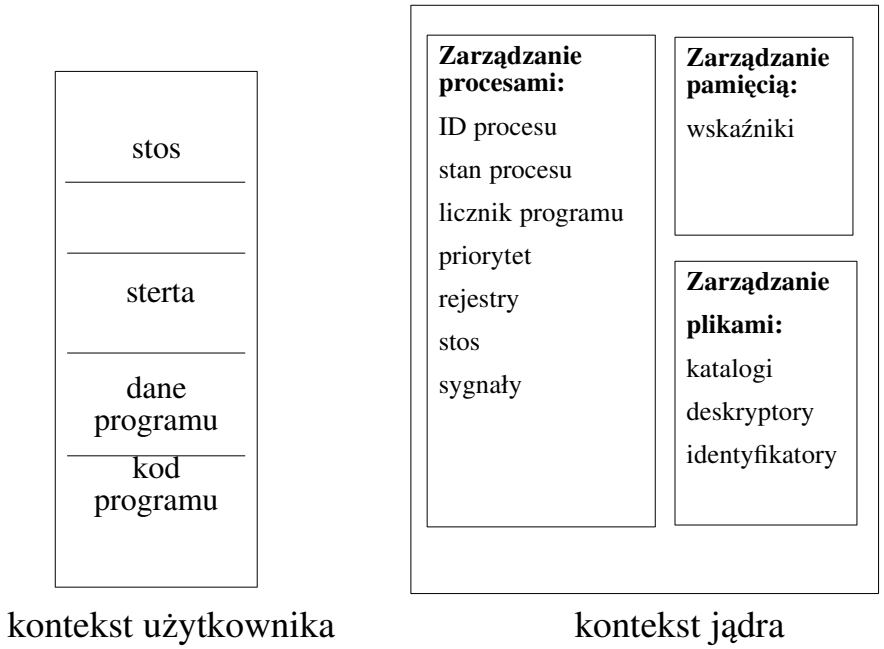
Podstawy programowania współbieżnego

IV

PODSTAWY TWORZENIA APLIKACJI WIELOPROCESOWYCH W SYSTEMIE LINUX

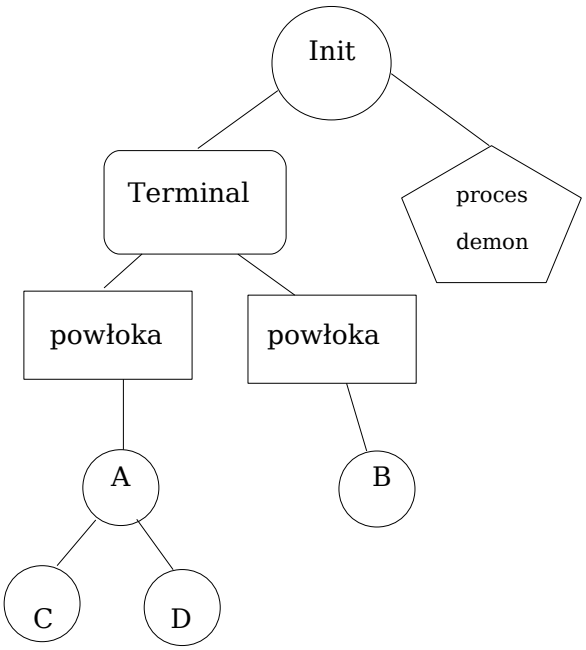
Proces unixowy

Proces jako elementarna jednostka systemu operacyjnego grupująca zasoby



Proces jest podstawową jednostką wykonawczą systemu operacyjnego pozwalającą wykonać program użytkownika. Zarządzanie procesami odbywa się na poziomie jądra systemu operacyjnego.

Organizacja procesów w systemie linux



Procesy są zorganizowane w strukturę drzewiastą. Rysunek pokazuje 9 procesów, których procesem pierwotnym jest proces **init**. Nowy proces jest tworzony przez proces już istniejący. Każdy proces ma swój unikalny numer (PID) w systemie.

Określenia:

Proces macierzysty – proces w relacji do procesu, którego utworzył.

Proces potomny – proces w relacji do procesu, który go utworzył

Procesy A, B są tworzone przy pomocy powłoki (shell'a) np. poprzez polecenie użytkownika, a więc powłoka jest ich procesem macierzystym. Procesy C,D są utworzone przez działający proces A, a więc są jego procesami potomnymi. Jeżeli proces A zakończy się przed procesami C i D to proces init będzie ich procesem macierzystym.

Terminal – proces, który pozwala kontrolować działanie innych procesów

Procesy uruchamiane przez użytkownika np. poleceniem shell'a posiadają terminal sterujący. Wysłanie z tego terminala np. sygnału zakończenia (CTRL_C) powoduje zwykle zakończenie procesu. Zamknięcie procesu terminala może skutkować zakończeniem wszystkich procesów, które są do niego przywiązane.

Proces demon - usługowy proces, który często działa na drugim planie (w tle).

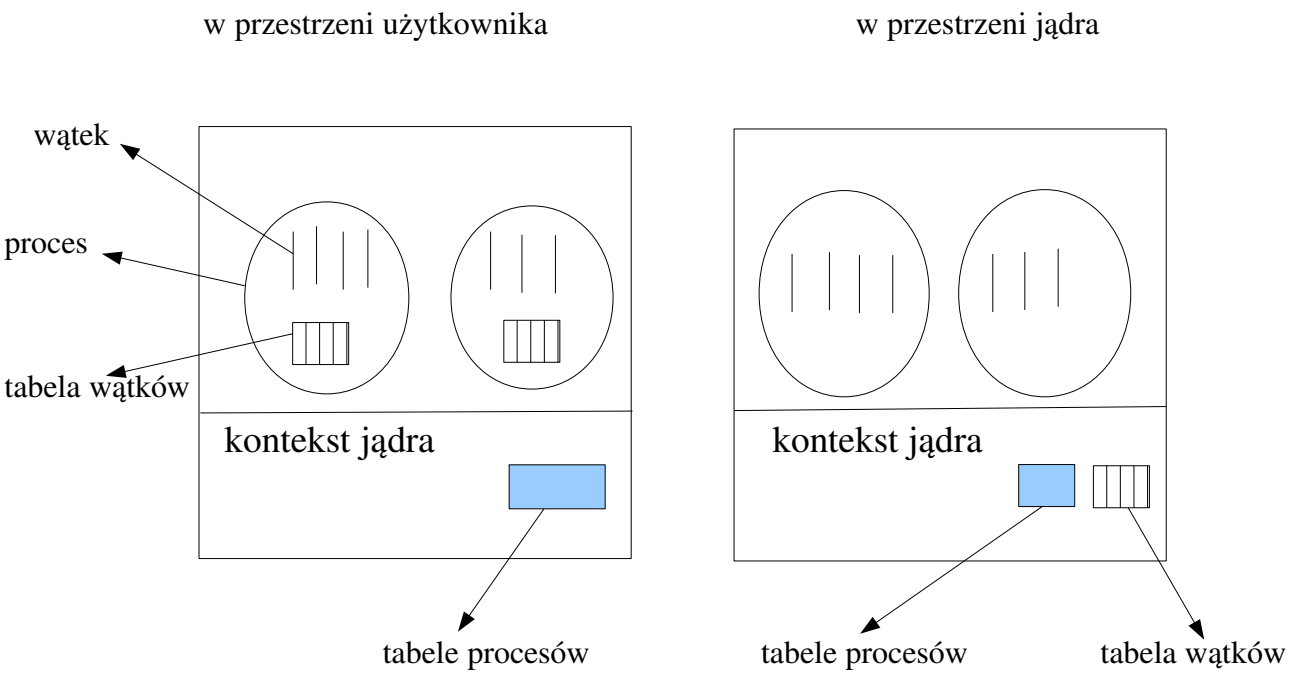
Procesy demony nie posiadają terminala sterującego. Zwykły proces może stać się demonem gdy odłączy się od terminala sterującego. Takie procesy są tworzone jako procesy potomne zwykłych procesów, a następnie ich procesy macierzyste kończą działanie. W tym czasie procesy potomne odłączają się od terminala i od grupy procesów, którą odziedziczyły po procesie macierzystym.

Procesy organizowane są w większe jednostki tj. grupy, sesje co ułatwia zarządzanie nimi. Jądro systemu zajmuje się zarządzaniem procesami wykorzystując sygnały oraz mechanizm przełączania kontekstu.

Wątki a procesy

Wątek jest częścią wykonawczą procesu. W jednym procesie może działać wiele wątków. Każdy wątek ma wtedy oddzielny stos ale wspólną pamięć główną (patrz. schemat procesu str. 2). Zarządzanie wątkami zależy od sposobu ich implementacji (patrz. Rys. poniżej). Wątki mogą być implementowane w przestrzeni użytkownika lub w przestrzeni jądra. Jeżeli wątki są implementowane w przestrzeni jądra ich mechanizm zarządzania jest podobny do zarządzania procesami. Dlatego często te wątki nazywa się lekkimi procesami. Takie właśnie wątki są tworzone przy pomocy biblioteki *Pthread*. Więcej na temat implementacji wątków można znaleźć w książce: *Systemy operacyjne A.S. Tannenbaum.*

Implementacje wątków



Rysunek: Systemy operacyjne A.S. Tannenbaum , H. Bos

Aplikacja wielowątkowa czy wieloprocesowa

Programy współbieżne mogą być tworzone jako programy wielowątkowe lub wieloprocesowe. Oto porównanie głównych cech aplikacji wieloprocesowych i wielowątkowych:

- Tworzenie nowego procesu trwa dłużej niż wątku.
  To ma znaczenie gdy w aplikacji wątki lub procesy mają być tworzone dynamicznie, tzn. gdy ich ilość zmienia się w czasie działania aplikacji.

- Przełączanie miedzy wątkami jest szybsze niż między procesami.
  Programy wielowątkowe są zwykle efektywniejsze niż programy wieloprocesowe.

- Proces zajmuje więcej miejsca w pamięci niż wątek.
  Liczba działających wątków może znacznie przekraczać liczbę procesów.

- Wątki komunikują się za pośrednictwem wspólnej pamięci globalnej. Procesy wymagają do komunikacji systemowych narzędzi do wysyłania i odbierania komunikatów.
  Komunikacja wątkami jest efektywniejsza, ale programy wielowątkowe częściej wymagają synchronizacji niż programy wieloprocesowe.

Aplikacje wielowątkowe wykorzystuje się efektywnych programach działających w systemach wieloprocesorowych z pamięcią wspólną (model PRAM). Aplikacje wieloprocesowe wykorzystuje się w systemach wieloprocesorowych z pamięcią lokalną (model sieciowy). Przykładem aplikacji wieloprocesowej jest architektura klient-serwer. Tutaj też często wykorzystuje się podejście hybrydowe. Np. serwer jest procesem wielowątkowym.

Funkcje systemowe wspomagające tworzenie aplikacji wieloprocesowej w systemie linux

- Tworzenie procesu

int fork() - funkcja systemowa tworzy proces, który jest kopią procesu wywołującego funkcję *fork*. Proces wywołujący funkcję *fork* nazywa się procesem macierzystym, a utworzony nowy proces jest procesem potomnym. Funkcja *fork* kończy działanie po stworzeniu procesu potomnego i zwraca różne wartości w dwóch procesach:

- w procesie macierzystym jest to numer **PID procesu potomnego** lub **-1** (gdy wystąpi błąd)
- w procesie potomnym **0**

Przykład1:

```
int main() {
...
fork();
...
}
```

Uruchomienie tego programu np. poleceniem z powłoki shell'a spowoduje, że proces powłoki shell'a utworzy proces, który wykona funkcję *main*. Wykonywania instrukcji *fork()* powoduje powstanie nowego procesu, który jest kopią procesu wywołującego *fork*. Od tej chwili oba procesy macierzysty i potomny działają współbieżnie. Proces potomny zawsze dziedziczy stan po procesie macierzystym (wartości zmiennych, deskryptory plików itd. istniejące tuż przed wywołaniem funkcji fork). Jako oddzielny proces ma inny numer *PID* niż jego proces macierzysty.

Przykład2:

```
int main() {
…        // sekcja (1)

if ( fork() == 0 ) {
                    … // sekcja (2)
                    }
… // sekcja (3)
}
```

Powyższy program wykona się następująco. Sekcję (1) wykona proces macierzysty. Sekcję (2) wykona tylko proces potomny (dlaczego?). Sekcję (3) wykonają oba procesy.

ćwiczenie1:

```
int main() {
… // sekcja (1)
fork() ;
… // sekcja (2)
fork() ;
… // sekcja (3)
fork();
… // sekcja (4)
}
```

Ile procesów wykona sekcję: 1,2,3,4 ?

Użyteczne funkcje systemowe:

void exit (int status) - funkcja powoduje zakończenie procesu który ją wywołał

int wait (int * status) - wstrzymuje wykonanie procesu macierzystego do póki proces potomny nie zakończy się.

int getpid() - funkcja zwraca PID procesu

int getppid() - funkcja zwraca PID macierzystego

int exec(…) **-** funkcja pozwalająca załadować nowy kod programu z dysku do procesu wywołującego

Pytanie: Jak proces macierzysty może poznać numer PID swojego potomka?

Objaśnienie wait: Jeżeli proces wywoła funkcję *wait* to zostaje zatrzymany na czas dopóki jego proces potomny nie zakończy działania. Wtedy proces macierzysty wykona funkcję *wait* i będzie mógł odczytać stan zakończenia procesu potomnego, który jest przekazywany poprzez argument funkcji *wait*. Jądro systemu zwalnia zasoby zakończonego procesu dopiero wtedy, gdy stan jego zakończenia zostanie odczytany przez proces macierzysty. Działanie funkcji *wait* w programach wieloprocesowych jest podobne do funkcji *pthread_join* w programach wielowątkowych. Jądro systemu tylko wtedy usuwa zakończony proces z tabeli procesów i zwalnia jego zasoby, gdy zostanie odebrany sygnał zakończenia przez funkcję *wait*. Dlatego proces macierzysty, który działa dłużej od swojego potomka powinien wywołać funkcję *wait*. Do momentu wywołania funkcji *wait* proces potomny, który się zakończył funkcjonuje jako proces 'zombie' (status Z), co jest niekorzystne z punktu widzenia systemu. Co w przypadku jeżeli proces macierzysty zakończy działanie i nie wywoła funkcji *wait?* W tej sytuacji proces **init** adoptuje jego potomka i on odbiera sygnał o zakończeniu procesu. Niestety funkcja *wait* działa blokująco na proces co może być niepożądane w pewnych sytuacjach. Dlatego jest możliwość ignorowania przez proces stanu zakończenia jego potomka, która nie skutkuje powstawaniem procesów 'zombie'. W tym celu należy ustawić maskę dla sygnału SIGCLD (patrz. Sygnały).

Objaśnienie exec: Jest to grupa funkcji systemowych, które w parze z funkcją *fork* służą do stworzenia procesu wykonującego dowolny kod programu najczęściej zapisany na dysku. Należy pamiętać, że po poprawnym wywołaniu funkcji *exec* kod programu dziedziczony po procesie macierzystym zostaje zastąpiony innym kodem. Pozostają natomiast odziedziczone deskryptory plików np. standardowe wejście, wyjście.

ćwiczenie2:

```
#include <stdio.h>
#include <unistd.h>
 main() {

   printf("AAA\n");

   if ( fork() ) == 0 ) execlp("./prog","prog",NULL);

   printf ("BBB\n");

  return 0;

 }
```

Zakładając że polecenie shell'a *./prog* daje wydruk na ekranie w postaci: "*****"

a) jaki będzie wydruk działania powyższego programu.
b) jaki będzie wydruk programu, jeżeli usuniemy z dysku plik "*prog*"

uwagi:

1) procesy potomny i macierzysty mają wspólne:
   a) tekst programu
   b) identyfikator użytkownika
   c) identyfikator grupy procesów
   d) katalog roboczy i główny
   e) otwarte pliki przez proces macierzysty przed utworzeniem procesu potomnego

2) różnice
   a) identyfikator procesu i identyfikator procesu macierzystego
   b) kopie deskryptorów plików
   c) kopia segmentu danych (przestrzeń adresowa)

Komunikacja międzyprocesowa

- Modele komunikacji

Komunikacja synchroniczna - dwa procesy muszą się spotkać aby doszło do wymiany komunikatu. Następuje synchronizacja procesu nadawcy i procesu odbiorcy.

Przykład:

kanał c

p1: sekcja_lokalna	q1: sekcja_lokalna
p2: c ⇒ x	q2: c ⇐ y
p3:	q3:

Możliwe scenariusze:

a) p1-q1-p2-q2-p3-q3
b) p1-q1-q2-p2-p3-q3

Instrukcja q3 nie może się wykonać przed instrukcją p2, a instrukcja p3 przed instrukcją q2

Komunikacja asynchroniczna - dwa procesy nie muszą się spotkać aby doszło do wymiany komunikatu. Nie następuje synchronizacja procesu nadawcy i procesu odbiorcy.

Przykład:

kanał c

p1: sekcja_lokalna	q1: sekcja_lokalna
p2: c ⇒ x	q2: c ⇐ y
p3:	q3:

Dowolny przeplot jest możliwy

Komunikacja z adresowaniem symetrycznym – proces nadawcy i proces odbiorcy znają się wzajemnie

Komunikacja z adresowaniem niesymetrycznym – architektura klient-serwer. Proces klienta zna proces serwera, a proces serwera nie zna klienta.

Komunikacja bez adresowania – nadawca wysyła komunikat o określonym typie a proces odbiorcy odbiera komunikat o określonym typie.

Komunikacja jednokierunkowa – w czasie jednej sesji komunikaty są wysyłane lub odbierane zawsze przez ten sam proces. Tak jest zawsze w przypadku komunikacji asynchronicznej.

Komunikacja dwukierunkowa – w przypadku komunikacji synchronicznej komunikaty mogą płynąć w dwóch kierunkach. Na przykład gdy komunikat wymaga natychmiastowej odpowiedzi.

Metody komunikacji międzyprocesowej (IPC)

Rodzaje komunikacji	Przestrzeń nazw	Identyfikacja
1. potoki anonimowe	brak	deskryptor pliku
2. potoki nazwane	ścieżka	deskryptor pliku
3. kolejki komunikatów	klucz	identyfikator
4. pamięć wspólna	klucz	identyfikator
5. semafor	klucz	identyfikator
6. gniazda – dziedzina unixa	ścieżka	deskryptor pliku

SYGNAŁY

Sygnały są najprostszą formą komunikacji międzyprocesowej. Sygnał nie przekazuje żadnych danych, ale daje informację że zaszło zdarzenie. Sygnały są przykładem komunikacji asynchronicznej (proces do którego jest kierowany sygnał nie czeka na niego).

Źródła sygnałów:

- sprzęt (np. procesor)
- jądro systemu
- proces

Reakcja na sygnał:

- reakcja domyślna (związana z rodzajem sygnału)
- przechwycenie sygnału i własna reakcja
- przechwycenie sygnału i zignorowanie

Funkcje systemowe do komunikacji przy pomocy sygnałów

int kill (int pid, int nsyg) - wysyła sygnał do procesu lub grupy procesów

void (*)(int) signal (nsyg, void (*)(int))

- ustawia funkcję obsługi sygnału dla rodzaju sygnału *nsyg,* to znaczy ustawia funkcję która się wykona w reakcji na sygnał

- niektóre rodzaje sygnałów UNIXA

SIGKILL - zakończenie procesu (nie można go zignorować ani przechwycić)

SIGTERM - sygnał zakończenie procesu

SIGINT - sygnał przerwania (ctrl_C)

SIGTSTP - sygnał zawieszenia procesu (ctrl_Z)

SIGCONT - sygnał wznowienia procesu

SIGSTOP - bezwarunkowo zatrzymuje proces

SIGCLD - sygnał wysyłany do procesu macierzystego gdy zakończy się proces potomny

SIGALRM - sygnał generowany przez budzik (funkcja alarm(sek))

SIGUSR1,2 - sygnały definiowane przez użytkownika do komunikacji między procesami

Przykład3: Ustawienie obsługi sygnału przerwania *ctrl_c*

```
#include <signal.h>

void obsluga() {
puts("otrzymalem sygnal");
fflush(stdout);
}

main() {

signal (SIGINT, obsluga );

while (1) pause();          // oczekiwanie nieaktywne

return 0;
}
```

Uruchomienie programu spowoduje że proces będzie działał w nieskończoność, próba zakończenia poleceniem *ctrl_c* spowoduje jedynie komunikat "otrzymałem sygnał".

POTOKI

Potoki są narzędziem komunikacji między procesami w modelu scentralizowanym. Przesyłana informacja jest strumieniem bajtów. Jest to rodzaj pliku służący do komunikacji.

cechy komunikacji przez potoki:

- komunikacja jednokierunkowa asynchroniczna
- proces odbierający nie wie czy dane zostały wysłane po jednym bajcie czy większym blokiem (długość komunikatu nieokreślona)
- dane odczytane nie mogą być odczytane ponownie

Rodzaje potoków:

- potoki (łącza) anonimowe
- potoki (łącza) nazwane

Funkcje systemowe do komunikacji przy pomocy potoków

int pipe (int des[2]) - funkcja tworzy potok anonimowy

funkcja przekazuje w argumencie deskryptory potoku:
des[0] – do odczytu
des[1] – do zapisu
funkcja zwraca -1 w przypadku błędu

uwaga: Potok anonimowy służy do przesyłania strumieni bajtów pomiędzy procesami spokrewnionymi. Jest to rodzaj wewnętrznego pliku. Jeżeli komunikacja ma się odbywać pomiędzy procesami niespokrewnionymi to należy używać potoków nazwanych.

Przykład4: Schemat komunikacji między procesem macierzystym i potomnym poprzez potok anonimowy.

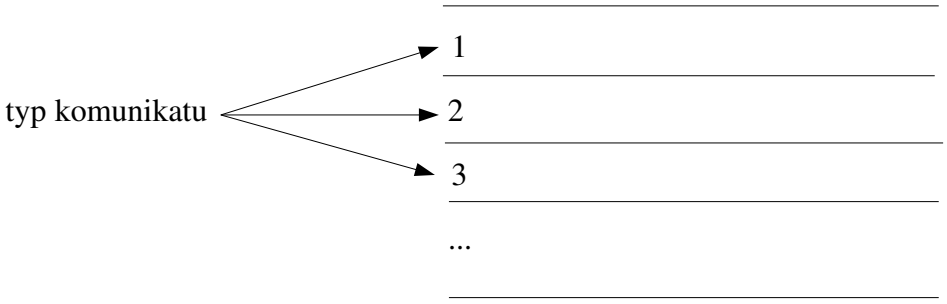
```
int main (int argc, char** argv) {
  int fds[2];
  int  proces;

  if (pipe(fds) < 0) blad("nieudany pipe()");          // tworzy potok i otrzymuje deskryptory w tablicy fds

  proces = fork();                                    // proces tworzy proces potomny który dziedziczy deskryptory potoku

  if (proces  == 0) {
        close(fds[0]);                                // proces potomny zamyka deskryptor do czytania

        double pi=3.14159;

        write(fds[1], &pi, sizeof(pi) );              // wysyła liczbę (jako strumień 8 bajtów) do potoku

     }

   if (proces >0) {                                   // proces macierzysty

        double d;
        close(fds[1]);                                // proces macierzysty zamyka deskryptor do pisania

        read( fds[0], &d, sizeof(d) );                // odczytuje z potoku strumień bajtów i zapisuje go u siebie w zmiennej d
        }


return 0;
}
```

Uwagi: Potok jest formą komunikacji między parą procesów. Komunikatem jest tutaj strumień bajtów. Wysyłanie i odbiór komunikatów przypomina zapis i odczyt z pliku. Proces powinien korzystać z dwóch różnych potoków w przypadku wysyłania i odbierania komunikatów. Jest gwarancja wzajemnego wykluczania na operacjach zapisu i odczytu.

Kolejka komunikatów

Kolejka komunikatów jest to mechanizm przesyłania komunikatów między procesami w systemie scentralizowanym. Komunikaty posiadają określony typ oraz długość.

model kolejki komunikatów :



wzorzec komunikatu:

```
struct {
 long    typ_komunikatu

 …      pola struktury  komunikatu

}
```

W kontekście jądra systemu kolejka komunikatu jest zapamiętywana jako lista powiązana komunikatów, gdzie komunikat najstarszy jest na początku listy. Kolejka komunikatów jest dostępna dla procesów przy pomocy unikalnego klucza i może istnieć nawet wtedy gdy procesy, które jej używały już nie istnieją. Kolejkę z systemu może usunąć proces, który zna klucz do niej.

Sposób komunikacja przy pomocy kolejki komunikatów:

a) proces przy pomocy klucza tworzy nową kolejkę lub uzyskuje dostęp do kolejki istniejącej
b) proces umieszcza komunikat w kolejce
c) proces pobiera komunikat określonego typu z kolejki

Cechy komunikacji:

a) kolejka komunikatów jest przetwarzana według schematu FIFO
b) komunikacja jest asynchroniczna
c) komunikacja odbywa się bez adresowania tzn. procesy komunikujące się nie znają.

Funkcje systemowe do komunikacji przy pomocy kolejek komunikatów

msgget	- tworzy kolekę komunikatów
msgsnd	- wysyła komunikat
msgrcv	- pobiera komunikat
msgctl	- wykonuje operacje sterujące na kolejce komunikatów (np. usuwa)

Przykład5: wysyłanie komunikatu

```
#define klucz 8000
#define prawa 0644

struct KOMUNIKAT {
    long typ;
    char tekst[100];
    };

[1]   int main (int argc, char*argv[]) {


[2]  int ident, dlugosc;
[3]  struct KOMUNIKAT  komunikat;

[4]  double tab[6]={2, 4, 1.5, 2.5, 3.5, 4.5 };

[5]   komunikat.typ = 5;  // określenie numeru podkanału w kolejce komunikatów

[6]   bcopy(tab, komunikat.tekst, 6*sizeof(double) );          // kopiuje 48 bajtów do tablicy tekst
[7]   dlugosc=sizeof(komunikat)-sizeof(long);

[8]  ident=msgget(klucz, prawa | IPC_CREAT )                  // proces tworzy kolejkę komunikatów

[9]  msgsnd(ident, &komunikat, dlugosc, 0 )                   // wysyła komunikat

}
```

uwaga: Struktura komunikatu musi zawierać pierwsze pole typu long. Pole to określa typ komunikatu. Pozostałe pola są dowolne i stanowią treść komunikatu.

Komunikatem w powyższym programie jest ciąg znaków, w którym jest zakodowane 6 liczb double. Kodowanie wykonuje instrukcja [6]. Alternatywne podejście polega na zdefiniowaniu struktury komunikatu typu:

```
struct KOMUNIKAT {
    long typ;
    double t[6];
    }
```

Instrukcja [7] oblicza długość komunikatu, a instrukcja [9] wysyła komunikat. Instrukcja [8] tworzy kolejkę komunikatów. Flaga IPC_CREAT wymusza utworzenie nowej kolejki komunikatów w przypadku, gdy stara kolejka nie istnieje. W przeciwnym razie pozostaje stara kolejka.

Komunikat w kolejce jest przechowywany tak długo dopóki inny proces go nie odbierze lub kolejka komunikatów nie zostanie usunięta. Zauważ że tutaj kolejka będzie istnieć, nawet gdy ten proces się zakończy.

Przykład6: odbieranie komunikatu

```
#define klucz 8000
#define prawa 0644

struct KOMUNIKAT {
    long typ;
    char tekst[100];
    };

[1] int main (int argc, char*argv[]) {


[2]    int ident, dlugosc;
[3]    struct KOMUNIKAT  komunikat;

[4]    double tab[6];

[5]     dlugosc=sizeof(komunikat)-sizeof(long);

[6]    ident=msgget( klucz, prawa  );          // tworzy kolejkę

[7]   if (ident == -1 ) return 1;                  // obsługa błedu

[8]    msgrcv(ident, &komunikat, dlugosc, 5, 0 );     // odczyt komunikatu z kolejki

[9]    bcopy( komunikat.tekst, tab,  6*sizeof(double) );

}
```

uwaga: instrukcja [6] tworzy identyfikator od istniejącej kolejki komunikatów. W przypadku, gdy kolejka nie istnieje zwracany jest kod błędu. Zauważ, że oba procesy (wysyłający i odbierający) używają jednakowego klucza (8000).

Struktura komunikatu musi taka sama jak w przypadku procesu wysyłającego. Instrukcja [8] służy do pobrania komunikatu z kolejki. Funkcja podbierająca komunikat określa w swoim czwartym argumencie typ komunikatu, na który czeka. Pobierany jest pierwszy komunikat określonego typu, który następnie zostaje usunięty z kolejki. Ostatni argument funkcji decyduje o jej zachowaniu, gdy odpowiedniego komunikatu nie ma w kolejce. Zero oznacza odbiór blokujący tzn. czekanie na komunikat. Użycie flagi IPC_NOWAIT powoduje zwrócenie kodu błędu (-1).