posiada dany host) wskazującej bezpośrednio na urządzenie sieciowe oraz trasy domyślnej wskazującej na router zapewniający dostęp do innych sieci, nazywany bramką (gateway). Jeżeli router nie posiada adresu w tej samej sieci co host konieczna jest dodatkowa trasa wskazująca poprzez jakie urządzenie dostępny jest router domyślny.

Oprócz opisanego powyżej routingu unicastowego (kierowania do jednego odbiorcy) realizowane są także transmisje:

- *anycast* – do dowolnego / najbliższego hosta o danym adresie; zasadniczo jest to transmisja unicast, tyle że adres docelowy nie jest unikalny w skali globalnej a różne routery kieruje te pakiety do różnych hostów docelowych (typowo wybierając najbliższy taki host)
- *multicast* – do grupy hostów, w tym wypadku (multicastowy) adres IP identyfikuje "kanał nadawczy" a nie unikalny host docelowy
- *broadcast* – do wszystkich hostów (w ramach danej sieci – nie są routowne), transmisje rozgłoszeniowe można traktować jako szczególny przypadek transmisji multicastowych w których grupa multicastowa obejmuje wszystkie hosty (można je zastąpić takimi transmisjami multicastowymi)

Klient-serwer

W oparciu o protokół IP działają protokoły warstwy transportowej takie jak UDP, TCP, czy też (mniej znane protokoły czasu rzeczywistego, transmisji strumieniowych): RTP, RTCP i SCTP.

Jednym z zadań tych protokołów jest identyfikowanie usługi (procesu) w ramach systemu posiadającego dany adres IP, do którego mają trafić dane. W tym celu zarówno UDP jak i TCP na każdym z hostów wyróżniają numeryczny identyfikator dla aplikacji/procesu/usługi będącego odbiorcą czy też nadawcą informacji zwany `strong{numerem portu}`.

Najprostszym protokołem warstwy transmisji wydaje się być UDP, protokół ten umożliwia przesłanie informacji pomiędzy dwoma hostami IP i nie kontroluje on tego czy została ona przesłana poprawnie. Natomiast TCP kontroluje to czy przesłana informacja dotarła do adresata i nie została uszkodzona, a w przypadku problemów informacja wysyłana jest ponownie. TCP w związku z tym w przeciwieństwie do UDP musi utworzyć połączenie i wykorzystywać je do kontroli poprawności przesłania informacji, wymaga zatem przesłania większej liczby pakietów (co może prowadzić do pewnych opóźnień itp). W związku z tym TCP używany jest tam gdzie konieczna jest kontrola poprawności transmisji (oraz ponowne wysłanie zgubionego pakietu), UDP tam gdzie nie jest to potrzebne (a liczy się czas). Z połączeniem występującym w protokole TCP związane jest także pojęcie wielkości okna, czyli tego co ile (tysięcy) bajtów odbiorca musi potwierdzać odbiór pakietów. Wielkość ta jest dynamicznie dostosowywana do parametrów łącza, pozwalając na sterowanie przepływem - jeżeli wysycana jest dostępna przepustowość łącza, dochodzi do strat pakietów i wielkość okna jest zmniejszana.

RTP często jest zaliczany do warstwy transportowej gdyż działa poniżej typowych protokołów warstwy aplikacji, jednak dane które opisują nie są informacjami systemowymi/sieciowymi a właśnie aplikacyjnymi, ponadto działa on powyżej protokołów transportowych (zazwyczaj nad UDP). Może przenosić w jednej sesji kilka strumieni, ale zazwyczaj dla każdego strumienia tworzona jest osobna sesja (zestaw portów). Protokół ten umożliwia identyfikację zawartości pakietu, identyfikację kolejności pakietów w strumieniu. Pozwala także na przepuszczanie ruchu przez mostki maskujące oryginalne adresy, zmieniające medium, replikujące strumień (replikowany unicast, który wydaje się być bardzo dobrym rozwiązaniem gdy z jakiś powodów multicast nie może zadziałać), czy też nawet łączące strumienie z kilku źródeł. Współpracując z nim RTCP służy do przesyłania raportów z informacjami o stratach, opóźnieniach itp oraz wzajemnej synchronizacji mediów pomiędzy strumieniami - np. synchronizacji audio z wideo).

Popularne usługi

W ramach sieci mogą być realizowane różne usługi w oparciu o różne protokoły warstwy aplikacyjnej. Standardowe usługi posiadają zdefiniowane domyślne adresy portów dla swoich protokołów. Wśród usług i protokołów sieciowych należy wymienić przynajmniej:

- DNS (Domain Name System) - odpowiedzialny za system mapujący nazwy alfanumeryczne hostów na adresy IP.
- mechanizmy auto konfiguracji hostów - DHCP, rozgłaszanie informacji routingowej poprzez ICMPv6 (protokół warstwy 3)
- WWW - udostępnianie treści z użyciem protokołu HTTP
- pocztę elektroniczną - przesyłanie wiadomości (protokoły SMTP, IMAP, POP)
- komunikację natychmiastową i telefonię IP (protokoły SIP, XMPP, IAX)
- SSH - zdalny, szyfrowany dostęp do systemów IT, przesył plików oraz tunelowanie innych usług

Na uwagę zasługuje fakt że większość z wymienionych protokołów warstwy aplikacyjnej to protokoły tekstowe (istotne wyjątki stanowią DNS, DHCP i SSH).

Domain Name System

DNS umożliwia mapowanie nazwy na adres IP (lub wiele adresów IP) oraz przechowywanie dodatkowych informacji na temat domeny i znajdujących się w niej usług. Domeny posiadają budowę hierarchiczną / drzewiastą:

- precyzja rośnie od prawej do lewej
- kolejne poziomy oddzielane są kropkami
- najwyższym poziomem jest kropka będąca ostatnim znakiem w pełnej nazwie domenowej (np. ciekawi.icm.edu.pl.), którą najczęściej pomija się w zapisie
- hierarchia ta jest niezależna od hierarchii routingu i wynika z faktu posiadania/użytkowania danej (pod)domeny

Realizacja odpowiedzi na zapytanie DNS wygląda następująco:

1. host kieruje zapytanie do określonego w jego konfiguracji serwera "rozwiązającego" DNS (DNS resolver),

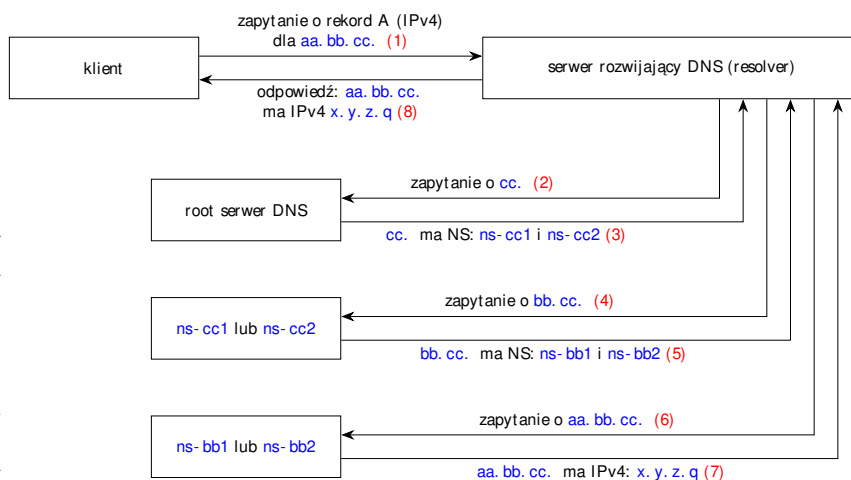
2. serwer taki sprawdza w swojej pamięci podręcznej czy zna odpowiedź na to zapytanie (i nie jest ona przeterminowana - nie upłynął czas TTL od odnalezienia), jeżeli nie ma jej w swojej pamięci to

3. serwer taki zna adresy głównych serwerów DNS (root serwerów)

zawierających informacje na temat serwerów obsługujących domeny najwyższego rzędu i kieruje do jednego z nich zapytanie o serwer obsługujący skrajnie prawą część adresu (np. .org),

4. do otrzymanego serwera kierowane jest zapytanie o większą część adresu (np. eu.org),

5. itd. aż do uzyskania odpowiedzi o pytany adres



(X) oznacza kolejność wykonywanych operacji. Przed wykonaniem zapytania resolver sprawdza czy nie posiada w swoim cache zapamiętanej (i nie przeterminowanej) odpowiedzi na to zapytanie.

DNS przechowuje informacje w postaci rekordów mających określony typ (w większości przypadków dla danej nazwy domenowej może być zdefiniowanych wiele rekordów, tego samego lub innych typów). Wśród najważniejszych typów rekordów należy wymienić:

- NS - informacja o serwerach obsługujących DNS danej domeny
- A - mapowanie nazwy na adres IPv4
- AAAA - mapowanie nazwy na adres IPv6
- MX - informacja o serwerach obsługujących pocztę danej domeny
- SRV - informacje o hoście świadczącym usługę (usługa określana jest w nazwie domeny o którą pytamy)

- PTR – mapowanie adresów IP na nazwy domenowe, realizowane w specjalnym drzewie in-addr.arpa (dla IPv4) lub ip6.arpa (IPv6), gdzie adres IP zapisywany jest w odwróconej kolejności po bajcie dla IPv4 lub cyfrze szesnastkowej dla IPv6 (zobacz wynik polecenia host z adresem IPv4 i IPv6)
- TXT – informacje dodatkowe (np. jakie serwery pocztowe, są upoważnione do wysyłania poczty z tej domeny)
- SOA – informacje podstawowe o strefie opisującej domenę
- CNAME – alias na inną domenę (domena którą aliasujemy nie może mieć innych wpisów, nawet SOA)

Translacja adresów

Routery oprócz zwykłego kierowania pakietów na odpowiednie łącze, mogą także modyfikować adresy IP i numery portów. Mechanizm ten określany jest mianem translacji adresów sieciowych (Network Address Translation). Modyfikowaniu mogą ulegać zarówno źródłowe (SNAT) jak i docelowe (DNAT) adresy IP, jak też numery portów protokołów warstwy transportowej (takich jak TCP, czy UDP).

Translacja może odbywać się w oparciu o stałe reguły (dany adres, port lub zestaw adres+port zawsze mapowany jest na takie same wartości) jak też w sposób dynamiczny, gdzie mapowanie odbywa się np. na grupę adresów lub pojedynczy adres i dynamicznie dobierane porty. Dynamiczne mapowanie wymaga od routera śledzenia przechodzących przez niego połączeń i ich stanu, celem umożliwienia przekazania pakietu z odpowiedzią oraz zwolnienia mapowania po zakończeniu połączenia TCP.

Filtracja pakietów

Względy bezpieczeństwa często wymuszają filtrację ruchu sieciowego docierającego do danego hosta lub przechodzącego przez router. Filtracja taka może odbywać się zarówno w oparciu o dane z nagłówków IP (np. adresy źródłowe i docelowe), dane z nagłówków warstwy transportowej (np. numery portów TCP lub UDP), jak również informację z systemu śledzenia połączeń czy też zawartość samego pakietu.

Tunelowanie i VPN

Możliwa jest enkapsulacja pakietów IP w pakietach innych protokołów (w szczególności także IP). Pozwala to na realizację różnego rodzaju tuneli IP. Przykładem takiego tunelu jest VPN zapewniający szyfrowany dostęp (w warstwie trzeciej - IP lub nawet w warstwie drugiej - LAN) do jakiejś innej sieci.

Multicast

Multicast jest transmisją jeden do wielu, pozwalającą na uniknięcie powielania identycznych pakietów (różniących się tylko adresem odbiorcy). Metoda ta jest użyteczna np. przy streamingu audio-video na żywo. Multicastowy adres IP identyfikuje dynamiczną (mogącą zmieniać się w czasie) grupę odbiorców i jest on przydzielany danej transmisji przez jej nadawcę, a nie przyznawany administracyjnie konkretnemu odbiorcy. W związku z tym często wszystkie pakiety adresowane do danej grupy multicastowej (mające taki sam multicastowy adres docelowy) pochodzą od tego samego hosta (a więc mają taki sam unicastowy adres źródłowy). Zależność ta została wykorzystana w *source-specific multicast*, który eliminuje problem przydzielania adresów multicastowych nadawcom – dowolnego adresu multicastowego z puli przeznaczonej na potrzeby source-specific multicast może użyć każdy, a kanał multicastowy (grupa odbiorców) identyfikowana jest w oparciu o dwa adresy (unicastowy nadawcy i multicastowy odbiorcy).

W celu obsłużenia transmisji multicastowych router musi posiadać informacje o aktualnie "zasubskrybowanych" przez jego klientów transmisjach multicastowych. Posiadając takie dane (czyli adres multicastowy na który kierowane są pakiety danej transmisji i ewentualnie unicastowy adres źródłowy z którego są wysyłane) może on pozyskać stosowne transmisje od routerów z którymi jest połączony i przekazać je swoim klientom. Do poinformowania routera o żądaniu dostarczenia transmisji multicastowej (a także do przekazania informacji o zakończeniu odbioru takiej transmisji) hosty wykorzystują protokół Internet Group Management Protocol (dla IPv4) lub Multicast Listener Discovery (dla IPv6). Celem obsługi zaprzestania transmisji gdy host nie poinformował o zakończeniu odbioru, router z wykorzystaniem tych protokołów co pewien czas zadaje pytanie czy nadal są hosty zainteresowane daną transmisją.

Quality of Service

QoS są to techniki gwarantujące jakość usługi. Najczęściej stosowaną techniką zaliczaną do QoS jest dynamiczny podział przepustowości łącza z wykorzystaniem algorytmów takich jak: HTB (Hierarchical Token Bucket), CBQ (Class Based Queueing). Nie jest to jednak prawdziwy QoS, gdyż mechanizmy te to coś więcej niż tylko prosty podział pasma. O QoS'ie możemy zaczynać mówić gdy system kolejowania uwzględnia priorytety dla pewnych typów ruchu (m.in. na podstawie pola ToS - Type of service). Kolejki priorytetowe mogą być realizowane jako wagowe - cyklicznie z każdej klasy wysyłamy proporcjonalnie do wagi, lub bez-wagowy (klasa ważniejsza może zagłodzić klasy o mniejszym priorytecie, gdyż brany jest z ważniejszej dopóki są lub nie pojawią się jeszcze ważniejsze). Istnieją

dwie podstawowe architektury systemów QoS:

- DiffServ - nie ma ścisłych gwarancji dla jakiegokolwiek strumienia, na brzegu użytkownik negocjuje SLA i ruch jest kontrolowany pod względem tego SLA (m.in. ustawianie i kontrola pola ToS). Może występować także zarządca pasma (Bandwidth Broker), z którym klienci komunikują się przy pomocy protokołu RSVP (Resource Reservation Protocol) i negocjują z nim kontrakty SLA, natomiast sam BB dokonuje rezerwacji zasobów w routerach i kontroluje ich stan (zazwyczaj z wykorzystaniem protokołu Common Open Policy Service).
- IntServ - odbiorca w oparciu o informacje uzyskane od nadawcy na temat parametrów transmisji dokonuje rezerwacji zasobów w wszystkich routerach na ścieżce łączącej go z nadawcą (wykorzystuje do tego protokół RSVP), każdy z routerów decyduje czy może spełnić te wymagania (gdy jeden nie może cała operacja nie udaje się). Każdy strumień negocjowany jest indywidualnie, poprzez mechanizm Admission Control (podawane są specyfikacje filtru - jak identyfikować ten ruch, specyfikacje przepływu, itd), ponadto informacje o rezerwacji muszą być okresowo odnawiane. Wyróżnia się klasy ruchu: Best Effort (standardowa), Guaranteed Service (gwarancja dostępności pasma i opóźnień), Control Load Service (zapewniamy mało obciążony kanał, ale bez gwarancji).

Systemy operacyjne

Co robi system operacyjny?

System operacyjny jest oprogramowaniem odpowiedzialnym za zarządzanie zasobami systemu komputerowego (sprzętem, ale nie tylko) oraz uruchomionymi na nim aplikacjami. Do najistotniejszych zadań systemu operacyjnego zalicza się podział czasu procesora i szeregowanie zadań oraz zarządzanie pamięcią - w szczególności obsługa pamięci wirtualnej, najczęściej z wykorzystaniem mechanizmu stronicowania.

Oprócz tego system zajmuje się także zarządzaniem plikami, wejściem/wyjściem (najczęściej jest ono realizowane w oparciu o przerwania (IRQ), ale znane są także modele programowego we/wy polegającego na aktywnym czekaniu), obsługą urządzeń (wejście/wyjście, sterowniki, dostęp), obsługą sieci (stos protokołów sieciowych), itd. Część zadań realizowana jest z minimalnym udziałem procesora (a więc także i systemu) jest to na przykład transfer danych w trybie DMA polegający na tym iż dane kopiowane są całymi blokami bez udziału procesora do/z pamięci (system zajmuje się tylko inicjacją transmisji). Należy tu jednocześnie zaznaczyć iż w przypadku nie stosowania tej technologii dane też kopiowane są pomiędzy dyskiem a procesorem całymi blokami (minimum sektor) gdyż dysk (w odróżnieniu od pamięci operacyjnej) nie jest bezpośrednio dostępny dla procesora.

Współczesne systemy korzystają z co najmniej dwóch poziomów pracy - uprzywilejowanego poziomu "nadzorcy" w którym działa jądro systemu operacyjnego oraz trybu użytkownika. Operacje I/O muszą odbywać się w trybie uprzywilejowanym. Również pamięć posiada obszar chroniony, w którym umieszczany jest m.in. tablica wektorów przerwań (inaczej zwołań) systemowych zapewniających doprowadzić do przejścia systemu w trybie uprzywilejowanym).

Procesy i szeregowanie zadań

Istotną rolą systemu operacyjnego w zarządzaniu procesami (obok czynności administracyjnych jak ich tworzenie powielanie, usuwanie, czy też wstrzymywanie itp) jest zapewnienie ochrony pamięci (każdy proces może pisać po swojej i ewentualnie współdzielonej gdy dostał do tego prawo) oraz procesora (przerwanie zegarowe powoduje wywołanie planisty, który ustala jaki proces dostanie następny kwant czasu procesora). Niektóre systemy wyróżniają obok procesów także wątki, które różnią się od nich współdzieloną (między wątkami jednego procesu) pamięcią i zasobami (np. otwartymi plikami). System operacyjny zapewnia także zestaw usług i funkcji (wywołań) systemowych zapewniających pośrednictwo między interfejsem trybu użytkownika a sprzętem.

Istotnym zadaniem systemu operacyjnego jest przeciwdziałaniem tzw. blokadom, czyli sytuacji gdy dwa lub więcej procesów blokują się wzajemnie w oczekiwaniu na zasoby (a ma zasób X, którego potrzebuje aby zwolnić zasób Y, którego potrzebuje a do zwolnienia X). Realizowane to może być na kilka sposobów:

- zapobieganie blokadzie (czyli niedopuszczenie do zajęcia warunków koniecznych) - np. poprzez konieczność deklarowania wszystkich zasobów na początku, zwalniania przydzielonych zasobów przed zgłoszeniem zapotrzebowania na następne
- unikanie blokady (czyli określamy maksymalne zapotrzebowanie i tak przydzielamy zasoby aby uniknąć zajęcia blokady) - np. poprzez kontrolę czy po spełnieniu żądania dalej będziemy działać w stanie "bezpiecznym", tj takim że istnieje sekwencja (zwana bezpieczną) w której maksymalne zapotrzebowanie każdego procesu może być spełnione w oparciu o zasoby zwolnione przez procesy będące wcześniej w tej sekwencji oraz zasoby wolne
- wykrywanie i usuwanie blokady gdy do niej doszło

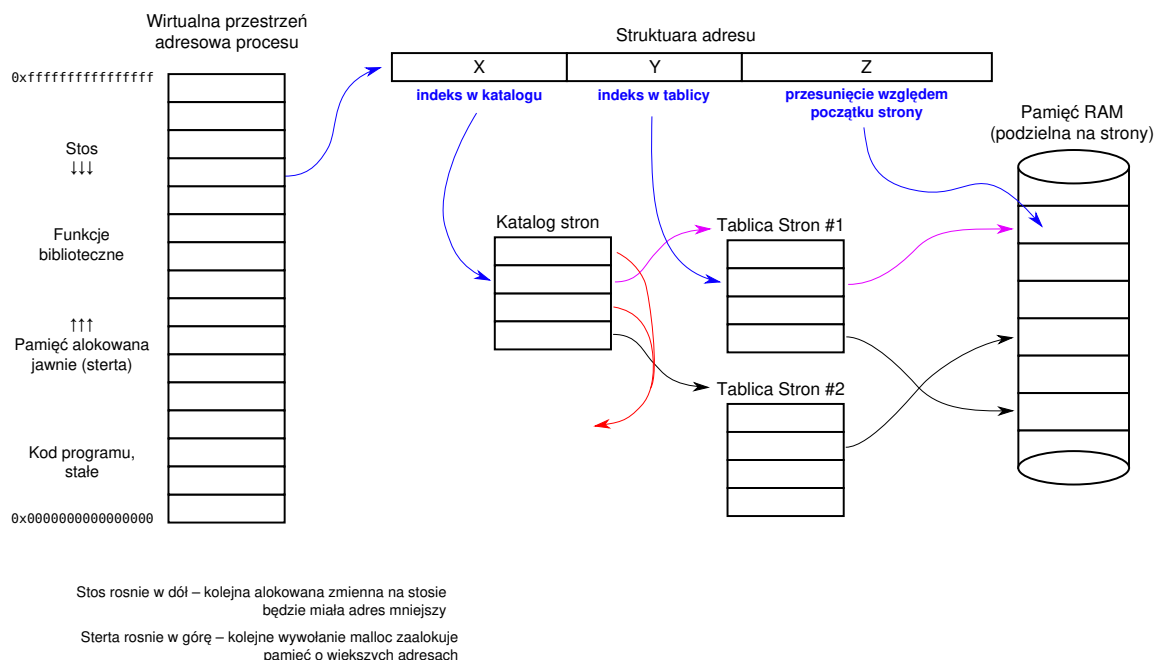
Planista procesora, czyli fragment systemu odpowiedzialny za przydzielanie procesora procesom, może

pracować w trybie z wywłaszczaniem lub bez. W tym pierwszym wypadku proces otrzymuje kwant czasu procesora który może wykorzystać w całości (wtedy przejdzie z stanu wykonywania w stan gotowości) lub z niego wcześniej zrezygnować (gdy np. czeka na I/O, wtedy przejdzie z stanu wykonywania w stan oczekiwania). W drugim przypadku proces wykonuje swój kod do momentu aż sam odda procesor. Forma ta zbliżona jest do wykorzystywanej w szeregowaniu czasu rzeczywistego - proces będzie wywłaszczony tylko przez proces o wyższym priorytecie i będzie to natychmiastowe (przy najbliższym przerwaniu zegarowym). Istnieje wiele algorytmów szeregowania takich jak:

- FCFS - pierwszy zgłoszony = pierwszy obsłużony
- SJF - najkrótszy zgłoszony będzie pierwszym wykonanym (wersja z wywłaszczaniem - SRTF - gdy nowy najkrótszy pozostały), algorytm raczej nie do zastosowania praktycznego - trzeba by przewidywać długość wykonania
- priorytetowe - zawsze o najwyższym priorytecie (jak wspomniałem wyżej wykorzystywane w systemach real-time
- rotacyjne - każdy po kawałku, potem na koniec kolejki
- kolejki wielopoziomowe - system z priorytetami, podziałem czasu pomiędzy kolejki, przenoszeniem procesów między kolejkami, ...

Zarządzanie pamięcią

Drugą podstawową funkcją systemu operacyjnego, wspomnianą na początku, jest zarządzanie pamięcią. Zarządzanie pamięcią polega na odpowiednim mapowaniu adresów logicznych (używanych przez procesy) na adresy fizyczne (używane przez procesor), korzysta ono z wsparcia sprzętowego ze strony procesora. Jest to najczęściej realizowane w oparciu o wspomniany mechanizm stronicowania. Polega to na podziale pamięci dostępnej pamięci fizycznej na jednakowe bloki zwane ramkami oraz podziale pamięci logicznej na jednakowe bloki (o tej całej wielkości co ramki) zwane stronami. Strony które są wykorzystywane przez program są mapowane na dowolne ramki pamięci fizycznej (w przypadku gdy dana strona nie zamapowana - w zależności od okoliczności błąd braku strony lub błąd ochrony strony). Rozwiązuje to problem fragmentacji zewnętrznej, polegającej na braku spójnego obszaru pamięci o żądanej długości pomimo iż łączna ilość wolnej pamięci jest dostateczna, jednak nie rozwiązuje problemu fragmentacji wewnętrznej, polegającej na przydzielaniu zbyt dużych fragmentów pamięci dla procesu (a wręcz można powiedzieć że go pogłębia). Mechanizm ten wymaga trzymania tablicy wolnych ramek, tablicy stron dla każdego procesu (zawierającej przypisania mapowań stron danego procesu na ramki) oraz wykonywania tłumaczenia adresów logicznych (strona + przesunięcie na stronie). Także sama tablica stron procesu może być stronicowana (mamy tablicę która informuje nas że przypisania stron w danym zakresie adresów są przechowywane w jakiejś ramce).



Strony i ramki mogą być współdzielone pomiędzy procesami (np. przy rozgałęzianiu procesu strony są kopiowane dopiero gdy zajdzie taka potrzeba). W przypadku braku miejsca w pamięci fizycznej wybrane strony nieaktywnego aktualnie procesu mogą być umieszczane na dysku (swap). Niekiedy może to powodować szamotanie procesu polegające na zbyt dużej liczbie wymian stron. Zawsze jednak prowadzi to do konieczności ustalania które strony najlepiej jest przenieść na dysk. Optymalne byłoby przenoszenie tych które najdłużej nie będą potrzebne (jednak z oczywistych względów jest to

praktycznie nie do zrealizowania). Stosuje się różne algorytmy tego wyboru:

- FIFO - usuwamy najdłużej będącą w pamięci
- LRU - usuwanie tej do której najdawniej się odwoływano (licznik czasu, bit odniesienia, bit odniesienia w określonym czasie, bit modyfikacji)
- LFU - usuwamy z najmniejszą liczbą odwołań
- MFU - usuwamy z dużą liczbą odwołań

Alternatywną (i mniej wymagającą) wobec stronicowania metodą zarządzania pamięcią jest segmentacja. W przypadku architektury x86 jest ona zawsze wykorzystywana jednak może być przykryta dużym segmentem na którym wykorzystujemy stronicowaniem.

Proces uruchamiania komputera

Po otrzymaniu sygnału resetu (także przy uruchamianiu systemu - "Power-on Reset") procesor po inicjalizacji rejestrów zaczyna wykonywanie kodu znajdującego się pod jakimś ustalonym adresem (typowo w wbudowanej lub zewnętrznej pamięci typu ROM lub Flash). W zależności od danej architektury / procesora może to być m.in.: bezpośrednio kod programu użytkownika, wbudowany bootloader danego procesora umożliwiający dalsze ładowanie np. z karty SD, zewnętrzny niskopoziomowy bootloader (np. u-boot).

W przypadku architektur zgodnych z x86 jest to BIOS, który po zakończeniu procesu inicjalizacji sprzętu i testów rozruchowych ładuje do pamięci kod znajdujący się w pierwszym sektorze dysku twardego (sektorze rozruchowym rozpoczynającym się od adresu zerowego) i uruchamia go (przekazuje do niego kontrolę). Znajduje się tam kod (lub tylko początek kodu) programu rozruchowego, którego zadaniem jest załadowanie systemu operacyjnego. W przypadku współczesnych systemów linuxowych jest to zazwyczaj GRUB.

W przypadku sektorów rozruchowych typu MBR (i kompatybilnych z nim), kod ten może liczyć maksymalnie 446 bajtów (gdyż na kolejnych pozycjach znajduje się tablica partycji) i jego zadaniem jest załadowanie i uruchomienie pozostałej części programu rozruchowego (musi znać jej położenie na dysku). Pozostała część programu rozruchowego może znajdować się tuż za MBR (w przerwie pomiędzy MBR a pierwszą partycją - dla tablicy partycji MBR/msdos), w dedykowanej partycji BIOS boot partition (dla tablicy partycji GPT) lub w partycji oznaczonej jako bootowalna. Część ta zawiera moduły pozwalające na dostęp do systemu plików zawierającego konfigurację, obraz jądra, itp oraz informację o jego położeniu (dysk, partycja, ścieżka).

W przypadku komputerów opartych na UEFI firmware odpowiedzialny jest za zinterpretowanie tablicy partycji (GPT) i załadowanie programu rozruchowego z pliku znajdującego się na specjalnej partycji EFI (EFI System partition) z systemem plików FAT32. W pliku tym umieszczana jest całość (obie opisane powyżej części programu rozruchowego).

Start systemu rozpoczyna się od załadowania do pamięci obrazu jądra wraz z parametrami oraz (opcjonalnie) initrd i przekazania kontroli do jądra przez program rozruchowy (np. GRUB). W przypadku jądra linuxowego i korzystania z initrd obraz ten przekształcany jest na RAM-dysk w trybie zapisu-odczytu i montowany jako rootfs z którego uruchamiany jest /sbin/init (którego podstawowym zadaniem jest zamontowanie właściwego rootfs). Po jego zakończeniu (lub od razu gdy nie używamy initrd) uruchamiany jest program wskazany w opcji init= jądra (domyślnie typowo /sbin/init) z rootfs wskazanego w opcji root= jądra. W opcji init= można wskazać dowolny program lub skrypt (uruchomiony zostanie z prawami root'a).

Systemy "unix-owate"

Jest to grupa systemów operacyjnych, będących (przynajmniej w jakimś stopniu) zgodnych z specyfikacją POSIX / Single UNIX Specification. Charakteryzują się zbliżonym zbiorem wywołań systemowych dostępnych z poziomu języka C oraz funkcji biblioteki standardowej C, co ułatwia zapewnienie przenośności oprogramowania (w postaci źródłowej). Zapewniają podobny zbiór podstawowych komend linii poleceń (konsoli).

Systemy unix'owe posiadają drzewiasty system plików zaczynający się w katalogu głównym oznaczanym przez ukośnik (/), w którym zamontowany jest główny system plików (rootfs), inne systemy plików mogą być montowane w kolejnych katalogach. Do najistotniejszych katalogów należy zaliczyć:

- /bin
zawierający pliki wykonywalne podstawowych programów
- /sbin
zawierający pliki wykonywalne podstawowych programów administracyjnych
- /lib
zawierający pliki podstawowych bibliotek
- /usr
zawierający oprogramowanie dodatkowe (wewnętrznie ma podobną strukturę do głównego - tzn.

katalogi */usr/bin*, */usr/sbin*, */usr/lib*, itd)

/etc
zawierający konfiguracje ogólnosystemowe

/var
zawierający dane programów i usług (takie jak kolejka poczty, harmonogramy zadań, bazy danych)

/home
zawierający katalogi domowe użytkowników (często montowany z innego systemu plików, dlatego też root ma swój katalog domowy w */root*, aby był dostępny nawet gdy takie montowanie nie doszło do skutku)

/tmp
zawierający pliki tymczasowe (typowo czyszczony przy starcie systemu); w Linuxie występuje też */run* przeznaczony do trzymania danych tymczasowych działających usług takich jak numery pid, blokady, itp

/dev
zawierający pliki reprezentujące urządzenia; w Linuxie występuje też */sys* zawierający informacje i ustawienia dotyczące m.in. urządzeń

/proc
zawierający informacje o działających procesach (w Linuxie także interfejs konfiguracyjny dla wielu parametrów jądra)

Pliki i katalogi których nazwa rozpoczyna się od kropki traktowane są jako pliki ukryte.

Z punktu widzenia programisty czy też użytkownika (prawie) wszystko jest plikiem, których istnieją różne rodzaje (zwykły plik, katalog, urządzenie znakowe, urządzenie blokowe, link symboliczny, kolejka FIFO, ...); pewnym wyjątkiem są urządzenia sieciowe (które nie mają reprezentacji w systemie plików (ale gniazda związane z nawiązanymi połączeniami obsługuje się zasadniczo tak jak pliki).

Polecenia

Unixowe komendy (czyli polecenia rozumiane przez bash lub inny interpreter zgodny z sh) składają się z nazwy polecenia oraz opcji i argumentów. Nazwą polecenia może być nazwa funkcji wbudowanej, nazwa programu (znajdującego się w ścieżce wyszukiwania programów) lub pełna ścieżka do programu. Po nazwie polecenia mogą występować opcje i/lub argumenty. Są one oddzielane od nazwy polecenia i od siebie przy pomocy spacji (zasadniczo dowolnego ciągu białych znaków: spacji, tabulatorów, escapowanych nowych linii). Nie ma silnego rozróżnienia opcji od argumentów, typowo stosowaną konwencją jest rozpoczynanie opcji od pojedynczego myślnika (opcje krótkie - jednoliterowe) lub dwóch myślników (opcje długie). W przypadku stosowania tej konwencji po pojedynczym myślniku może występować kilka bezargumentowych opcji jednoliterowych. Typowo argumenty opcji oddzielane są od nich spacją (w przypadku opcji krótkich) lub znakiem równości (w przypadku opcji długich). Jeżeli któryś z składników komendy (np. argument) zawiera spację należy je zabezpieczyć przy pomocy odwrotnego ukośnika lub ujęcia zawierającego je napisu w apostrofy lub cudzysłowa.

Przekierowania

Typowo program posiada trzy strumienie danych: jeden wejściowy (stdin) i dwa wyjściowe (stdout i stderr). Standardowe wyjście możemy przekierować na standardowe wejście innego programu przy pomocy `|`, np: `ls --help | less`. Konstrukcja ta przekieruje wynik komendy `ls` uruchomionej z opcją `--help` do komendy `less`.

Możemy także przekierować standardowe wyjście do pliku (przy pomocy `>` lub `>>`, gdy chcemy dopisywać do pliku) lub pobrać standardowe wejście z pliku (przy pomocy `<`). `>` pozwala na przekierowanie standardowego wyjścia błędu do pliku.

Jeżeli zachodzi potrzeba połączenia obu strumieni możemy użyć `>&1` w celu przekierowania strumienia drugiego do pierwszego. Następnie możemy użyć `|` aby przekierować połączony strumień do następnej komendy. Jeżeli chcemy przekierować go do pliku połączenie strumieni powinno mieć miejsce po przekierowaniu pierwszego z nich do pliku, np.: `ls . NieIstniejącyPlik >log.txt 2>&1`. Bash pozwala użyć `>&1` i `&2`, które przekierowują oba strumienie odpowiednio do pliku lub standardowego wejścia innego polecenia, ale jest to rozszerzenie wykraczające poza standardową składnię sh.

Kod powrotu polecenia oraz łączenie poleceń

Każde uruchamiane polecenie po zakończeniu działania zwraca liczbowy kod powrotu (w przypadku programów w C jest to wartość zwracana z funkcji `main`). Zero oznacza że polecenie zakończyło się sukcesem (np. znaleziono szukane pliki), wartość nie zerowa że zakończyło się porażką (np. nie ma pasujących plików) lub błędem (np. składnia wprowadzonego polecenia była niepoprawna).

Polecenia mogą być łączone na różne sposoby – z wykorzystaniem tej informacji lub nie:

- `a && b` – polecenie `b` wykona się gdy `a` zakończyło się sukcesem (zwróciło kod 0)

- `a || b` – polecenie `b` wykona się gdy `a` zakończyło się porażką lub błędem (zwróciło kod różny od 0)
- `a ; b` – polecenie `b` po zakończeniu polecenia `a` (bez względu na jego kod powrotu)
- `a & b` – polecenie `b` będzie wykonywane równocześnie z `a` (dokładniej polecenie `a` zostanie uruchomione w tle, a na terminalu zajmie polecenie `b`)

Spacje w powyższych konstrukcjach są opcjonalne. Średnik i pojedynczy `&` mogą być dodane do polecenia także gdy nie ma kolejnego w ciągu:

- `a&` uruchomi polecenie `a` w tle i odda linię poleceń,
- `a;` uruchomi polecenie `a` (dokładnie tak samo jakby nie było tego średnika).

Najważniejsze polecenia

Wiele z poniższych komend stanowi standardowe polecenia określone w POSIX / Single UNIX Specification, co gwarantuje ich obecność na kompatybilnych systemach (jednak nie jest to pełna lista poleceń wymaganych przez POSIX). Część z nich jest powszechnie dostępnym, wieloplatformowym oprogramowaniem dla systemów "unix-owatych". Niektóre (zwłaszcza związane z działaniami administracyjnymi) są specyficzne dla Debiana lub Linuxa.

Uzyskiwanie pomocy

```
man [nazwa polecenia]
    wyświetla stronę podręcznika systemowego na temat wskazanego polecenia, funkcji systemowej,
    pliku konfiguracyjnego

pinfo [nazwa polecenia] lub info [nazwa polecenia]
    wyświetla stronę nowego podręcznika systemowego na temat wskazanego polecenia

apropos [tekst]
    wyszukiwanie w opisach stron podręcznika systemowego
    apropos -s 1 '' pozwala zapoznać się z spisem wszystkich stron w rozdziale pierwszym (co jest w
    którym rozdziale można zobaczyć w man man)
```

Większość poleceń obsługuje także opcje `--help` lub `-h`, które wyświetlają informację na temat ich użycia. W tym miejscu warto też zauważyć że typowo opcje krótkie (pojedyncza litera) poprzedzane są pojedynczym myślnikiem (i po jednym myślniku może wystąpić kilka kolejnych opcji), natomiast opcje długie poprzedzane są dwoma myślnikami.

Zarówno w tekstach pomocy jak i w tym dokumencie stosowana jest konwencja polegająca na oznaczaniu opcjonalnych argumentów poprzez umieszczanie ich w nawiasach kwadratowych (jeżeli podajemy ten argument do komendy nie obejmujemy go już tymi nawiasami) oraz rozdzielaniu alternatywnych opcji przy pomocy `~|`. Np. `a [b] c|d` oznacza iż polecenie `a` wymaga argumentu postaci `c` albo `d`, który może być poprzedzony argumentem `b`.

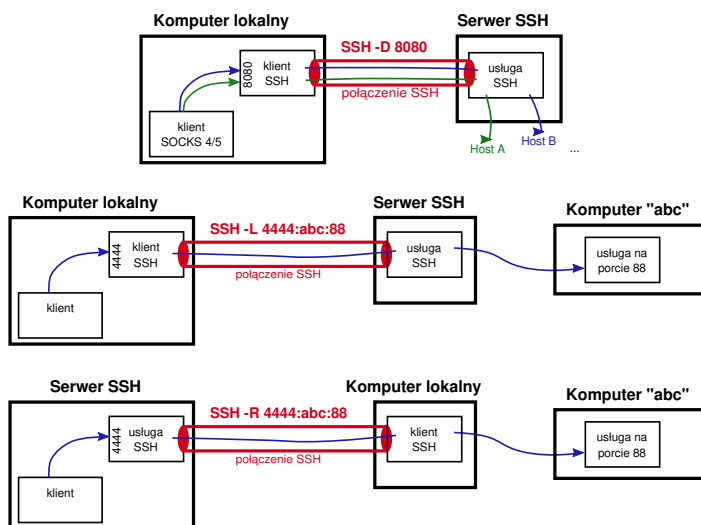
Powłoka (i okolice)

```
bash
    popularna powłoka (interpreter poleceń) zgodna z sh, zapewnia m.in. obsługę zmiennych
    (zasadniczo napisowych) oraz znaków uogólniających takich jak:
    ? oznaczający dowolny znak
    * oznaczający dowolny (także pusty) ciąg znaków
    [a-z AD] oznaczający dowolny znak z wymienionych w zbiorze ujętym w nawiasach kwadratowych,
    zbiór może być definiowany z użyciem zakresów, np. a-z AD oznacza dowolną małą literę od a do z
    włącznie, spację, dużą literę A lub D
    [!a-z] oznaczający dowolny znak z wyjątkiem znaków wymienionych w podanym zbiorze, zbiór
    może być definiowany z użyciem zakresów, np. a-z oznacza dowolną małą literę od a do z włącznie
    Argumenty linii poleceń przekazywane do wywoływanych przez powłokę poleceń rozdzielane są
    dowolną ilością spacji. Jeżeli argument ma być napisem zawierającym spację konieczne jest
    zabezpieczenie ich przy pomocy odwrotnego ukośnika (\, np. aaa\ bbb) lub ujęcie całego napisu w
    cudzysłowie pojedynczym (', np. 'aaa bbb') lub podwójnym (", np. "aaa bbb"). Oba typy
    cudzysłówów zabezpieczają przed rozwijaniem znaków uogólniających (zastępowaniem napisu ze
    znakami listą pasujących nazw / ścieżek). Cudzysłów pojedynczy (w odróżnieniu od podwójnego)
    zabezpiecza także przed interpretacją umieszczonych wewnątrz innych znaków specjalnych
    takich jak czy odwołania do zmiennych.
```

```
ssh [user@]host
    umożliwia uzyskanie powłoki zdalnego systemu poprzez szyfrowane połączenie, przydatne opcje:
    -D port tworzy tunel dynamiczny na wskazanym porcie (może on być użyty jako proxy typu SOCKS
    np. w Firefoxie w celu zapewnienia dostępu do zasobów WWW dostępnych z serwera SSH a
    niedostępny z komputera lokalnego)
    -L portNastuchiwania:hostZdalny:portZdalny tworzy tunel przekierowujący dane kierowane na
    portNastuchiwania komputera na którym działa klient ssh do portu portZdalny na serwerze hostZdalny
    poprzez serwer SSH (przydatne gdy hostZdalny jest osiągalny z hostSSH ale nie z komputera
```

lokalnego)

-R *portNasłuchiwania:hostZdalny:portZdalny* tworzy tunel przekierowujący dane kierowane na *portNasłuchiwania* komputera na którym działa serwer ssh do portu *portZdalny* na serwerze *hostZdalny* dostępnego z klienta SSH (podobnie jak -L, tyle że w drugą stronę - nasłuchiwanie po stronie serwera a host zdalny po stronie klienta)



-p *port* określa inny niż domyślny port serwera SSH

-X aktywuje przekazywanie komend X serwera ze strony zdalnej do klienta (pozwala na uruchomienie po stronie zdalnej aplikacji z GUI, które zostanie wyświetlone na lokalnym X serwerze)

tmux lub screen

multiplexer terminala - pozwala na uzyskanie wielu okien konsoli (także wyświetlanych jedno obok drugiego) na pojedynczym terminalu; ponadto pozwalają na odłączanie i podłączanie sesji, co pozwala na łatwe pozostawienie działającego programu po wylogowaniu i powrót do niego później

watch [*opcje*] *polecenie*

okresowo uruchamia *polecenie* i wyświetla jego wynik, opcja -i pozwala na określenie co ile sekund ma aktualizować wynik polecenia

script

tworzy zapis sesji (stdin, stdout, ...)

catalog roboczy

Możliwe jest wyrażanie wszystkich ścieżek od korzenia systemu plików (/), jednak nie jest to zbyt wygodne. Interpreter poleceń taki jak bash potrafi "znajdować się" gdzieś w tej strukturze plików i miejsce to nazywane jest bieżącym katalogiem roboczym (Present Working Directory). Względem niego będą wyrażane ścieżki nie zaczynające się od korzenia, może być też oznaczony jawnie przy pomocy pojedynczej kropki.

cd [*ścieżka*]

zmiana bieżącego katalogu,

warto zauważyć, iż katalogi w ścieżce oddzielamy ukośnikami /, bieżący katalog oznaczamy kropką ., nadrzędny oznaczamy dwiema kropkami ..

ścieżki zaczynające się od ukośnika / oznaczają ścieżki bezwzględne (od korzenia systemu plików), pozostałe oznaczają ścieżkę względną (wyrażoną względem bieżącego katalogu),

katalog domowy oznacza się tyldą ~

pwd

wyświetla ścieżkę do bieżącego katalogu

wyświetlanie i edycja plików

vim lub vi

vi jest chyba najbardziej zaawansowanym edytorem, którego obecność gwarantuje standard POSIX. vim jest mocno rozbudowanym jego klonem, oferującym bardzo zaawansowane funkcjonalności, powszechnie stosowanym jako zamiennik oryginalnego vi. vim obsługuje 3 podstawowe tryby pracy: komend (służący do wydawania opisanych niżej poleceń), wizualny (służący do zaznaczania i wydawania niektórych komend), edycji (wstawiania/nadpisywania - służący do wprowadzania tekstu). Podstawowa klawiszologia:

- przełączanie pomiędzy trybami:

- Esc powrót do trybu komend
- i tryb wstawiania; A tryb wstawiania ze skokiem na koniec linii, o / O tryb wstawiania ze wstawieniem nowej linii po / przed bieżącą
- R tryb zastępowania
- Insert zmiana trybu wstawiania i zastępowania
- v tryb wizualny (umożliwia zaznaczenie przy pomocy strzałek); ctrl+v tryb wizualny blokowy, V tryb wizualny liniowy
- :set paste włącza :set nopaste wyłącza tryb wklejania (nie będzie działać automatyczne formatowanie itp.)
- gv ponawia ostatnie zaznaczenie trybu wizualnego
- wycinanie i kopiowanie:
 - y skopiuj; d - wytnij (skopiuj i usuń) po y, d można podać np. 20l lub 20[strzałka w prawo] co oznacza 20 kolejnych znaków, 2w oznacza dwa słowa (więcej o takich punktach skoku poniżej)
 - x wytnij (skopiuj i usuń) znak (może być poprzedzone ilością znaków do wycięcia); wielkie X działa analogicznie, tyle że w tył
 - yy skopiuj linię; dd - wytnij (skopiuj i usuń) w obu wypadkach może być poprzedzone ilością linii do skopiowania/wycięcia
 - p wkleja po; P - wkleja przed
 - komendy kopiowania i wklejania mogą być poprzedzone jedno-znakową nazwą rejestru w którym umieszczane są dane (poprzedzamy ją znakiem " i podajemy przed licznikiem, np. "a3dd wytnie do rejestru a 3 linie), część rejestrów jest używana automatycznie, a niektóre są tylko do odczytu, podgląd aktualnej zawartości rejestrów możliwy jest przy pomocy komendy :registers
- wyszukiwanie, zastępowanie, skok do linii:
 - / szukanie w przód, ? szukanie w tył; * szukanie w przód słowa pod kursorem, # szukanie w tył słowa pod kursorem
 - n wyszukanie następnego wystąpienia; N wyszukanie poprzedniego wystąpienia
 - G przejście do wskazanej linii, numer podajemy przed G, 0 oznacza ostatnią linię w pliku, więc 0G spowoduje przejście do niej
 - :[zakres]s@regexp@napis[g] wyszukaj i zastąp wyrażenie regularne *regexp* przez *napis*; zakres może być: numerem linii, przedziałem z numerami linii postaci *pierwsza,ostatnia*, gdzie: . oznacza bieżącą linię, \$ oznacza ostatnią linię w pliku, wartość numeryczna poprzedzona + oznacza tyle kolejnych linii od bieżącej, a poprzedzona - przed bieżącą, znakiem % (co oznacza cały plik), zakresem zaznaczonym w trybie wizualnym; podanie opcji *g* powoduje zastępowanie wszystkich wystąpień a nie tylko pierwszego; znak @ pełni rolę separatora i może zostać zamiast niego użyty inny znak
- otwieranie, zapisywanie, zamykanie plików:
 - :e *ścieżka* otwarcie wskazanego pliku
 - :w zapis (można także podać ścieżkę pod jaką ma zostać zapisany plik)
 - :q wyjście
 - :q! wyjście bez zapisywania
 - :wq zapis i wyjście
- przełączanie się między otwartymi plikami i oknami:
 - :n następny plik; :N poprzedni plik
 - :split poziomy podział okna; :vs pionowy podział okna; Ctrl+W a następnie strzałka - przełączanie między oknami
- cofanie i ponawianie edycji:
 - u, :undo cofa ostatnią operację
 - Ctrl+r, :redo ponawia cofniętą operację
- punkty skoku (mogą być używane jako polecenia do poruszania się lub jako adresy w poleceniach takich jak d, y):
 - l / h / k / j jeden znak/linię w prawo / lewo / górę / dół (działa tak jak strzałki)
 - 0 / ^ / \$ początek linii / początek tekstu w linii, koniec linii
 - w / b / e następne słowo / poprzednie słowo / koniec słowa; wielkie W / B / E działa analogicznie, różni się traktowaniem spacji przy słowie

- f / F następny / poprzedni znak podany po tej komendzie, włącznie z nim (np. dFx usunie wszystko do najbliższego wystąpienia X wraz z tym X); t / T działa analogicznie, tyle wyłącza podany znak
- poprzedzenie powyższych komend liczbą powoduje powtórzenie ich tyle razy - np. 10l - 10 znaków w prawo, 3F: - trzeci dwukropek w lewo
- punktem skoku jest też wyżej opisane polecenie G poprzedzane numerem linii do której ma się odbyć skok
- punktem skoku mogą być także swobodnie umieszczane z dokumencie zakładki identyfikowane pojedynczym znakiem:
 - m i następie znak ją identyfikujący - utworzenie zakładki w miejscu kursora (np. ma - utworzy zakładkę a)
 - ` (backtick) / ' (apostrof) skok do zakładki / linii z zakładką podaną po tej komendzie
 - :marks - lista zakładek; :delmarks / :delmarks! - usunięcie zakładki / usunięcie wszystkich nie automatycznych zakładek
- inne:
 - :r plik, wstawienie zawartości pliku
 - :%!xxd pokazanie wartości numerycznych i umożliwienie edycji pliku jako binarnego; :%!xxd -r powrót do normalnej edycji
 - > / < zwiększanie / zmniejszanie wcięcia zaznaczonego (w trybie wizualnym) tekstu
 - zc zwija bieżący blok, zC zwija bieżący blok aż do najwyższego poziomu, zo rozwija bieżące zwinięcie, zO rozwija rekurencyjnie bieżące zwinięcie, zR rozwija wszystkie zwinięcia w dokumencie
 - :set wrap włącza :set nowrap wyłącza zawijania linii w podglądzie

less lub more

przeglądanie tekstu ekran po ekranie,

less posiada większe możliwości od more (w szczególności posiada możliwość przeglądanie dokumentu w tył), przydatne opcje:

-X nie czyści ekranu przy wychodzeniu z less'a (całość historii wyświetlania pliku pozostaje w historii terminala)

-F automatycznie kończy gdy wyświetlany tekst mieści się na jednym ekranie -R przepuszcza surowe sekwencje sterujące terminalem dotyczące kolorów

bvi

wzorowany na vim'ie edytor hexalny (binarny)

od lub xxd lub hexdump

wypisywanie plików interperowanych binarnie w postaci liczby oktalne, szesnastkowe, itd, przykłady:

hexdump -e '1/4 "%010d " 1/2 "%05d " 2/1 "%02x " "\n"' plik każde kolejne 8 bajtów pliku zostanie zinterpretowane jako: jedna liczba 32 bitowa liczba całkowita (4 bajty), 16 bitowowa liczba całkowita (2 bajty) i dwie liczby jedno bajtowe (wyświetlane szesnastkowo)

vbindiff

porównuje wizualnie pliki binarne (interfejs ncurces)

Operacje na systemie plików

listowanie i wyszukiwanie plików

ls [opcje] [ścieżka]

listowanie zawartości katalogu, do ważniejszych opcji należy zaliczyć:

-a wyświetlaj pliki ukryte (zaczynających się od kropki)

-l wyświetlaj pliki w formie listy z szczegółowymi informacjami (uprawnienia, rozmiar, data modyfikacji, właściciel, grupa, rozmiar)

-1 wyświetlaj pliki w formie 1 plik w jednej linii (bez dodatkowych informacji; stosowane domyślne gdy wynik komendy przekazywany jest strumieniem do innej komendy lub pliku)

-h stosuj jednostki typu k, M, G zamiast podawać rozmiar w bajtach

-t sortuj wg daty modyfikacji

-S sortuj wg rozmiaru

-r odwróć kolejność sortowania

-c użyj daty utworzenia zamiast daty modyfikacji (stosowane w połączeniu z -l i/lub -t)

-d wyświetlaj informacje o katalogu zamiast jego zawartości

find [opcje] [katalog startowy] [wyrażenie]

wyszukiwanie w systemie plików w oparciu o nazwę/ścieżkę lub właściwości pliku, do ważniejszych opcji należy zaliczyć:

-P wypisuj informacje o linkach symbolicznych a nie plikach przez nie wskazywanych (domyślne)

-L wypisuj informacje o wskazywanych przez linki symboliczne plikach

do ważniejszych elementów wyrażenia należy zaliczyć:

-name "wyrażenie" pliki których nazwa pasuje do wyrażenia korzystającego z shellowych znaków uogólniających

komenda find (w odróżnieniu np. od ls) samodzielnie interpretuje wyrażenia zawierające shellowe znaki uogólniające w argumentach tych opcji, w związku z czym konieczne może się okazać zabezpieczenie ich przed interpretacją przez powłokę np. przy pomocy umieszczenia wewnątrz pojedynczych cudzysłówów

-iname "wyrażenie" jak -name, tyle że nie rozróżnia wielkości liter

-path "wyrażenie" pliki których ścieżka pasuje do wyrażenia korzystającego z shellowych znaków uogólniających

-ipath "wyrażenie" jak -path, tyle że nie rozróżnia wielkości liter

-regex "wyrażenie" pliki których ścieżka pasuje do wyrażenia regularnego

-iregex "wyrażenie" jak -regex, tyle że nie rozróżnia wielkości liter

warunek -o warunek łączy warunki sumą logiczną OR zamiast domyślnego iloczynu logicznego AND

! warunek negacja warunku

-mtime [+|-]n pliki których modyfikacja odbyła się n*24 godziny temu

-mmin [+|-]n pliki których modyfikacja odbyła się n minut temu

-ctime [+|-]n pliki które zostały utworzone n*24 godziny temu

-cmin [+|-]n pliki które zostały utworzone n minut temu

-size [+|-]n[c|k|M|G] pliki których rozmiar wynosi n (c - bajtów, k - kilobajtów, M - Megabajtów, G - gigabajtów)

w powyższych testach + oznacza więcej niż, - oznacza mniej niż, uwaga: porównywaniu podlegają liczby całkowite, np. +1 oznacza > 1 w liczbach całkowitych tzn. ≥ 2

-exec polecenie \{\} \; dla każdego znalezionej pliku wykonaj polecenie podstawiając ścieżkę do tego pliku pod \{\} (zastosowane odwrotne ukośniki służą zabezpieczeniu nawiasów klamrowych i średnika przed zinterpretowaniem ich przez powłokę)

-execdir polecenie \{\} \;, podobnie jak -exec tyle że polecenie zostanie uruchomione w katalogu w którym znajduje się wyszukany plik

tree [ścieżka]

wyświetla drzewo katalogów i plików

stat ścieżka

wyświetla informacje o podanym pliku lub katalogu

realpath ścieżka

wypisuje uproszczoną / rozwiniętą ścieżkę (tzn. ze zredukowanymi nadmiarowymi odniesieniami względnymi),

standardowo wypisuje ścieżkę bezwzględną (opcja --relative-to=katalog powoduje wypisanie względnej w stosunku co do katalog),

standardowo zastępuje linki symboliczne wskazywaną przez nie ścieżką (opcja -s wyłącza tę funkcjonalność),

opcja -m pozwala na operowanie nie istniejącymi ścieżkami,

standardowe zachowanie odpowiada działaniu komendy readlink

du [opcje] ścieżka1 [ścieżka2 [...]]

wyświetlanie informacji o zajętej przestrzeni dyskowej przez wskazane pliki / katalogi, do ważniejszych opcji należy zaliczyć:

-s podaje łączną ilość zajętego miejsca dla każdego argumentów (zamiast wypisywać rozmiar każdego pliku)

-c podaje łączną ilość zajętego miejsca dla wszystkich argumentów

-h stosuje jednostki typu k, M, G

podawany rozmiar może się różnić (w obie strony) od wyniku ls: ls podaje rozmiar pliku (ile zawiera informacji lub ile zostało zadeklarowane że może jej zawierać), a du to ile zajmuje na dysku

df [opcje]

wyświetlanie informacji o zajętości miejsca na poszczególnych systemach plików

kopiowanie, przenoszenie, usuwanie, ...

cp [opcje] źródło1 [źródło2 [...]] cel

kopiuje wskazany plik (lub pliki) do wskazanej lokalizacji, w przypadku kopiowania wielu plików cel powinien być katalogiem, do ważniejszych opcji należy zaliczyć:

-r pozwala na (rekursywne) kopiowanie katalogów

-a podobnie jak -r, dodatkowo zachowując atrybuty plików

-l zamiast kopiować tworzy twarde dowiązania (hard links)

-s zamiast kopiować tworzy linki symboliczne do plików

-f nadpisywanie bez pytania

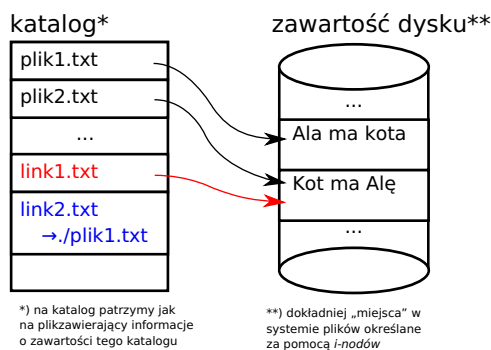
-i zawsze pytaj przed nadpisaniem

ln źródło1 [źródło2 [...]] cel

tworzy link twardy do wskazanego pliku (lub plików) w wskazanej lokalizacji, w przypadku wskazania wielu plików źródłowych cel powinien być katalogiem, do ważniejszych opcji należy

zaliczyć:

- s tworzy dowiązania symboliczne zamiast twardych
- r używa ścieżki względnej zamiast bezwzględnej przy tworzeniu dowiązań symbolicznych



ln plik2.txt link1.txt

powoduje utworzenie linku twardego link1.txt, czyli wpisu w katalogu wskazującego na te same dane (ten sam i-node) co plik2.txt

ln -s ./plik1.txt link2.txt

powoduje utworzenie linku symbolicznego link2.txt, czyli wpisu w katalogu wskazującego na podaną ścieżkę (./plik1.txt)

Link twardy jest innym uchwytem do tych samych danych i może być używany także po skasowaniu oryginalnego pliku. Liczbę dowiązań do danego pliku pokazuje m.in. komenda `ls` z opcją `-l`. Nie można utworzyć linków twardych do katalogów, ani do plików na innym zasobie dyskowym (innym systemie plików).

Link symboliczny wskazuje na konkretną ścieżkę (względną lub bezwzględną – co może mieć znaczenie przy przenoszeniu takiego linku) do dowolnego (nawet nie istniejącego – wtedy mówimy o zerwanym linku) pliku lub katalogu.

- mv [opcje] źródło1 [źródło2 [...]] cel**
przenosi wskazane pliki / katalogi do wskazanej lokalizacji, w przypadku przenoszenia wielu plików `cel` powinien być katalogiem, do ważniejszych opcji należy zaliczyć:
- f nadpisywanie bez pytania
 - i zawsze pytaj przed nadpisaniem
- rm [opcje] ścieżka1 [ścieżka2 [...]]**
usuwa wskazane pliki, do ważniejszych opcji należy zaliczyć:
- r pozwala na na (rekursywne) kasowanie katalogów wraz z zawartością
 - f usuwanie bez pytania
 - i zawsze pytaj przed usunięciem
- touch [opcje] ścieżka1 [ścieżka2 [...]]**
zmiana daty modyfikacji pliku (wykorzystywany także do tworzenia plików)

katalogi

- mkdir [opcje] ścieżka1 [ścieżka2 [...]]**
tworzy wskazane katalogi, do ważniejszych opcji należy zaliczyć:
- p pozwala na tworzenie całej ścieżki a nie tylko ostatniego elementu, nie zgłasza błędu gdy wskazany katalog istnieje
- rmdir [opcje] ścieżka1 [ścieżka2 [...]]**
usuwa wskazane katalogi (muszą być puste)

zdalne kopiowanie

- scp [opcje] źródło1 [źródło2 [...]] cel**
kopiuje wskazany plik (lub pliki) do wskazanej lokalizacji, w przypadku kopiowania wielu plików `cel` powinien być katalogiem, do ważniejszych opcji należy zaliczyć:
- r pozwala na (rekursywne) kopiowanie katalogów
 - P `port` określa port SSH
- W odróżnieniu od `cp` `źródło` lub `cel` w postaci `[user@]host:[ścieżka]` wskazują na zdalny system dostępny poprzez SSH.
- rsync [opcje] źródło cel**
kopiuje (synchronizację) pliki i drzewa katalogów (zarówno lokalnie jak i zdalnie), do ważniejszych opcji należy zaliczyć:
- r pozwala na (rekursywne) kopiowanie katalogów
 - l kopiuje linki symboliczne jako linki symboliczne (zamiast kopiowania zawartości pliku na który wskazują)
 - t zachowuje czas modyfikacji plików
 - u kopiuje tylko gdy plik źródłowy nowszy niż docelowy
 - c kopiuje tylko gdy plik źródłowy i docelowy mają inne sumy kontrolne
 - delete usuwa z docelowego drzewa katalogów elementy nie występujące w drzewie źródłowym
 - e 'ssh' pozwala na kopiowanie na/z zdalnych systemów za pośrednictwem ssh, `źródło` lub `cel` w postaci `[user@]host:[ścieżka]` wskazują na zdalny system
 - partial --partial-dir=".tmp-" zachowuje skopiowane częściowo pliki w katalogu `.tmp` (pozwala na przerwanie i wznowienie transferu pliku)
 - progress pokazuje postęp kopiowania

- `--exclude="wzorzec"` pomija (w kopiowaniu i kasowaniu) pliki pasujące do *wzorzec* (wzorzec może zawierać znaki uogólniające powłoki)
- `n` symuluje pracę (pokazuje co zostałoby skopiowane, ale nie kopiuje)

`sshfs [opcje] host:ścieżka`
montuje zdalny system plików z użyciem FUSE (filesystem in userspace) oraz SSH, do ważniejszych opcji należy zaliczyć:

- `p port` określa inny niż domyślny port serwera SSH
- `o workaround=rename`, który zapewnia poprawne `mv` na istniejący plik

Operacje na archiwach

`tar`
archiwizuje wiele plików w postaci jednego, przykłady:
`tar -xf plik.tar` - wypakowuje zawartość niekompresowanego archiwum *plik.tar* interpretując zapisane w archiwum ścieżki względem bieżącego katalogu
`tar -czf - ścieżka1 [ścieżka2 [...]] | ssh [user@]host 'cat > plik.tgz'` - archiwizuje wskazane pliki/katalogi bezpośrednio na zdalny system z użyciem ssh i kompresji gzip

`ar polecenie[modyfikator] archiwum [pliki]`
archiwizuje wiele plików w postaci jednego (format wykorzystywany w plikach bibliotek linkowanych statycznie i w pakietach Debiana), przykłady:
`ar rcs libAA.a aa*.o` - tworzy bibliotekę *libAA.a* z wszystkich plików *aa*.o*
`ar x pakiet.deb` - wypakowuje zawartość pakietu Debiana z pliku *pakiet.deb* (będą to dwa archiwa i plik z wersją formatu pakietu)

`7z`
archiwizator obsługujący wiele formatów, w tym formaty z silną kompresją

Operacje na plikach (tekstowych)

`cat [opcje] [plik1 [plik2 [...]]]`
wypisywanie i łączenie plików

`tac [opcje] [plik]`
wypisywanie plików z odwróceniem kolejności linii (odwrócenie kolejności wypisywania; może być użyte np. do odwrócenia porządku sortowania)

`tail [opcje] [plik]`
wyświetla ostatnie linie pliku, przydatne opcje:

- `n x` określa że ma zostać wyświetlone *x* ostatnich linii
- `f` uruchamia dopisywania (gdy do pliku zostaną dopisane nowe linie `tail` je wyświetli)

`head [opcje] [plik]`
wyświetla początkowe linie pliku, przydatne opcje:

- `n x` określa że ma zostać wyświetlone *x* pierwszych linii

`multitail`
`tail` lepiej operujący na wielu plikach równocześnie

`rev [plik]`
wypisuje plik odwracając kolejność znaków w każdym wierszu

`wc`
liczy linie, słowa, znaki i bajty

`grep [opcje] wyrażenie [plik1 [plik2 [...]]]`
wyszukuje pasujące do wyrażenia regularnego *wyrażenie* linie w plikach, przydatne opcje:

- `v` zamiast pasujących wypisz nie pasujące
- `i` ignoruj wielkość liter
- `a` przetwarzaj pliki binarne jak tekstowe
- `E` korzystaj z *Extended Regular Expressions* (ERE) zamiast *Basic Regular Expressions* (BRE)
- `P` korzystaj z *Perl-compatible Regular Expressions* (PCRE) zamiast *Basic Regular Expressions* (BRE)
- `r` rekursywnie przetwarzaj podane katalogi wyszukując w wszystkich znalezionych plikach
- `R` jak `-r`, ale zawsze podąża za linkami symbolicznymi
- `--exclude="wyrażenie"` pomini pliki pasujące do *wyrażenie* (może zawierać znaki uogólniające powłoki)
- `l` wypisuje pliki z pasującymi liniami
- `L` wypisuje pliki z bez pasujących linii
- `f` wczytaj wyrażenia z podanego pliku
- `e` może być użyta do poprzedzenia wyrażenia (przydatne zwłaszcza jeżeli chcemy podać kilka)

`diff ścieżka1 ścieżka2`
porównuje pliki lub katalogi (w przypadku tych drugich porównuje ze sobą pliki o takich samych nazwach oraz zgłasza fakt występowania pliku tylko w jednym z katalogów), przydatne opcje:

- `r` rekursywnie przetwarzaj podane katalogi
- `u` wypisuje różnice w formacie "unified"
- `c` wypisuje różnice w formacie "context"

`vimdiff ścieżka1 ścieżka2`
 porównuje pliki wyświetlając je jeden obok drugiego (podobnie jak `diff` z opcją `-y`), pozwalając jednak na edycję tych plików

`patch`
 stosuje plik łat (wynik `diff'a`) w celu zmodyfikowania plików, typowo:
`patch -pn < plik.diff` co powoduje zastosowanie zmian opisanych w `plik.diff` na plikach w bieżącym katalogu, `n` określa ilość poziomów ścieżek podanych w pliku łat które mają zostać zignorowane

`combinediff / splitdiff`
 łączy dwa pliki łat / dzieli plik łat złożony z przyrostowych zmian na osobne (pakiet *patchutils*)

`flipdiff`
 zmienia kolejność w której będą nakładane dwie łat dotyczące tych samych plików (pakiet *patchutils*)

`interdiff`
 porównywanie dwóch plików łat (pakiet *patchutils*)

`lsdiff`
 wyświetlanie plików modyfikowanych przez łatę (pakiet *patchutils*)

`filterdiff`
 wyciągnięcie z pliku łat zmian dotyczących wybranych plików (pakiet *patchutils*)

`grepdiff`
 przeszukiwanie pliku łat celem znalezienia plików w modyfikacja których pasuje do wyrażenia regularnego (pakiet *patchutils*)

`rediff / editdiff / recountdiff`
 przeliczanie numerów linii w pliku łat (pakiet *patchutils*)

`unwrapdiff / dehtmldiff / fixcvsdiff`
 naprawa pliku łat uszkodzonego przez zawijanie linii, html-izację, błędy w generowaniu łat przez cvs (pakiet *patchutils*)

`quilt`
 narzędzie ułatwiające utrzymywanie zmian do jakiś plików w postaci zbioru łat

`sort [plik]`
 sortuje linie w wskazanym pliku, przydatne opcje:
`-n` traktuj liczby jako wartości numeryczne a nie napisy
`-i` ignoruj wielkość liter
`-r` odwróć kolejność sortowania
`-k n` sortuj wg kolumny `n`
`-t sep` kolumny rozdzielane są przy pomocy separatora `sep`

`tr [opcje] zbiór1 [zbiór2]`
 zastępuje znaki z `zbiór1` odpowiadającymi im pod względem kolejności znakami z `zbiór2`, przydatne opcje:
`-d` zamiast zastępować usuwa znaki występujące w `zbiór1`
 uwaga: często działa poprawnie wyłącznie dla znaków jedno bajtowych - można wtedy zastosować `sed -e 'y#zbiór1#zbiór2#' lub tr -s -f 'zbiór1 zbiór2'`

`trs`
 (wchodzący w skład pakietu `konwert`) bardziej inteligentny od `tr` program służy do zamieniania znaków z jednego zbioru na znaki z drugiego zbioru (w odróżnieniu od `tr` nadaje się do znaków kilku bajtowych - utf8, lepiej od `sed'a` sprawdza się przy zmianie kodowań - patrz przykład w manualu)

`sed [opcje] [pliki]`
 edytuje plik zgodnie z podanymi poleceniami, przydatne opcje:
`-e "polecenie"` - wykonuj na pliku `polecenie` (może wystąpić wielokrotnie celem podania wielu poleceń)
`-f "plik"` - wczytaj polecenia z pliku `plik`
`-E` - używaj rozszerzonych wyrażeń regularnych
`-i` - modyfikuj podany plik zamiast wypisywać zmieniony na stdout
`-n` - wyłącza domyślne wypisywanie linii, wypisanie musi być wykonane jawnie poleceniem `p`
 przydatne polecenia:
`s@regexp@napis@g` - wyszukaj dopasowania do wyrażenia regularnego `regexp` i zastąp je przez `napis`, podanie opcji `g` powoduje zastępowanie wszystkich wystąpień a nie tylko pierwszego, znak `@` pełni rolę separatora i może zostać zamiast niego użyty inny znak
`y@zbiór1@zbiór2@` - zastąp znaki z `zbiór1` znakami odpowiadającymi im pod względem kolejności znakami z `zbiór2`, znak `@` pełni rolę separatora i może zostać zamiast niego użyty inny znak
 możliwe jest też m.in. adresowanie linii na których ma być wykonywana operacja, np: `0,/regexp/s@regexp@napis@` wykona polecenie `s` na liniach od początku pliku do linii pasującej do wyrażenia regularnego `regexp`, czyli zastąpi tylko pierwsze wystąpienie w pliku

`cut [opcje] [pliki]`

wybiera z pliku zadany zestaw kolumn, przydatne opcje:

- f *nnn* wypierz kolumny określone przez *nnn* (np. 1,3-4,6- oznacza kolumnę 1, kolumny od 3 do 4 i od 6, a -3 oznacza 3 pierwsze kolumny)
- d *sep* kolumny rozdzielane są przy pomocy separatora *sep* (musi być pojedynczym jedno bajtowym znakiem, aby ominąć to ograniczenie należy skorzystać z `awk`)

`awk`

skryptowy język umożliwiający przetwarzanie tekstowych baz danych postaci linia=rekord, pola oddzielane ustalonym separatorem (m.in. łączy funkcjonalność `grep`, `cut`, itp), szerzej omówiony w przykładzie przetwarzania napisów w bashu

`paste`

łączy (odpowiadające sobie pod względem numerów) linie z dwóch plików

`join`

łączy linie z dwóch plików w oparciu o porównanie wskazanego pola

`comm`

porównuje dwa posortowane pliki pod względem unikalności linii (może wypisać wspólne lub występujące tylko w jednym z plików)

`uniq`

usuwa powtarzające się linie z posortowanego pliku, przydatne opcje:

- c wypisz liczbę powtórzeń
- d wypisz tylko linie z 2 lub więcej wystąpieniami
- u wypisz tylko linie z 1 wystąpieniem

`xmlstarlet`

narzędzie do przetwarzania, modyfikowania plików XML (można powiedzieć taki odpowiednik `sed/awk` dla xml'a)

`xsltproc`

narzędzie do przetwarzania XSLT 1.0

Uprawnienia do plików

Podstawowe unixowe uprawnienia do plików składają się z trzech członów: uprawnienia dla właściciela (u), grupy (g) i pozostałych użytkowników (o).

W każdym z członów mogą być przyznane uprawnienia do czytania (r), pisania (w) i wykonywania (x); w odniesieniu do plików jest to intuicyjne (uprawnienie do wykonywania jest potrzebne do uruchomienia programów), natomiast w stosunku do katalogów wygląda to następująco: uprawnienia do czytania pozwalają na listowanie zawartości, do wykonania pozwalają na dostęp, do zawartości katalogu (wejścia do niego) do pisania na tworzenie nowych obiektów wewnątrz niego i zmienianie nazw istniejących.

Warto zaznaczyć iż program w pliku nie mającym praw wykonywalności też może być wykonany (w przypadku skryptu poprzez uruchomienie interpretera i podanie tego programu jako odpowiedniego argumentu wywołania, w przypadku typowych binarek poprzez wywołanie `/lib/ld-linux.so.2` z tym plikiem i jego opcjami jako parametrem.

Rozszerzeniem podstawowych uprawnień opisanych powyżej jest mechanizm Filesystem Access Control List (ACL, fACL).

Jest on opcjonalnym mechanizmem który (na wspierających go systemach plików) pozwala na definiowanie indywidualnych uprawnień do pliku dla poszczególnych użytkowników i grup - plik ma nadal swojego właściciela, grupę i wszystkich pozostałych, ale przed prawami dla "others" wchodzić prawa użytkowników i grup definiowanych w ACL. Wypadkowe prawa obliczane są jako suma wyników z praw użytkownika i grup do których należy.

ACL pozwala ponadto definiować uprawnienia domyślne dla nowo powstałych plików w katalogu (są one opcją katalogu).

Wszystkie poniższe komendy przyjmują opcję `-R` powodującą rekursywne wykonywanie zmian na drzewku katalogów/plików rozpoczynającym się w podanej ścieżce.

`chown [opcje] właściciel ścieżka`

zmiana właściciela pliku

`chgrp [opcje] grupa ścieżka`

zmiana grupy do której należy plik pliku

`chmod [opcje] uprawnienia ścieżka`

zmiana prawa dostępu do pliku(ów)

`getfacl [opcje] [ścieżka]`

dczyt uprawnień związanych z listami kontroli dostępu fACL

`setfacl [opcje] [ścieżka]`

ustawianie uprawnień związanych z listami kontroli dostępu fACL

Dodatkowo należy wspomnieć też o poleceniach takich jak:

lsattr / chattr
wyświetla / modyfikuje atrybuty plików związanych z systemem plików (np. zabrania jakiegokolwiek modyfikacji pliku)

getcap / setcap
wyświetla / modyfikuje atrybuty plików związanych z właściwościami jądra (zasadniczo zwiększonymi uprawnieniami programów je posiadających, ale bardziej ograniczonymi niż wykonanie na prawach root przez SUID)

Użytkownicy

id [użytkownik]
informacja o użytkowniku (m.in. grupy do których należy)

whoami
informacja o aktualnym użytkowniku

w lub who
informacja o zalogowanych użytkownikach

last
ostatnio logujący się użytkownicy

lastb
ostatnie nieudane logowania

lastlog
ostatnie logowania użytkowników

passwd [użytkownik]
zmiana hasła

su [użytkownik]
przełącza użytkownika (aby przełączony użytkownik miał dostęp do "naszego" x serwera wcześniej wydajemy xhost LOCAL:użytkownik)

sudo
program pozwalający na wykonywanie uprzywilejowanych komend przez wyznaczonych użytkowników

Procesy i zasoby

ps [opcje]
wyświetla aktualnie działające procesy i informacje o nich
np. kombinacja opcji -Af powoduje wyświetlenie wszystkich procesów w rozszerzonym formacie wypisywania

pgrep
wyświetla filtrowaną listę aktualnie działających procesów

pstree
wyświetla drzewo aktualnie działających procesów

pwdx
podaje katalog roboczy podanego procesu

top
monitorowanie procesów obciążających CPU, pamięć, itd

iotop
monitorowanie procesów obciążających I/O

fuser
informacje o procesach korzystających z plików

lsdf
informacje o plikach otwartych przez program (można podglądać także w /proc/PID/fd/)

socklist
wyświetla listę otwartych socketów

vmstat
informacje o obciążeniu systemu

ipcs
informacje o wykorzystaniu pamięci współdzielonej

kill [opcje] pid
przesyła sygnał do procesów o podanych PID

killall [opcje] nazwa
przesyła sygnał do procesów o pasującej nazwie

time
czas działania programu

nice / renice / ionice / cpulimit
zarządzanie priorytetami i limitami procesu

taskset
zarządzanie przypisaniem procesu do CPU / rdzenia

free
informacja o zajętości pamięci

kill i zabijanie procesu

Polecenie kill domyślnie wysyła sygnał SIGTERM, który jest prośbą o zakończenie procesu (proces może ją uszanować lub nie, np. zignorować). Więc sam kill nie zabija procesu.

Wiele sygnałów może zostać przechwyconych i obsłużonych (zignorowanych) przez proces do którego są adresowane. Istnieją także sygnały, które nie mogą zostać obsłużone bądź zignorowane są to m.in.: SIGKILL (zakończenie procesu bez dania mu jakiegokolwiek szansy zrobienia czegoś na „do widzenia”, wysyłany przez \Verb#kill -9#), SIGSTOP (wstrzymanie procesu).

Ctrl+C / Ctrl+Z / Ctrl+D / ...

Ctrl+C wysyła sygnał SIGINT do procesu zajmującego terminal na którym został on wprowadzony. Sygnał ten jest prośbą o zakończenie procesu, którą proces może uszanować lub nie (np. może całkiem zignorować lub poprosić o potwierdzenie). Jest on podobny do SIGTERM, jednak jest innym sygnałem i może być inaczej obsłużony (np. w SIGTERM nie ma większego sensu pytać o potwierdzenie).

Ctrl+Z wysyła sygnał SIGTSTP do procesu zajmującego terminal na którym został on wprowadzony. Sygnał ten jest prośbą o wstrzymanie procesu i oddanie terminala, prośba ta może być zignorowana przez proces. Proces przerywany w ten sposób może być wznowiony poleceniem fg (które wznowi go jako pierwszoplanowy – okupujący terminal) lub bg (które wznowi go jako proces w tle – oddając terminal, przodkowi który go posiadał wcześniej).

Ctrl+D nie wysyła żadnego sygnału, działa tylko gdy proces czyta dane z terminala (podłączonego zazwyczaj do jego standardowego wejścia). Wysyła on do terminala znak EOT (End-of-Transmission), w efekcie czego:

- (jeżeli bufor wejściowy jest niepusty) terminal wypycha bufor wejściowy do programu (tak jak po wprowadzeniu nowej linii), albo
- (jeżeli nie ma znaków w buforze) terminal zamyka strumień wprowadzanych danych do programu

Program nie otrzymuje w strumieniu znaku EOT (jest on przechwycony przez terminal). Zamknięcie strumienia wejściowego na ogół prowadzi także do zakończenia działania programu, jednak (w odróżnieniu od Ctrl-C) pozwala programowi na normalne przetworzenie wprowadzonych danych.

Ctrl+S wstrzymuje przewijanie (odświeżanie) terminala, aby wznowić należy użyć **Ctrl+Q**

Informacje o środowisku i systemie

env / printenv
zmienne środowiskowe wypisywanie oraz przekazywanie do uruchamianego programu

export
ustawianie i usuwanie zmiennych środowiskowych

set / unset
ustawianie, wyświetlanie, usuwanie zmiennych / funkcji / opcji w środowisku powłoki bash

uptime
informacja o tym jak długo działa system ...

uname
informacja o systemie

lsb_release
informacja o systemie zgodnym z LSB

getent
informacje z baz administracyjnych (typu użytkownicy, grupy, hosty), z uwzględnieniem ich konfiguracji w /etc/nsswitch.conf

Diagnostyka sieci

Adresy

ipcalc oraz sipcalc
kalkulator IP (pozwalający na obliczanie adresów sieci rozgłoszeniowych, zmianę notacji itd)

whois
informacje z bazy whois (o domenie lub adresie IP)

Dostępność i trasy do hostów

`ping [opcje] host` lub `ping6 [opcje] host`

sprawdzanie dostępności hosta z użyciem protokołu ICMP (obecnie komenda `ping6` najczęściej jest równoważna poleceniu `ping` z opcją `-6` wymuszającą używanie jedynie IPv6, na starszych systemach komenda `ping` może nie obsługiwać adresów IPv6 i wtedy konieczne jest stosowanie do nich polecenia `ping6`), ważniejsze opcje:

-c *n* wykonaj *n* (domyślnie pyta do momentu przerwania przy pomocy np. Ctrl-C, lub sygnału wysłanego z użyciem komendy `kill`)

-n nie zamieniaj adresu IP hosta który odpowiedział na nazwę domenową

`tracert`, `tracert6`, `tracert`, `tracert6`, `tracert` lub `tracert6`

sprawdzanie ścieżki do hosta (wypisanie listy routerów przez które przechodzi pakiet w drodze do wskazanego hosta)

Istnieją różne warianty tych poleceń (nawet pod tą samą nazwą), różnią się one stosowanymi mechanizmami i domyślnymi opcjami. Generalnie wszystkie uruchamia się na zasadzie polecenie `[opcje] host`. Warianty z 6 na końcu nazwy będą używały jedynie adresów IPv6, natomiast polecenia bez 6 na końcu nazwy mogą potrafić ich używać lub nie. Wszystkie popularne warianty pozwalają na podanie opcji `-n` wyłączającej zamienianie adresu IP hosta który odpowiedział na jego nazwę domenową.

Może zdarzyć się że śledzenie urwie się na jakimś hoście (np. z powodu jego konfiguracji lub błędów w jego oprogramowaniu sieciowym), może się zdarzyć że przy użyciu innej komendy z tej grupy (lub zmianie opcji) uda się prześledzić dalszą trasę pakietu.

`tracert` `[opcje] host`

sprawdzanie ścieżki do hosta (czyli podobnie jak `tracert` i `tracert`) w trybie ciągłym (z ciągłym odświeżaniem) wraz z wypisywaniem informacji o stratach pakietów i opóźnieniach na poszczególnych odcinkach, ważniejsze opcje

-n nie zamieniaj adresu IP hosta który odpowiedział na nazwę domenową

`nmap`

skaner sieciowy - sprawdzanie dostępnych hostów w sieci, otwartych portów, itd

`arping`

narzędzie do pingowania z wykorzystaniem zapytań ARP zamiast ICMP

istnieją dwie zasadnicze odmiany: z `iputils` oraz z `synscan`; ta druga zawarta w debianowym pakiecie "arping" umożliwia także pingowanie po adresie MAC (ale nie przez RARP, bo on nie do tego służy), aby to jednak działało host docelowy nie może ignorować pingów rozgłoszeniowych, metoda obejścia opisana jest w README arping'a

`arp-scan`

wyszukiwanie hostów w oparciu o zapytania ARP (można powiedzieć że jest to równoważne uruchamianiu komendy `arping` w pętli)

DNS

`dig [opcje] nazwa [typRekordu]`

narzędzie do uzyskania informacji z DNS, pozwala na określenie poprzez `@adres` serwera który chcemy odpytać oraz na określenie (poprzez drugi argument) typu rekordu który chcemy uzyskać zamiast typu rekordu można podać: ANY (powoduje odpytanie o wszystkie rekordy) lub AXFR (powoduje wysłanie prośby o transfer całej strefy, działa jeżeli dany host ma prawo transferu całej strefy z danego serwera)

`host [opcje] nazwa[ip [server]]`

narzędzie do zamiany adresów domenowych na IP i odwrotnie oraz wyciągania innych informacji z DNS (np. rekordy MX)

`nslookup [opcje] nazwa[ip [server]]`

narzędzie do zamiany adresów domenowych na IP i odwrotnie oraz wyciągania innych informacji z DNS (np. rekordy MX)

`dnstracer`

śledzenie trasy zapytań DNS

`dnswalk`

debuger DNS

IPv6

`ndisc6`

testowanie ICMPv6 Neighbor Discovery

`rdisc6`

testowanie ICMPv6 Router Discovery

`rltracert6`

trasa pakietów do danego hosta IPv6 z użyciem UDP/ICMP

`tcpspray6`

pomiar prędkości łącza z użyciem TCP/IP Discard/Echo

na6 / ns6
wysyłanie pakietów Neighbor Advertisement / Solicitation

ra6 / rs6
wysyłanie pakietów Router Advertisement / Solicitation

ni6 / rd6
wysyłanie pakietów ICMPv6 Node Information / Redirect

scan6
skanowanie sieci IPv6

debugowanie łączności sieciowej

netcat lub nc lub netcat6
program pozwalający na wysyłanie pakietów TCP i UDP z zdefiniowaną przez nas zawartością, oraz odbiór pakietów TCP i UDP (słuchanie na wskazanym porcie), umożliwia m.in. testowanie usług sieciowych (takich jak smtp, www, jabber, ...); uwaga: występuje w kilku wersjach różniących się opcjami

telnet
program umożliwiający zdalny (nieszyfrowany, łącznie z hasłem!) dostęp do powłoki, a także (podobnie jak netcat) m.in. testowanie usług sieciowych

swaks
narzędzie do testowania SMTP

tcpdump
przechwytuje komunikację sieciową celem analizy nagłówków lub pełnej zawartości pakietów (wsparcie dla niektórych z protokołów warstw wyższych wymaga doinstalowania - np. obsługę DHCP zapewnia dhcpcap)

wireshark lub tshark
przechwytuje komunikację sieciową celem analizy nagłówków lub pełnej zawartości pakietów, wspiera dekodowanie wielu protokołów warstwy aplikacyjnej, wireshark posiada graficzny interfejs użytkownika, tshark jest wersją konsolową

informacje o wykorzystaniu i prędkości sieci

netstat
informacje o sieci

iptraf
monitor IP LAN

nload
graficzne (ncurses) pokazywanie wykorzystania (prędkości) interfejsów sieciowych

ttcp
testuje prędkość połączenia sieciowego (strona domowa, najnowsza wersja oraz mutacja)

iperf
pomiar prędkości połączenia sieciowego

Konfiguracja sieci

podstawowa konfiguracja interfejsów

ip
informacje na temat interfejsów sieciowych (adresów IP, etc) i routingu oraz ich konfiguracja

ifconfig
informacje na temat interfejsów sieciowych (adresów IP, etc) oraz ich konfiguracja

route
konfiguracja routingu

arp
wyświetlanie i manipulowanie tablicą protokołu ARP

dhclient
konfiguracja automatyczna interfejsu w oparciu o DHCP

konfiguracja mechanizmów jądrowych w Linuxie

vconfig
konfiguracja (tagowanych) VLANów

brctl
konfiguracja bridge'ów (softwarowego switcha działającego w warstwie 2)

ifenslave
konfiguracja bondingu

iptables
jądrowe reguły filtrowania i routingu pakietów

iptables
jądrowe reguły filtrowania i routingu pakietów dla bridge'ów

iptables
jądrowe reguły filtrowania i routingu pakietów w warstwie drugiej OSI

konfiguracja kart sieciowych (w tym WiFi)

tc
konfiguracja ustawień kontroli przepływu (kolejkowanie, itp)

ethtool
konfiguracja kart ethernetowych pod względem takich parametrów jak prędkość transmisji, duplex, wake on lan, etc

iwconfig
konfiguracja (większości - do części trzeba użyć wlanctl-ng) bezprzewodowych interfejsów sieciowych

wpa_supplicant
konfiguracja większości bezprzewodowych interfejsów sieciowych z wsparciem dla WPA

wpa_cli
konfiguracja większości bezprzewodowych interfejsów sieciowych z wsparciem dla WPA

iwlist
dodatkowe informacje z bezprzewodowych interfejsów sieciowych (przydatna zwłaszcza opcja scan pokazująca dostępne sieci)

Narzędzia programistyczne

clang lub gcc
kompilator C

clang++ lub g++
kompilator C++

ld
linker

python3 lub python
interpreter Pythona

cmake
system generacji plików Makefile

make
system budowania aplikacji w oparciu o pliki Makefile

ldd
wyświetlanie wykorzystywanych przez program bibliotek dynamicznych

ldconfig
konfiguracja bibliotek dynamicznych (m.in. określanie ścieżek w jakich są wyszukiwane), refresh bazy dostępnych bibliotek; patrz też /etc/ld.so.conf

git, hg, svn
systemy kontroli wersji

Narzędzia administracyjne

shutdown
wyłączanie i restartowanie systemu (komputera), podobnie do bezpośredniego operowania na poziomach init

parted lub fdisk
oglądanie i modyfikowanie podziału dysku na partycje

mount / umount
montowanie / odmontowywanie systemu plików
warto zwrócić uwagę na opcję montowania -o bind umożliwiającą montowanie katalogu w innym katalogu oraz na -o remount umożliwiającą zmianę parametrów podmontowanego systemu plików

chroot / fakechroot / schroot
narzędzia do uruchamiania programu / programów z podmienionym katalogiem głównym

dd / ddrescue
tworzenie kopii urządzenia / pliku obrazu (ddrescue z lepszą tolerancją błędów odczytu)

apt lub aptitude
menager pakietów systemach opartych o Debiana

dpkg
niskopoziomowe narzędzie do zarządzania pakietami deb

adduser / deluser oraz addgroup / delgroup

- zarządzanie użytkownikami
- dmesg
 - logi jądra
- lspci
 - listowanie urządzeń podłączonych do magistrali PCI, PCI-express, ... (aktualizacja bazy identyfikacyjnej update-pciids)
- lsusb
 - listowanie urządzeń podłączonych do magistrali USB
- lsscsi
 - listowanie urządzeń obsługiwanych przez podsystem SCSI (m.in. SATA, SCSI, FC)
- dmidecode
 - odczyt informacji o sprzęcie
- ddcprobe
 - odczyt informacji o monitorze
- smartctl
 - odczyt informacji S.M.A.R.T. dotyczących dysku twardego
- hdparm / sdparm
 - ustawienia parametrów dysków twardych (np. opcja -S ustawia czas po jakim usypiany jest dysk)
- hddtemp
 - narzędzie do kontroli temperatury dysku twardego
- mbmon / xmbmon
 - monitorowanie parametrów pracy płyty głównej
- sensors
 - dane z czujników w komputerze (takie jak napięcia, temperatury, prędkości wentylatorów)
- acpi
 - stan zasilania / baterii

Inne narzędzia

- echo / printf
 - wypisuje argumenty na standardowe wyjście
- xargs
 - pobierz argumenty z stdin
- mkfifo
 - tworzy łącze nazwa (specjalny plik który może być wykorzystywany do komunikacji między procesami na zasadzie podobnej do |)
- seq
 - wypisuje kolejne liczby (przydane np. w for)
- tee
 - zapisuje stdin do podanego pliku, równocześnie wypisując go na stdout
- file
 - rozpoznaje typ pliku (w oparciu o zawartość)
- iconv
 - konwersje kodowań plików tekstowych
- konwert
 - konwersje kodowań plików tekstowych - zarówno pomiędzy różnymi kodowaniami danego zbioru znaków, jak też pomiędzy kodowaniami nie pokrywającymi się czy też kodowaniami znaków 8 bitowych na mniejszej ilości bitów, na przykład:
konwert utf8-ascii "inteligentnie" usunie znaki nie ascii z pliku kodowanego w utf-8 (np. znaczki z polskimi ogonkami zamieni na odpowiednie znaki ASCII bez tych ogonków);
konwert qp-8bit pozwoli zamienić kodowanie quoted printable na normalne 8 bitowe (rtf-8bit zrobi to z kodowaniem rtf'u)
- mewencode / mewdecode
 - program (stanowiący część pakietu narzędzi dodatkowych dla klienta pocztowego Mew) do obsługi kodowań mime (w tym Quoted-Printable, base64), m.in. zmienia kodowanie base64 na 8 bitowe
- qprint
 - program do kodowania i dekodowania "Quoted-Printable"
- base64
 - program do kodowania i dekodowania base64
- strings
 - wypisuje sekwencje znaków drukowanych (określanie zawartości plików nietekstowych)
- command -v komenda

zwraca wykonywaną ścieżkę / polecenie przy wykonywaniu *komenda*

date
data i czas, program ten potrafi także przeliczać datę i czas - np. `date -d @847103830 '+%Y-%m-%d %H:%M:%S'`, `date -d '1996-11-04 11:37:10' '+%s'`, `date -d '1996-11-04 11:37:10' +3week -2days'`

cal
kalendarz

wget / curl
pobieranie stron internetowych i plików

calc albo apcalc
zaawansowany kalkulator o składni podobnej do C umożliwiający wykonywanie m.in. operacji logicznych (w tym binarnych)

bc
kalkulator (warto rozważyć utworzenie aliasu `alias 'bc'='bc <(echo scale=3)'` (aby zaraz po starcie mieć 3 miejsca po przecinku) lub `alias 'bc'='bc -l'` (dla pełniejszej precyzji)

numconv
program do konwersji systemów liczbowych oprócz systemów pozycyjnych o różnych podstawach obsługuje także inne systemy liczbowe - np. rzymski

gnuplot
rysowanie wykresów

clear
czyści terminal

reset
przywraca ustawienia terminala (np. gdy po wyświetleniu pliku binarnego są dziwne krzaczkę)

stty
konfiguruje terminal (np. port szeregowy `/dev/ttyS*`, później możemy korzystać z niego np. za pomocą `cat`, `echo`, ... i przekierowań potoków - np. `cat /dev/ttyS0 > ~/serial.log` będzie logowało informacje z portu do wskazanego pliku)

startx
uruchamia środowisko graficzne

xinit
uruchamia program w środowisku graficznym (wraz z startem środowiska), np. `xinit -e __WYBRANY_PROGRAM__ - - :1` uruchomi X'y na ekranie 1 (dawniej `Alt+Ctrl+F8`, obecnie na ogół na bieżącym terminalu) a w nich wskazany program

xset
konfigurowanie ekranu x-serwera, np. `sleep 1; xset dpms force off` spowoduje wyłączenie (uśpienie) monitora

Standardowe usługi

Systemy operacyjne zapewniają także szereg usług. Znaczna część z nich dostępna jest z poziomu biblioteki standardowej C (np. obsługa plików, komunikacji sieciowej, zamiany nazw domenowych oraz nazw usług na adresy numeryczne). Inne z nich realizowane są poprzez dedykowane procesy działające w tle (bez bezpośredniej interakcji z użytkownikiem), tzw. *daemon-y*. Poniżej omówione zostały niektóre z usług zapewnianych przez system operacyjny.

Zamiana nazw na adresy numeryczne

Biblioteka standardowa zapewnia możliwość zamiany nazw hostów na ich numery IP, zamiany nazw usług na numery portów. Funkcjonalność ta korzysta z m.in. następujących plików konfiguracyjnych:
`/etc/services` - mapowanie nazwy usługi na numer portu
`/etc/hosts` - mapowanie nazw hostów na numery IP (lokalna baza)
`/etc/resolv.conf` - adresy serwerów DNS do których mogą być kierowane zapytania o adresy hostów

Poczta elektroniczna

Typowo system zapewnia obsługę (co najmniej lokalnego) dostarczania poczty elektronicznej. Do wysyłania listów można skorzystać z komend `mail` lub `sendmail`. Z kolei komenda `sendEmail` pozwala także na wysyłanie poczty przez wskazany serwer SMTP.

Planowanie zadań

Typowo system zapewnia usługę uruchamiania zadań o zadanym czasie. Z usługi tej można skorzystać przy pomocy poleceń:
`crontab` pozwala oglądać i edytować tablice zaplanowanych zadań cyklicznych (dla *cron'a*)
`at` pozwala jednorazowo zaplanować zadanie

Pliki konfiguracyjne *crona* / obsługiwane *crontab-em* mają postać: minuty godzina dzienMiesiaca miesiac

dzienTygodnia polecenie. Wpis oznacza że polecenie ma zostać wykonane jeżeli wszystkie warunki będą spełnione, jeżeli jakiś warunek nie jest nam potrzebny można użyć gwiazdki *, z kolei */n oznacza wykonywanie jeżeli dana wartość jest podzielna przez n. Np.:

* /20 3 * * 1 ls oznacza wykonanie komendy ls w każdy poniedziałek o godzinie 3:00 3:20 i 3:40

Standardowe wyjście, wyjście błędu oraz powiadomienie o niezerowym kodzie powrotu domyślnie są wysyłane na lokalny adres mailowy użytkownika będącego właścicielem danego conta. Niekiedy dostępny jest także anacron pozwalający na mniej precyzyjne planowanie zadań

Zdalny dostęp - SSH

Serwer SSH zapewnia zdalny, szyfrowany dostęp do komputera na którym jest uruchomiony z innych hostów. Usługa SSH pozwala na kopiowanie plików (sftp, scp, rsync over ssh), montowanie zdalnych systemów plików (sshfs), tunelowanie protokołu X'ów (serwera środowiska graficznego, dzięki czemu można używać zdalnych aplikacji z GUI) oraz tworzenie szyfrowanych tuneli (nasłuchujących na połączenia przychodzące tak po stronie klienta ssh, serwera ssh, jak też pełnych tuneli VPN).

Podsumowanie

W kontaktach z technologią, byciu dobrym "komputerowcem", elektronikiem, ... najważniejsze wydaje się być **bawienie się technologią**, eksperymentowanie, łączenie i integrowanie różnych systemów i technologii, poznawanie nowego, zagłębianie się w szczegóły aby nie tylko wiedzieć jak to obsłużyć, ale też wiedzieć jak działa, traktowanie problemów jako wyzwań, ... oraz czerpanie z tego radości. Do określenia kogoś o takiej postawie, takim podejściu do techniki, najlepszym terminem byłby **haker**, pomimo iż jest to termin różnie rozumiany, u wielu osób wywołujący negatywne skojarzenia, wydaje się on jednak najbliższy takiej postawie życiowej.

Zajmowanie się technologią (a szczególnie z takim nastawieniem) oznacza tak naprawdę ciągłe uczenie się - jeżeli się nie rozwijasz, jeżeli zostajesz w miejscu to tak naprawdę się cofasz. Zatem już od samego początku poznawania świata elektroniki, programowania, komputerów, sieci komputerowych, itp., przy nauce tych dziedzin bardzo istotne (jeżeli nie najistotniejsze) wydaje się bawienie się tym, otwarte podejście do stawianych problemów, **eksperymentowanie** i samodzielne szukanie rozwiązań.

Nie ma większego sensu uczenia się szczegółów, konkretnych implementacji, itp na zapas. Jeżeli możemy sobie tylko na to pozwolić warto uczyć się na "żywych" zastosowaniach (rozwiązując problemy, które gdzieś w jakiś sposób samoistnie się pojawiły), przy takiej nauce warto też pamiętać o ogromnych zasobach Internetu. Przy poznawaniu obsługi bądź konfiguracji jakiegoś nowego narzędzia równie istotne jak zapoznanie się z dokumentacją, podręcznikiem jest wypróbowanie różnorodnych opcji, ich różnych kombinacji w praktyce, poszukiwanie takiego sposobu konfiguracji, użytkowania (takiego stylu) jaki jest najbardziej odpowiedni dla nas i dla rozwiązywanego problemu. W rozwiązywaniu konkretnych zagadnień najistotniejsze jest wiedzieć jak się do tego zabrać i umieć dać sobie radę.

Wydaje się że właśnie w tych dziedzinach często od wiedzy szczegółowej, ważniejsza jest szeroka wiedza ogólna (oraz oczywiście odpowiednie predyspozycje i chęci), wraz z umiejętnościami szukania informacji (i wiedzą gdzie należy jej szukać). Często nie ma nawet możliwości posiadania pełnej wiedzy na początku rozwiązywania jakiegoś problemu, istotniejsza jest ogólne zrozumienie "jak to działa", umiejętność korzystania z dokumentacji, wyszukiwania i sprawdzania szczegółów oraz umiejętność kombinowania / informatyczny spryt. Ważna jest także spora doza praktyki (rzeczywistość tworzy sytuacje które nie śniłyby się teoretykowi ...) i tak zwane obycie z techniką / technologią. Gdy napotka się jakiś problem należy próbować go rozwiązać, eksperymentować, szukać porady (ale raczej na zasadzie "że chcemy dostać wędkę a nie rybę") i nie żałować na to czasu.

Informacja cyfrowa

Podstawowym zadaniem chyba wszystkich systemów komputerowych i telekomunikacyjnych oraz ogromnej większości systemów elektronicznych jest przetwarzanie lub przekazywanie **informacji**. Obecnie jest to w znacznej większości informacja **cyfrowa** (w odróżnieniu od analogowej jest ona "skwantowana" - sygnał będący jej źródłem zmieniający się w sposób ciągły został poddany procesowi pomiaru i zamiany na liczbę o skończonej dokładności). Fakt dyskretyzacji zbioru wartości oraz (możliwe do wprowadzenia dzięki postaci liczbowej) sumy kontrolne umożliwiają wierne kopiowanie informacji. Dzięki wprowadzeniu tego dodatkowego matematycznego poziomu w zapisie informacji odrywamy się po części od ograniczeń naszej fizycznej rzeczywistości; "bity to nie atomy" i fizyka (w szczególności kwantowa) ich nie dotyczy.

Podstawową jednostką informacji jest **bit**, mogący przyjmować dwa rozróżnialne stany nazywane zazwyczaj **zero** i **jeden**, będący odpowiedzią typu tak/nie. Przy pomocy mniejszej ilości stanów nie jest możliwe przekazanie informacji (pozornie mogłoby wydawać się że niekiedy wystarczy jeden stan - np. podnosimy n razy chorągiewkę i w ten sposób przekazujemy liczbę n, jednak tutaj też jest ukryty drugi stan - chorągiewka opuszczona). Z stanami tymi łatwo łączyć odpowiednie stany napięciowe na wejściach/wyjściach układów elektronicznych - np. napięcie powyżej X voltów ("jest napięcie") oznacza 1, napięcie poniżej Y voltów ("brak napięcia") oznacza 0. Wydaje się że właśnie z tych dwóch powodów

system dwójkowy (mający tylko dwie cyfry - 0 i 1) wraz z **logiką dwuwartościową** zyskał tak duże zastosowanie w **elektronice** cyfrowej i komputerach.

Elektronika

Elektronika (zarówno analogowa jak i cyfrowa) zajmuje się zasadniczo przetwarzaniem informacji w postaci sygnałów elektrycznych. Podstawowymi pojęciami jest tutaj prąd (przepływ ładunku) oraz napięcie (**różnica potencjałów** czyli siła ten przepływ wymuszająca). Elementy elektroniczne podzielić możemy na elementy bierne (oporniki, kondensatory, cewki, diody) - wpływające na przepływ prądu w sposób nieregulowany oraz elementy aktywne (których działanie w odróżnieniu od biernych jest regulowane na drodze sygnałów elektrycznych). W elektronice (zwłaszcza cyfrowej) często dużo ważniejszym pojęciem niż prąd jest napięcie, a dokładniej nawet tylko potencjał w odniesieniu do umownego potencjału zerowego - masy (**GND**).

Dla przetwarzania danych przez współczesną elektronikę najważniejszym z podstawowych jej elementów wydaje się być **tranzystor** - jest to element o regulowanym poprzez przyłożenie odpowiedniego napięcia oporze elektrycznym (może być zatem wykorzystywany jako elektronicznie sterowany przełącznik). Obecnie większość tranzystorów wchodzących w skład urządzeń znajduje się we wnętrzach układów scalonych, liczących często tysiące czy nawet miliony tranzystorów.

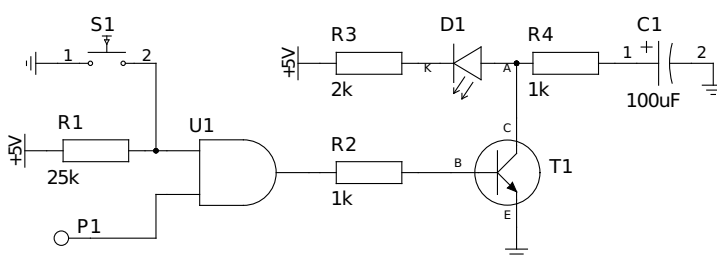
Podstawowymi dla techniki cyfrowej układami scalonymi są **bramki logiczne** realizujące poszczególne funkcje logiczne na przetwarzanych sygnałach oraz **rejestry** pozwalające na zapamiętanie stanu (jakiejś informacji). Jednak także i bramki ustąpiły już miejsca **układowi programowalnemu** złożonym z tysięcy tranzystorów, bramek i tym podobnych elementów - np. programowalnym układom logicznym, procesorom czy też mikrokontrolerom (zawierającym także pamięć zmienną - dla danych programu oraz pamięć do przechowywania samego programu); sprowadzając w zasadzie elektronikę opartą na elementach dyskretnych do roli pomocniczej.

Właśnie niczym mniej ani więcej tylko takim programowalnym urządzeniem elektronicznym, o zdolności do przetwarzania znacznych ilości informacji (w postaci liczb) w krótkim czasie (znacznej mocy obliczeniowej), jest współczesny **komputer**. Złożony on jest zasadniczo z procesora, pamięci operacyjnej (wraz z układem odpowiedzialnym za umożliwienie dostępu do niej procesorowi - kontrolerem pamięci) i pamięci stałej (dysk twardy itp.) służącej do przechowywania programów i danych. Niezależnie od roli jaką pełni (maszyna do pisanie czy grania, serwer WWW czy też serwer nadzorujący procesy w reaktorze jądrowym, ...) każdy komputer tylko i wyłącznie wykonuje jakiś **program**.

Poprzez powyższe twierdzenie, że komputer wykonuje tylko i wyłącznie program będący ciągiem instrukcji, nie twierdę że nie może istnieć coś takiego jak **sztuczna inteligencja**, a nawet sztuczna świadomość. Obecnie istnieją systemy zdolne do samodzielnej nauki na podstawie historycznych doświadczeń, istnieją systemy decyzyjne oraz wiele innych rozwiązań które w jakimś stopniu można by uważać za inteligentne. Z problemem sztucznej inteligencji związana jest głównie nieprecyzyjność tego terminu (istnienie wielu sprzecznych definicji). Wydaje się, że nawet człowiek nie jest w stanie przekonać innej osoby o tym że jest istotą inteligentną a tym bardziej świadomą gdy tamta przyjmie inne założenie.

W rozumieniu elektroniki ważna jest umiejętność przeanalizowania prostego układu, ustalenia / zrozumienia jak on działa.

Dla przykładu: obok przedstawiony jest układ wykorzystujący zarówno elementy cyfrowe, jak i analogowe do uzyskania jakiegoś efektu. Jaki jest cel działania tego układu? Jak on działa w szczególach?



1. U1 jest dwu wejściową bramką logiczną AND (na wyjściu zwraca stan logiczny 1 gdy na obu wejściach ma stan logiczny 1, któremu w tym wypadku odpowiada stan wysoki +5V), do jednego z wejść podłączony jest jakiś sygnał sterujący z zewnątrz (P1) do drugiego podłączony jest natomiast przełącznik S1 zwierający do masy.
2. Rezystor R1 podłączony do +5V zapewnia iż gdy przełącznik nie jest zwarty to do wejścia bramki przyłożony jest stan wysoki (niektóre z układów scalonych mają takie rozwiązanie zaimplementowane wewnętrznie), a gdy zwieramy S1 to ogranicza płynący prąd - w ten sposób na wejście bramki może być podane logiczne 1 lub zero bez konieczności stosowania przełączania i bez obawy o zrobienie zwarcia.
3. Wyjście bramki poprzez kolejny opornik (R2), który ogranicza prąd pobierany z wyjścia układu, steruje tranzystorem NPN T1 (przewodzi on gdy napięcie między bramką (B) a emiterym (E) przekroczy określoną wartość).
4. Wprowadzenie tranzystora w stan przewodzenia powoduje przepływ prądu przez LED D1 oraz rezystor R3, który ogranicza ten prąd do wartości odpowiedniej dla D1 (oczywiście także T1 musi

być dostosowany do wartości tego prądu); warto tu także zwrócić uwagę na polaryzację D1 (LED jak każda dioda przewodzi tylko w jedną stronę).

5. Kondensator C1 wraz z rezystorem R4 (a także R3) powoduje spowolnione wygaszanie D1 - kondensator rozładowuje się przez R4 gdy T1 przewodzi, natomiast po zaprzestaniu przewodzenia przez T1 kondensator ten ładuje się przez R3 i R4 podtrzymując przez pewien czas świecenie diody (dioda przygasa powoli, gdyż spada na niej napięcie w skutek procesu ładowania C1).

Programowanie

Program jest to ciąg instrukcji (głównie operacji matematycznych i logicznych), zapisany w języku zrozumiałym dla danego procesora (kod maszynowy powstający w wyniku kompilacji kodu stworzonego w języku wyższego poziomu). Wśród tych instrukcji szczególne miejsce zajmują instrukcje skoków (warunkowych i bezwarunkowych), one właśnie umożliwiają warunkową realizację fragmentów programu oraz warunkowe powtarzanie (**iterację**) fragmentów programu. Za szczególny przypadek instrukcji skoku można uważać instrukcję wywołania procedury, powodującą przekazanie stosownych informacji i rozpoczęcie wykonywania podprogramu (poprzez warunkowe wywoływanie tej samej procedury z jej wnętrza możemy otrzymać "konkurencyjny" do iteracji sposób powtarzania fragmentu programu - **rekurencję**).

Podstawą działania każdego programu jest jakiś **algorytm**, jest to pojęcie trochę bardziej abstrakcyjne niż program (zwłaszcza rozumiany jako kod maszynowy) - ogólnie mówiąc jest to skończony ciąg czynności prowadzących do zrealizowania jakiegoś zadania. Jak już wspomniałem oprócz kodu maszynowego (złożonego z numerów rozkazów i ich parametrów) są również języki programowania. Najbardziej podstawowym jest **assembler**, który pozwala używać łatwiej zrozumiałych dla człowieka nazw zamiast numerów instrukcji, rejestrów (w których procesor przechowuje wartości którymi operuje), Jednak jego instrukcje odpowiadają niemalże w 100% instrukcjom procesora (kodowi maszynowemu), dlatego też jest bardzo silnie zależny od danej architektury sprzętowej (modelu procesora), powoduje to problemy z przenoszeniem programów pisanych w assemblerze na inne architektury, to było jednym z powodów stworzenia **języków wyższego poziomu**. W językach takich treść algorytmu wyraża się przy pomocy poleceń danego języka będących bytami bardziej abstrakcyjnymi od instrukcji procesora (np. pojęcie zmiennej i operowanie na niej a nie na miejscach w pamięci i rejestrach). Obecnie assemblera używa się w zastosowaniach wymagających dużej wydajności (zarówno w oddzielnych aplikacjach jak i fragmentach większych programów) bądź dla urządzeń o niewielkich rozmiarach pamięci.

Telekomunikacja

Oprócz przetwarzania informacji liczne systemy zajmują się jej przekazywaniem. Aby mógł zająć proces komunikacji musi istnieć jakieś medium transmisyjne, które będzie odpowiedzialne za fizyczne przekazanie sygnału zawierającego informację. W **telekomunikacji** przekaz ten odbywa się zazwyczaj z wykorzystaniem zjawiska przepływu prądu przez przewodnik (tradycyjna łączność kablowa), bądź rozchodzenia się fali elektromagnetycznej (łączność radiowa, światłowody, ...). Bardzo istotne jest dopasowanie parametrów medium transmisyjnego, urządzeń nadawczych i odbiorczych do sygnału który zamierzamy transmitować, na przykład w przypadku kabli jest to częstotliwość którą są one w stanie przenosić bez zakłóceń (głównie ona odpowiada za szybkość transmisji i najczęściej jest rzędu kilkuset MHz) oraz impedancja (wyrażana w omach, sygnały które transmitujemy są sygnałami zmiennymi dlatego jest ona ważnym parametrem przewodu ...). Jednak pomimo to zawsze należy mieć świadomość, że to tylko kabel, który przewodzi elektrony i niezależnie od parametrów jakoś będzie to robił

W tradycyjnych systemach telekomunikacyjnych sygnał był bezpośrednio informacją, a jego dotarcie do odbiornika oznaczało że informacja jest przeznaczona dla niego (telefonii) bądź dla każdego (radio, telewizja - systemy rozgłoszeniowe). Obecnie coraz większą rolę odgrywają komutowane **sieci pakietowe**, gdzie informacja nie jest tak ściśle związana z sygnałem. Dla przykładu w telefonii zrealizowanie każdego połączenia wymagało zestawienia go - odpowiedniego przełączenia styków w centralach (komutacji łączy); w Internecie informacja podążać może różnymi drogami (nie jest tworzona droga specjalnie dla danego połączenia) i na podstawie adresu odnaleźć host docelowy (jest to komutacja pakietów). W sieciach pakietowych właściwa informacja (jeżeli jest zbyt długa) dzielona jest na części, do każdej części dodawane są informacje kontrolne - nagłówki (zawierający m.in. adres odbiorcy). Tak utworzone pakiety podróżują niezależnie od siebie przez sieć do odbiorcy (bez zestawiania w tym celu specjalnego fizycznego połączenia).

Bardzo istotnym elementem jest adresacja (jest ona niezbędna aby informacja mogła być kierowana do konkretnego hosta - inaczej mamy sieć rozgłoszeniową (np. telewizja)): każdy host w sieci musi posiadać unikalny (w ramach tej sieci) adres (chyba zawsze o ustalonej długości - nie jestem w stanie zagwarantować że ktoś, gdzieś nie stworzył protokołu pakietowego z adresami zmiennej długości), często wydzielą się również **podsieci**, posiadają one swój adres (na ogół jest to wspólna część adresu wszystkich hostów podsieci dopełniona zerami), adres określający ile adresów jest w podsieci () oraz adres na który reagować powinien każdy host w tej podsieci (adres rozgłoszeniowy). W przypadku chyba najpopularniejszych obecnie **sieci IP** wielość podsieci określana jest przez **maskę podsieci**, która po wykonaniu binarnego AND z dowolnym adresem hosta tej sieci da adres sieci. W związku z tym iż

binarnie maska składa się z nieprzerwanego ciągu jedynek oraz nieprzerwanego ciągu zer, jest ona często zapisywana jest w postaci prefixowej, czyli liczby bitów mających wartość jeden. W sieciach IP (oprócz samego Internet Protocol - odpowiedzialnego za adresację itp) należy także wspomnieć o protokołach takich jak: ICMP - zapewniającym przesył komunikatów technicznych, oraz **UDP** czy TCP - umożliwiającym przesył danych poprzez sieć IP (odpowiedzialne one są m.in. za adresację usług na danym hoście poprzez numery portów, **TCP** czuwa także nad poprawnością transferu - kontroluje czy dane dotarły i czy dotarły nieuszkodzone). Protokół IP oprócz najbardziej dla niego typowej transmisji jeden do jednego (unicast) umożliwia transmisję rozgłoszeniową (broadcast - jeden do wszystkich w sieci) i multicast (jeden do wielu).

Oczywiście sieci teleinformatyczne to nie tylko IP - należy koniecznie wspomnieć o protokołach warstw wyższych - takich jak HTTP (sieć Web), SMTP (e-mail), ... - zapewniających realizację typowych usług sieciowych oraz standardach związanych z konkretną realizacją sprzętową sieci - np. ethernet (będący najpopularniejszym standardem sieci lokalnych - LAN). Ze sprzętową realizacją sieci wiąże się m.in. pojęcie takie jak **topologia sieci** - czyli zasada łączenia hostów (**gwiazda** - wszystkie do jednego punktu, drzewo - wiele gwiazd łączonych w gwiazdy, **magistrala** - od jednej linii każdy host, ...). Najpopularniejszą obecnie topologią (przekładającą się także na inne instalacje - np. elektryczną, gdzie jednak dalej istotna jest magistrala) jest topologia gwiazdy, gdzie mamy doczynienia z punktami dystrybucyjnymi (w profesjonalnym podejściu realizowanymi w oparciu o szafy krosownicze i specjalnie przeznaczone do tego celu pomieszczenia) od których odchodzą trasy kablowe. Przy realizacji systemów telekomunikacyjnych (i innych teleinformatycznych oraz elektrycznych) warto pamiętać o stosownej dokumentacji tak przebiegu tras kablowych, jak i samej instalacji oraz punktów dystrybucyjnych a także przejrzystości i opisanu samych instalacji. Warto tutaj także zwrócić uwagę na "nie informatyczne" bezpieczeństwo takich instalacji obejmujące takie zagadnienia jak zabezpieczenia przeciwpożarowe, przeciwprzepięciowe oraz monitoring i kontrolę dostępu do pomieszczeń przeznaczonych na wspomniane instalacje.

Oprogramowanie

Oczywiście aby się zajmować, bawić komputerami, elektroniką, sieciami telekomunikacyjnymi konieczna jest wiedza na temat samej obsługi komputerów - zwłaszcza systemów unix'owatych. W tych systemach poszczególne osoby pracują na własnych kontach (identyfikowanych loginem i chronionych hasłem), szczególnym przypadkiem jest konto super użytkownika "**root**" - służy ono do czynności administracyjnych i umożliwia zrobienie z systemem (niemalże) wszystkiego. Standardowo stosowany jest też znany z adresów e-mail zapis z małąpą - umożliwiający globalną identyfikację użytkownika - user@host.domena. Z użytkownikami związane są także uprawnienia - definiowane (w podstawowym wariacie) niezależnie dla właściciela, grupy i wszystkich innych w dziedzinach odczytu, zapisu i wykonania.

Powłoka tekstowa oparta jest na poleceniach (będących najczęściej osobnymi programami) do których przekazywane są opcje i argumenty (oddzielane są one - zarówno od siebie jak i od nazwy programu spacjami), opcje na ogół zaczynają się od myślnika (jednoliterowe) bądź dwóch myślników (wieloliterowe), ale są od tego wyjątki. Powłoka umożliwia także pracę w trybie wsadowym (nie interaktywnym) - jest to już język skryptowy sh/bash. Informacje o prawie każdym programie, większości plików konfiguracyjnych oraz funkcjach biblioteki standardowej C można znaleźć w podręczniku systemowym (polecenie **man**).

Epilog

Współczesna technika umożliwia korzystanie z komputera, a nawet tworzenie oprogramowania bez znajomości **niskopoziomowych mechanizmów** działania tych urządzeń. Wydaje się jednak, że aby zrozumieć współczesną technikę, być dobrym programistą, administratorem, ... poznanie tych mechanizmów wydaje się niezbędne (a na pewno jest bardzo pomocne) i jest też niewątpliwie bardzo przydatne gdy chcemy zrobić coś "bliżej sprzętu"

Przy robieniu czegokolwiek warto pamiętać o standardach i normach technicznych, one naprawdę są (w zdecydowanej większości) tworzone po to aby ułatwić wszystkim życie (jakby wyglądał Internet gdyby każdy portal używał własnych standardów e-mail i www?). Należy też tak tworzyć, aby to co zrobiliśmy, napisaliśmy było przejrzyste i zrozumiałe dla innych oraz (za parę miesięcy czy lat) dla nas samych - aby kod był czytelny, dobrze skomentowany, a cały projekt posiadał dobrą i szczegółową **dokumentację** (wiem że często się nie chce, ale naprawdę warto ...). Należy się także wystrzegać rozwiązań przekombinowanych, stanowiących przerost formy nad treścią (trzymać się reguły **KISS**). Należy pamiętać, że zanim zaczniemy coś pisać warto poszukać czy nie ma czegoś o podobnej funkcji, bądź nie ma jakiś **bibliotek**, które zrobią za nas połowę pracy - nie zawsze warto odkrywać Amerykę na nowo ;-).

Wszystkich czytelników zachęcam także do zapoznania się oprócz samego "vademecum" także z dolinkowanymi materiałami oraz przede wszystkim do dużej porcji obycia z technologią, samodzielnego zgłębiania wybranych zagadnień oraz dużej ilości **praktyki** - bo to właśnie ona w tym wszystkim jest najistotniejsza (a to niestety wymaga dużo czasu). Ważne jest aby robić to co się lubi - zawsze lepiej być dobrym sieciowcem z powołania niż kiepskim koderem (i na odwrót). Dokładny kierunek kształcenia nie jest bardzo istotny - lepiej żeby nie był zbyt odległy, ale i tak masy rzeczy trzeba się uczyć samemu.

Warto mieć szersze spojrzenie na branżę - to że specjalizujesz się np. w sieciach nie oznacza iż podstawy programowania (zarówno web-owego jak i systemowego) czy też elektroniki i innych pokrewnych zagadnień są zbędne - wręcz przeciwnie mogą być dodatkowym atutem (nie tylko w CV ale i w samej pracy). To czego się nauczysz jest Twoje, ale zdobytą wiedzę, doświadczenie warto notować (gdyż zdarza się ponownie odkrywać Amerykę i ze zdziwieniem stwierdzać "to ja się tym zajmowałem 10 lat temu???").

Literatura

zakres tematyczny	nazwa	autor	licencja	opis	link
Elektronika					
Elektronika	<i>Lessons In Electric Circuits</i>	Tony R. Kuphaldt	Design Science License	Rozbudowany podręcznik do elektroniki od podstaw	EN
Elektronika	<i>Electronics</i>	wikibooks.org	CC-SA	Podręcznik do elektroniki	EN
Podstawy elektroniki	<i>Electronics Fundamentals</i>	wikibooks.org	CC-SA	Podręcznik podstaw elektroniki	EN
Podstawy elektroniki	<i>Practical Electronics</i>	wikibooks.org	CC-SA	Przewodnik rozumienia i tworzenia obwodów elektronicznych	EN
Podstawy elektroniki	<i>Circuit Idea</i>	wikibooks.org	CC-SA	Zbiór wyjaśnień zasad działania popularnych obwodów	EN
Teoria obwodów	<i>Circuit Theory</i>	wikibooks.org	CC-SA	Podręcznik do teorii obwodów	EN
Teoria obwodów	<i>Metody analizy obwodów liniowych</i>	Wojciech Meller	CC-NC-SA	Podręcznik do teorii obwodów	PL
Elektronika analogowa	<i>Analogue Electronics</i>	wikibooks.org	CC-SA	Podręcznik elektroniki analogowej	EN
Elektronika cyfrowa	<i>Digital Circuits</i>	wikibooks.org	CC-SA	Podręcznik do elektroniki cyfrowej	EN
Systemy mikroprocesorowe	<i>Microprocessor Design</i>	wikibooks.org	CC-SA	Podręcznik projektowania systemów mikroprocesorowe	EN
Systemy mikroprocesorowe	<i>Embedded Systems</i>	wikibooks.org	CC-SA	Podręcznik do systemów wbudowanych	EN
Automatyka	<i>Control Systems</i>	wikibooks.org	CC-SA	Podręcznik do systemów kontroli i sterowania	EN
Telekomunikacja	<i>Communication Systems</i>	wikibooks.org	CC-SA	Podręcznik do systemów telekomunikacyjnych	EN
Podstawy informatyki					
Systemy operacyjne	<i>Pisanie OS</i>	Jarosław Pelczar i inni	CC-SA, oryginał: PD	Podręcznik przybliżający zagadnienia związane z działaniem i tworzeniem systemów operacyjnych	PL
Sterowniki urządzeń	<i>Linux Device Drivers, Third Edition</i>	Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman	CC-NC-SA	Podręcznik opisujący zagadnienia związane z tworzeniem sterowników urządzeń dla systemu Linux	EN

Algorytmy i struktury danych	<i>Algorithms</i>	wikibooks.org	CC-SA	Wprowadzenie do algorytmów	EN
Algorytmy i struktury danych	<i>Data Structures</i>	wikibooks.org	CC-SA	Wprowadzenie do struktur danych	EN
Algorytmy i struktury danych	<i>Algorithm Implementation</i>	wikibooks.org	CC-SA	Przykłady implementacji algorytmów	EN
Języki programowania					
C	<i>C Programming</i>	wikibooks.org	CC-SA	Podręcznik programowania w C	EN
C	<i>A Little C Primer</i>	wikibooks.org	CC-SA	Podręcznik programowania w C	EN
C++	<i>C++ reference</i>		CC-SA	Podręcznik referencyjny dla C i C++	EN
C++	<i>C++ Programming</i>	wikibooks.org	CC-SA	Podręcznik programowania w C++	EN
C i C++	<i>Programming Language Concepts Using C and C++</i>	wikibooks.org	CC-SA	Podręcznik wprowadzający do programowania	EN
Python	<i>Byte of Python / Ukąś Pythona</i>	Swaroop C H	CC-SA, przykłady: BSD	Podręcznik programowania w Pythonie	EN PL
Python	<i>Dive into Python / Zanurkuj w Pythonie</i>	Mark Pilgrim	GNU FDL	Podręcznik programowania w Pythonie	EN PL
Python	<i>How to Think Like a Computer Scientist: Learning with Python</i>	Peter Wentworth, Jeffrey Elkner, Allen B. Downey, Chris Meyers	GNU FDL	Podręcznik programowania w Pythonie	EN EN
Python	<i>Non-Programmer's Tutorial for Python 3</i>	wikibooks.org	CC-SA	Kurs programowania w Pythonie	EN
PHP	<i>PHP Programming</i>	wikibooks.org	CC-SA	Podręcznik programowania w PHP	EN
Bash	<i>Advanced Bash-Scripting Guide</i>	Mendel Cooper	PD	Podręcznik tworzenia skryptów w bashu	EN
AWK	<i>GAWK: Effective AWK Programming</i>	Arnold D. Robbins	GNU FDL	Podręcznik dla języka AWK w odmianie GNU	EN
AWK	<i>Opis języka AWK</i>	T. Przechlewski	CC-BY-SA type	Wprowadzenie do programowania w AWK	PL
AWK	<i>AWK Reference Card / Ściąga do AWK</i>	Arnold D. Robbins	CC-BY-SA type	Karta referencyjna dla języka AWK	EN PL
AWK	<i>An Awk Primer</i>	wikibooks.org	CC-SA	Podręcznik dla języka AWK	EN
JavaScript	<i>Programming Ajax</i>	wikibooks.org	CC-SA	Podręcznik technologii Ajax	EN

języki opisu sprzętu	<i>Programmable Logic</i>	wikibooks.org	CC-SA	Podręcznik programowania układów logicznych	EN
języki opisu sprzętu	<i>VHDL for FPGA Design</i>	wikibooks.org	CC-SA	Podręcznik programowania układów FPGA w VHDL	EN
języki opisu sprzętu	<i>VHDL quick reference card</i>	Qualis Design Corporation	CC-ND-SA type	Karta referencyjna VHDL	EN
Sieci komputerowe					
TCP/IP	<i>Linux Network Administrator's Guide, 2nd Edition</i>	Olaf Kirch, Terry Dawson	GNU FDL	Podręcznik opisujący zagadnienia związane z sieciami TCP/IP w systemie Linux	EN PL
TCP/IP	<i>Linux Network Administrator's Guide, 3rd Edition</i>	Olaf Kirch, Terry Dawson	CC-NC-SA	Podręcznik opisujący zagadnienia związane z sieciami TCP/IP w systemie Linux	EN
TCP/IP	<i>Computer network technologies and services</i>	Lorenzo David, Luca Ghio	CC-SA	Podręcznik opisujący zagadnienia związane z sieciami komputerowymi TCP/IP i usługami sieciowymi	EN EN
TCP/IP	<i>Routing protocols and architectures</i>	Luca Ghio	CC-SA	Podręcznik opisujący zagadnienia związane z routowaniem w sieciach komputerowych	EN EN
Ethernet (LAN)	<i>Local Area Network design</i>	Luca Ghio	CC-SA	Podręcznik opisujący zagadnienia związane z lokalnymi sieciami komputerowymi	EN EN
Telekomunikacja	<i>Communication Networks</i>	wikibooks.org	CC-SA	Podręcznik sieci komunikacyjnych (nie tylko komputerowych ale także radiowych, telewizyjnych, itp)	EN
Linux/BSD/Posix					
Polecenia systemów typu Posix	<i>Unix Toolbox</i>	Colin Barschel	CC-SA	Zbiór użytecznych poleceń systemów Unix/Linux/BSD	EN
Vim	<i>Ściągą do vi</i>	T. Przechlewski	CC-SA type	Karta referencyjna dla edytora vim	PL
Vim	<i>Ściągą do vi</i>	Laurent Grégoire	GNU GPL	Karta referencyjna dla edytora vim	EN
Debian	<i>Debian Reference</i>	Osamu Aoki	GNU GPL	Podręcznik Debiana	EN
Debian	<i>The Debian Administrator's Handbook</i>	Raphaël Hertzog, Roland Mas	CC-SA / GNU GPL	Podręcznik administratora Debiana	EN
Debian	<i>Debian GNU/Linux Reference Card</i>	W. Martin Borgert, Holger Wansing	GNU GPL	Karta referencyjna dla Debian GNU/Linux	EN PL
FreeBSD	<i>FreeBSD Handbook</i>		BSD-type	Podręcznik dla systemu FreeBSD	EN PL
Arch	<i>Arch Linux Wiki</i>		GNU FDL	Dokumentacja dla Arch Linux	EN

PLD Linux	<i>Podręcznik użytkownika, administratora i twórcy</i>		GNU FDL	Podręcznik dla PLD Linux Distribution	PL
Linux	<i>Linux Networking</i>	wikibooks.org	CC-SA	Podręcznik sieci w środowisku Linux	EN
Języki zapytań i oznaczeń					
SQL	<i>Structured Query Language</i>	wikibooks.org	CC-SA	Podręcznik do języka zapytań baz danych SQL	EN
wyrażenia regularne	<i>Regular Expressions</i>	wikibooks.org	CC-SA	Podręcznik wyrażeń regularnych	EN
XML	<i>XML Schema</i>	wikibooks.org	CC-SA	Podręcznik Schcemy XML	EN
LaTeX	<i>The Not So Short Introduction to LaTeX / Nie za krótkie wprowadzenie do systemu LaTeX</i>	Tobias Oetiker, Hubert Partl, Irene Hyna, Elisabeth Schlegl	GNU GPL	Podręcznik wprowadzający do systemu składu tekstu LaTeX	EN PL
LaTeX	<i>LaTeX</i>	wikibooks.org	CC-SA	Podręcznik wprowadzający do systemu składu tekstu LaTeX	EN
Inne, ogólne, wielotematyczne					
aparatura przemysłowa	<i>Lessons In Industrial Instrumentation</i>	Tony R. Kuphaldt	Design Science License	Rozbudowany podręcznik do aparatury przemysłowej od podstaw	EN
	<i>Wikipedia</i>		CC-SA	Otwarta, wolna, wielojęzyczna encyklopedia	EN PL
	<i>opcode.eu.org</i>	Robert Paciorek	BSD/MIT-type	Serwis internetowy poświęcony różnym zagadnieniom z elektroniki, programowania, sieci i instalacji budynkowych	PL

Licencja

Copyright (c) 2003-2020, Robert Ryszard Paciorek <rrp@opcode.eu.org>

To jest wolny i otwarty dokument/oprogramowanie. Redystrybucja, użytkowanie i/lub modyfikacja SĄ DOZWOLONE na warunkach licencji MIT.

This is free and open document/software. Redistribution, use and/or modify ARE PERMITTED under the terms of the MIT license.

The MIT License:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.