Mechanizmy komunikacji i synchronizacji

Witold Paluszyński
Katedra Cybernetyki i Robotyki
Wydział Elektroniki
Politechnika Wrocławska
http://www.kcir.pwr.wroc.pl/~witold/

2017

Procesy

Procesy umożliwiają w systemach operacyjnych (quasi)równoległe wykonywanie wielu zadań. Mogą być wykonywane naprzemiennie i/lub jednocześnie.

Zatem system operacyjny musi to (quasi)równoległe wykonywanie zapewnić.

Poza tym musi umożliwić procesom pewne operacje, których nie mogą one sobie zapewnić same, np. dostęp do globalnych zasobów systemu jak port komunikacyjny.

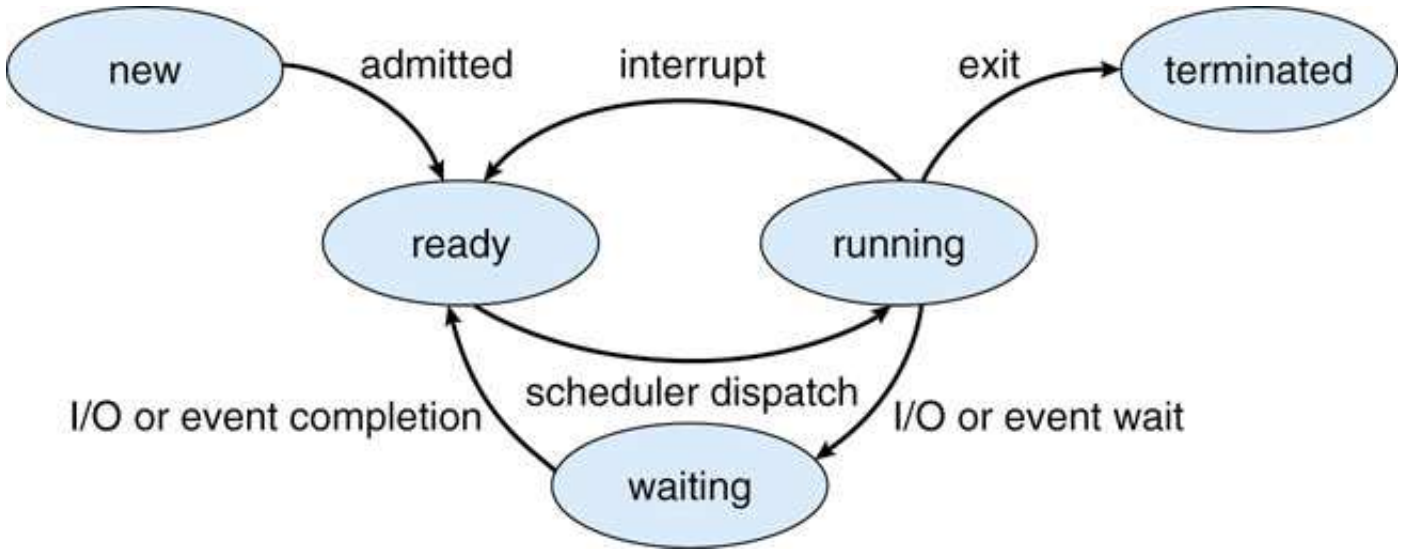
Idąc dalej, systemy operacyjne mogą również dostarczać procesom dalszych usług, jak np. komunikacja międzyprocesowa, a także pewnych funkcji synchronizacji, jak blokady, semafory, itp.

Proces uniksowy i jego środowisko

- Proces uniksowy jest wykonywalnym programem załadowanym do pamięci operacyjnej komputera. Każdy proces uniksowy wykonuje się we własnej wirtualnej przestrzeni adresowej.
- Proces jest normalnie tworzony z trzema otwartymi plikami: stdin, stdout i stderr przypisanymi do terminala użytkownika interakcyjnego.
- Jądro Uniksa utrzymuje tablicę wszystkich istniejących procesów wraz z zestawem informacji kontekstowych o nich, np. zestawem otwartych plików, terminalem sterującym, priorytetem, maską sygnałów, i innymi.
- Proces posiada **środowisko**, które jest zestawem dostępnych dla procesu zmiennych środowiskowych z wartościami (tekstowymi). Środowisko nazywa się zewnętrznym ponieważ jest początkowo tworzone dla procesu przez system, a potem jest również dostępne na zewnątrz procesu.
- Proces tworzony jest z zestawem argumentów wywołania programu, który jest wektorem stringów
- Proces charakteryzują: pid, ppid, pgrp, uid, euid, gid, egid

Stany procesów — model ogólny

Po pełnym zainicjalizowaniu procesu, jest on wielokrotnie (być może) przenoszony pomiędzy stanem gotowości (*ready*) i wykonywanym (*running*). Przenoszenie wykonywane jest przez system planowania (inaczej: szeregowania) systemu operacyjnego, który zarządza wykorzystaniem procesora (procesorów, lub rdzeni).



Okresowo proces może zażądać od systemu dostępu do jakichś zasobów, albo zainicjować operację wejścia/wyjścia. Jeśli to żądanie nie może być od razu zrealizowane (np. żądanych danych jeszcze nie ma), wykonywanie procesu musi zostać wstrzymane, i przechodzi on do stanu oczekiwania (*waiting*, w terminologii uniksowej jest to stan uśpienia). Gdy tylko możliwe jest wykonanie operacji I/O, lub spełnienie żądania procesu, zostaje on ponownie przeniesiony do kolejki gotowych.

Stany procesów — model Unix

stan	ang.	znaczenie
wykonywalny	*runnable*	proces w kolejce do wykonywania
uśpiony	*sleeping*	proces czeka na dostęp do zasobu
wymieciony	*swapped-out*	proces usunięty z pamięci
nieusuwalny	*zombie*	proces nie może się zakończyć
zatrzymany	*stopped*	wykonywanie wstrzymane sygnałem

- **Wymiatanie** procesów ma związek z funkcjonowaniem pamięci wirtualnej i jest objawem posiadania niewystarczającej ilości pamięci przez system. (Podstawowym mechanizmem odzyskiwania pamięci fizycznej jest **stronicowanie**). Wymiatanie polega na wytypowaniu pojedynczego procesu i chwilowym usunięciu go z kolejki procesów wykonujących się w celu odzyskania przydzielonej mu pamięci.

  Proces wymieciony jest normalnie w jednym z pozostałych stanów, lecz został usunięty z pamięci i aż do chwili ponownego załadowania nie może zmienić stanu.

- System przenosi proces w stan uśpienia jeśli nie może mu czegoś dostarczyć, np. dostępu do pliku, danych, itp. W stanie uśpienia proces nie zużywa czasu procesora, i jest *budzony* i wznawiany w sposób przez siebie niezauważony.

Tworzenie procesów

- Procesy tworzone są zawsze przez klonowanie istniejącego procesu, zatem każdy proces posiada tzw. proces nadrzędny, lub inaczej rodzicielski (*parent process*), który go utworzył. Wyjątkiem jest proces numer 1, utworzony przez jądro Uniksa, i wykonujący program `init`, który jest protoplastą wszystkich innych procesów w systemie.

- Proces potomny łączą ścisłe związki z jego procesem rodzicielskim. Potomek dziedziczy szereg atrybutów procesu nadrzędnego: należy do tego samego użytkownika i grupy użytkowników, ma początkowo ten sam katalog bieżący, otrzymuje środowisko, które jest kopią środowiska rodzica, ma tę samą sesję i terminal sterujący, należy do tej samej grupy procesów, dziedziczy maskę sygnałów i handlery, ograniczenia zasobów, itp.

- Poza dziedziczeniem atrybutów rodzica, potomek współdzieli z nim pewne zasoby, z których najważniejszymi są otwarte pliki. Objawy tego można łatwo zauważyć, gdy procesy uruchamiane przez interpreter poleceń mogą czytać z klawiatury i pisać na ekranie, a gdy jest kilka procesów uruchomionych w tle, to ich operacje wejścia/wyjścia mogą się mieszać. (Innymi współdzielonymi obiektami są otwarte wspólne obszary pamięci.)

Kończenie pracy procesów — status procesu

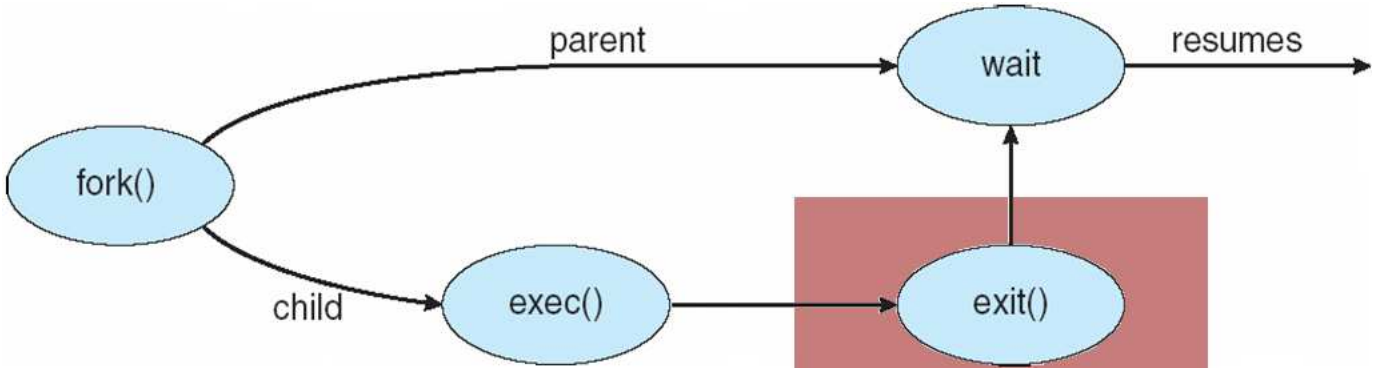
- Proces uniksowy kończy się normalnie albo anormalnie (np. przez otrzymanie sygnału). W chwili zakończenia pracy proces generuje kod zakończenia, tzw. **status**, który jest wartością zwracaną z funkcji `main()` albo `exit()`.
  W przypadku śmierci z powodu otrzymania sygnału statusem jest wartość 128+nr_sygnału.

- Po śmierci procesu jego rodzic normalnie powinien odczytać status potomka wywołując funkcję systemową `wait` (lub `waitpid`). Aby to ułatwić, w nowszych systemach w chwili śmierci potomka jego rodzic otrzymuje sygnał SIGCLD (domyślnie ignorowany).

  Dopóki status nie zostanie przeczytany przez rodzica, proces nie może zginąć do końca i pozostaje w stanie zwanym **zombie**. W tym stanie zakończony proces może pozostawać przez czas nieograniczony, co może być przyczyną wyczerpania jakichś zasobów systemu (np. zapełnienia tablicy procesów).

- Istnieje mechanizm adopcji polegający na tym, że procesy, których rodzic zginął przed nimi (sieroty), zostają adoptowane przez proces nr 1 (`init`). `init` wywołuje okresowo funkcję `wait` aby umożliwić poprawne zakończenie swoich potomków (zarówno naturalnych jak i adoptowanych).

Tworzenie procesów — funkcja `fork`




```
switch (pid = fork()) {
 case -1:
   fprintf(stderr, "Blad, nieudane fork, errno=%d\n", errno);
   exit(-1);

 case 0:
   /* jestem potomkiem, teraz sie przeobraze ... */
   execlp("program", "program", NULL);
   fprintf(stderr, "Blad, nieudane execlp, errno=%d\n", errno);
   _exit(0);

 default:
   /* jestem szczesliwym rodzicem ... */
   fprintf(stderr, "Sukces fork, pid potomka = %d\n", pid);
   wait(NULL);
}
```

Tworzenie procesów — funkcja `fork` (cd.)

- Funkcja `fork` tworzy podproces, który jest kopią procesu macierzystego.
- Od momentu utworzenia oba procesy pracują równolegle i kontynuują wykonywanie tego samego programu. Jednak funkcja `fork` zwraca w każdym z nich inną wartość. Poza tym różnią się identyfikatorem procesu PID (rodzic zachowuje oryginalny PID, a potomek otrzymuje nowy).
- Po sklonowaniu się podproces może wywołać jedną z funkcji grupy `exec` i rozpocząć w ten sposób wykonywanie innego programu.
- Po sklonowaniu się oba procesy współdzielą dostęp do plików otwartych przed sklonowaniem, w tym do terminala (plików `stdin`, `stdout`, `stderr`).
- Funkcja `fork` jest jedyną metodą tworzenia nowych procesów w systemach uniksowych.

Własności procesu i podprocesu

Dziedziczone (podproces posiada kopię atrybutu procesu):

- real i effective UID i GID, proces group ID (PGRP)
- kartoteka bieżąca i root
- środowisko
- maska tworzenia plików (umask)
- maska sygnałów i handlery
- ograniczenia zasobów

Wspólne (podproces współdzieli atrybut/zasób z procesem):

- terminal i sesja
- otwarte pliki (deskryptory)
- otwarte wspólne obszary pamięci

Różne:

- PID, PPID
- wartość zwracana z funkcji fork
- blokady plików nie są dziedziczone
- ustawiony alarm procesu nadrzędnego jest kasowany dla podprocesu
- zbiór wysłanych (ale nieodebranych) sygnałów jest kasowany dla podprocesu

Atrybuty procesu, które nie zmieniają się po exec:

- PID, PPID, UID(real), GID(real), PGID
- terminal, sesja
- ustawiony alarm z biegnącym czasem
- kartoteka bieżąca i root
- maska tworzenia plików (umask)
- maska sygnałów i oczekujące (nieodebrane) sygnały
- blokady plików
- ograniczenia zasobów

Sygnały

- Sygnały są mechanizmem asynchronicznego powiadamiania procesów przez system o błędach i innych zdarzeniach.

- Domyślną reakcją procesu na otrzymanie większości sygnałów jest natychmiastowe zakończenie pracy, czyli śmierć procesu. Oryginalnie sygnały miały służyć do sygnalizowania błędów i sytuacji nie dających się skorygować. Niektóre sygnały powodują również, tuż przed uśmierceniem procesu, wykonanie zrzutu jego obrazu pamięci do pliku dyskowego o nazwie `core`.

- Proces może zadeklarować własną procedurę obsługi, ignorowanie sygnału, albo czasowe wstrzymanie otrzymania sygnału (z wyjątkiem sygnałów SIGKILL i SIGSTOP, których nie można przechwycić, ignorować, ani wstrzymywać). Proces może również przywrócić reakcję domyślną.

- W trakcie wieloletniego rozwoju systemów uniksowych mechanizm sygnałów wykorzystywano do wielu innych celów, i wiele sygnałów nie jest już w ogóle związanych z błędami. Zatem niektóre sygnały mają inne reakcje domyślne, na przykład są domyślnie ignorowane.

nr	nazwa	domyślnie	zdarzenie
1	SIGHUP	śmierć	rozłączenie terminala sterującego
2	SIGINT	śmierć	przerwanie z klawiatury (zwykle: Ctrl-C)
3	SIGQUIT	zrzut	przerwanie z klawiatury (zwykle: Ctrl-\)
4	SIGILL	zrzut	nielegalna instrukcja
5	SIGTRAP	zrzut	zatrzymanie w punkcie kontrolnym (breakpoint)
6	SIGABRT	zrzut	sygnał generowany przez funkcję abort
8	SIGFPE	zrzut	nadmiar zmiennoprzecinkowy
9	SIGKILL	śmierć	bezwarunkowe uśmiercenie procesu
10	SIGBUS	zrzut	błąd dostępu do pamięci
11	SIGSEGV	zrzut	niepoprawne odwołanie do pamięci
12	SIGSYS	zrzut	błąd wywołania funkcji systemowej
13	SIGPIPE	śmierć	błąd potoku: zapis do potoku bez odbiorcy
14	SIGALRM	śmierć	sygnał budzika (timera)
15	SIGTERM	śmierć	zakończenie procesu
16	SIGUSR1	śmierć	sygnał użytkownika
17	SIGUSR2	śmierć	sygnał użytkownika
18	SIGCHLD	ignorowany	zmiana stanu podprocesu (zatrzymany lub zakończony)
19	SIGPWR	ignorowany	przerwane zasilanie lub restart
20	SIGWINCH	ignorowany	zmiana rozmiaru okna
21	SIGURG	ignorowany	priorytetowe zdarzenie na gniazdku
22	SIGPOLL	śmierć	zdarzenie dotyczące deskryptora pliku
23	SIGSTOP	zatrzymanie	zatrzymanie procesu
24	SIGTSTP	zatrzymanie	zatrzymanie procesu przy dostępie do terminala
25	SIGCONT	ignorowany	kontynuacja procesu
26	SIGTTIN	zatrzymanie	zatrzymanie na próbie odczytu z terminala
27	SIGTTOU	zatrzymanie	zatrzymanie na próbie zapisu na terminalu
30	SIGXCPU	zrzut	przekroczenie limitu CPU
31	SIGXFSZ	zrzut	przekroczenie limitu rozmiaru pliku

Mechanizmy generowania sygnałów

- wyjątki sprzętowe: nielegalna instrukcja, nielegalne odwołanie do pamięci, dzielenie przez 0, itp. (SIGILL, SIGSEGV, SIGFPE)
- naciskanie pewnych klawiszy na terminalu użytkownika (SIGINT, SIGQUIT)
- wywołanie komendy kill przez użytkownika (SIGTERM, i inne)
- funkcje `kill` i `raise`
- mechanizmy software-owe (SIGALRM, SIGWINCH)

Sygnały mogą być wysyłane do konkretnego pojedynczego procesu, do wszystkich procesów z danej grupy procesów, albo wszystkich procesów danego użytkownika. W przypadku procesu z wieloma wątkami sygnał jest doręczany do jednego z wątków procesu.

Obsługa sygnałów

Proces może zadeklarować jedną z następujących możliwości reakcji na sygnał:

- ignorowanie,
- wstrzymanie doręczenia mu sygnału na jakiś czas, po którym odbierze on wysłane w międzyczasie sygnały,
- obsługa sygnału przez wyznaczoną funkcję w programie, tzw. handler,
- przywrócenie domyślnej reakcji na dany sygnał.

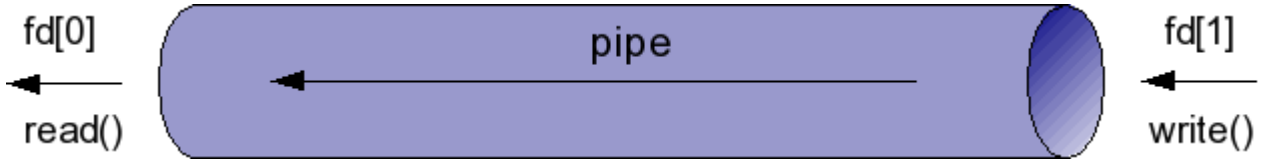
Typowa procedura obsługi sygnału przez funkcję handlera może wykonać jakieś niezbędne czynności (np. skasować pliki robocze), ale ostatecznie ma do wyboru:

- zakończyć proces,
- wznowić proces od miejsca przerwania, jednak niektórych funkcji systemowych nie można wznowić dokładnie od miejsca przerwania,
- wznowić proces od miejsca przerwania z przekazaniem informacji przez zmienną globalną,
- wznowić proces od określonego punktu.

Komunikacja międzyprocesowa — potoki

Potoki są jednym z najbardziej podstawowych mechanizmów komunikacji międzyprocesowej w systemach uniksowych. Potok jest urządzeniem komunikacji szeregowej, jednokierunkowej, o następujących własnościach:

- na potoku można wykonywać tylko operacje odczytu i zapisu, funkcjami `read` i `write`, jak dla zwykłych plików,
- potoki są dostępne i widoczne w postaci jednego lub dwóch deskryptorów plików, oddzielnie dla końca zapisu i odczytu,
- potok ma określoną pojemność, i w granicach tej pojemności można zapisywać do niego dane bez odczytywania,
- próba odczytu danych z pustego potoku, jak również zapisu ponad pojemność potoku, powoduje zawiśnięcie operacji I/O (normalnie), i jej automatyczną kontynuację gdy jest to możliwe; w ten sposób potok **synchronizuje** operacje I/O na nim wykonywane.

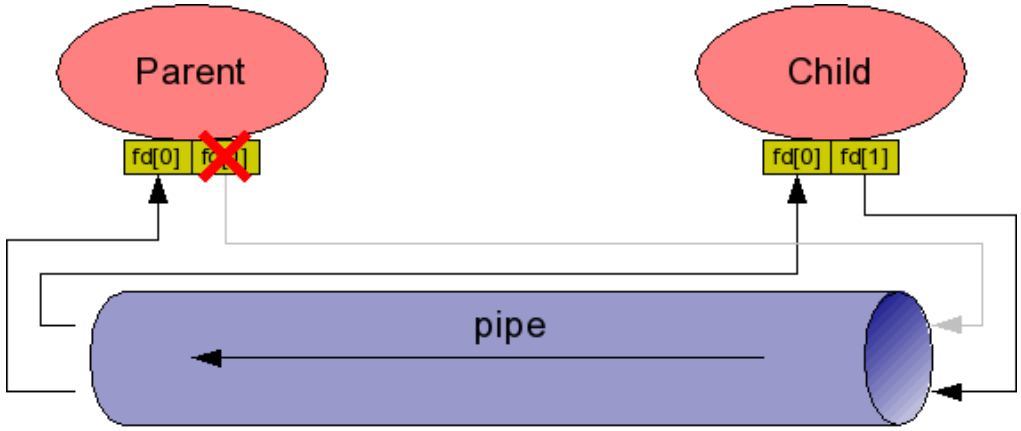


Dobrą analogią potoku jest rurka, gdzie strumień danych jest odbierany jednym końcem, a wprowadzany drugim. Gdy rurka się zapełni i dane nie są odbierane, nie można już więcej ich wprowadzić.

Potoki: funkcja `pipe`

Funkcja `pipe` tworzy tzw. „anonimowy” potok, dostępny w postaci dwóch otwartych i gotowych do pracy deskryptorów:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define KOM "Komunikat dla rodzica.\n"
int main() {
  int potok_fd[2], licz, status;
  char bufor[BUFSIZ];

  pipe(potok_fd);
  if (fork() == 0) {
    write(potok_fd[1], KOM, strlen(KOM));
    exit(0);
  }
  close(potok_fd[1]);  /* wazne */
  while ((licz=read(potok_fd[0], bufor, BUFSIZ)) > 0)
    write(1, bufor, licz);
  wait(&status);
  return(status);
}
```



Funkcja `read` zwraca 0 na próbie odczytu z potoku zamkniętego do zapisu, lecz jeśli jakiś proces w systemie ma ten potok otwarty do zapisu to funkcja `read` „zawisa” na próbie odczytu.

Potoki: zasady użycia

- Potok (anonimowy) zostaje zawsze utworzony otwarty, i gotowy do zapisu i odczytu.
- Próba odczytania z potoku większej liczby bajtów, niż się w nim aktualnie znajduje, powoduje przeczytanie dostępnej liczby bajtów i zwrócenie w funkcji `read()` liczby bajtów rzeczywiście przeczytanych.
- Próba czytania z pustego potoku, którego koniec piszący jest nadal otwarty przez jakiś proces, powoduje „zawiśnięcie" funkcji `read()`, i powrót gdy jakieś dane pojawią się w potoku.
- Czytanie z potoku, którego koniec piszący został zamknięty, daje natychmiastowy powrót funkcji `read()` z wartością 0.
- Zapis do potoku odbywa się poprawnie i bez czekania pod warunkiem, że nie przekracza pojemności potoku; w przeciwnym wypadku `write()` „zawisa" aż do ukończenia operacji.
- Próba zapisu na potoku, którego koniec czytający został zamknięty, kończy się porażką i proces piszący otrzymuje sygnał SIGPIPE.
- Implementacja potoków w większości współczesnych systemów uniksowych zapewnia komunikację dwukierunkową. Jednak standard POSIX jednoznacznie określa potoki jako jednokierunkowe.

Potoki nazwane (FIFO)

- istnieją trwale w systemie plików (`mknod potok p`)
- wymagają otwarcia `O_RDONLY` lub `O_WRONLY`
- zawisają na próbie otwarcia nieotwartego potoku (możliwe jest tylko jednoczesne otwarcie do odczytu i zapisu, przez dwa różne procesy)

SERWER:

```
#include <fcntl.h>
#define FIFO "/tmp/potok_1"
#define MESS \
  "To jest komunikat serwera\n"

void main() {
  int potok_fd;

  potok_fd = open(FIFO,
                  O_WRONLY);
  write(potok_fd,
        MESS, sizeof MESS);
}
```

KLIENT:

```
#include <fcntl.h>
#define FIFO "/tmp/potok_1"


void main() {
  int potok_fd, licz;
  char bufor[BUFSIZ];

  potok_fd = open(FIFO,
                  O_RDONLY);
  while ((licz=read(potok_fd,
                    bufor,
                    BUFSIZ)) > 0)
    write(1, bufor, licz);
}
```

Operacje na FIFO

- Pierwszy proces otwierający FIFO zawisa na operacji otwarcia, która kończy się gdy FIFO zostanie otwarte przez inny proces w komplementarnym trybie (`O_RDONLY`/`O_WRONLY`).
- Można wymusić nieblokowanie funkcji `open()` opcją `O_NONBLOCK` lub `O_NDELAY`, lecz takie otwarcie w przypadku `O_WRONLY` zwraca błąd.
- Próba odczytu z pustego FIFO w ogólnym przypadku zawisa gdy FIFO jest otwarte przez inny proces do zapisu, lub zwraca 0 gdy FIFO nie jest otwarte do zapisu przez żaden inny proces.

  To domyślne zachowanie można zmodyfikować ustawiając flagi `O_NDELAY` i/lub `O_NONBLOCK` przy otwieraniu FIFO. RTFM.
- W przypadku zapisu zachowanie funkcji `write` nie zależy od stanu otwarcia FIFO przez inne procesy, lecz od zapełnienia buforów. Ogólnie zapisy krótkie mogą się zakończyć lub zawisnąć gdy FIFO jest pełne, przy czym możemy wymusić niezawisanie podając opcje `O_NDELAY` lub `O_NONBLOCK` przy otwieraniu FIFO.
- Oddzielną kwestią jest, że dłuższe zapisy do FIFO ($\geq$`PIPE_BUF` bajtów) mogą mieszać się z zapisami z innych procesów.

Potoki — podsumowanie

Potoki są prostym mechanizmem komuniacji międzyprocesowej opisane standardem POSIX i istniejącym w wielu systemach operacyjnych. Pomimo iż definicja określa potok jako mechanizm komunikacji jednokierunkowej, to wiele implementacji zapewnia komunikację dwukierunkową.

Podstawowe zalety potoków to:

- brak limitów przesyłania danych,
- synchronizacja operacji zapisu i odczytu przez stan potoku.
  Praktycznie synchronizuje to komunikację pomiędzy procesem, który dane do potoku zapisuje, a innym procesem, który chciałby je odczytać, niezależnie który z nich wywoła swoją operację pierwszy.

Jednak nie ma żadnego mechanizmu umożliwiającego synchronizację operacji I/O pomiędzy procesami, które chciałyby jednocześnie wykonać operacje odczytu, lub jednocześnie operacje zapisu. Z tego powodu należy uważać potoki za mechanizm komunikacji typu 1-1, czyli jeden do jednego. Komunikacja typu wielu-wielu przez potok jest utrudniona i wymaga zastosowania innych mechanizmów synchronizacji.

Komunikacja międzyprocesowa — kolejki komunikatów

Kolejki komunikatów są mechanizmem komunikacji 1-1 o własnościach podobnych do potoków. Istnieje wiele wersji kolejek komunikatów z drobnymi różnicami.

Zasadnicza różnica pojawia się jednak ze względu na fakt, że w pewnych systemach kolejki komunikatów zostały zaimplementowane jako urządzenia sieciowe. Pozwalają one komunikować się procesom wykonującym się na różnych komputerach. Stanowią zatem mechanizm komunikacji międzyprocesowej dla systemów rozproszonych. W tych systemach takie kolejki komunikatów mogą stanowić bazowy mechanizm, na podstawie którego implementowane są mechanizmy komunikacyjne wyższego poziomu.

Komunikacja międzyprocesowa — gniazdka

Gniazdka są innym mechanizmem komunikacji dwukierunkowej typu 1-1 przeznaczonym zarówno do komunikacji międzyprocesowej w ramach jednego komputera, lub między procesami na różnych komputerach za pośrednictwem sieci komputerowej.

Zasadnicza różnica pomiędzy gniazdkami a potokami i kolejkami komunikatów polega na nawiązywaniu połączenia. Potoki i kolejki komunikatów są zawsze tworzone jako gotowe łącza komunikacyjne — posiadające dwa końce do których uzyskują dostęp procesy. Gniazdko jest jakby jednym końcem łącza komunikacyjnego. Przed rozpoczęciem komunikacji konieczne jest jego połączenie z innym gniazdkiem stworzonym na tym samym komputerze, lub innym, połączonym z nim siecią. Ze względu na ten proces nawiązywania połączenia, gniazdka wykorzystywane są na ogół w trybie komunikacji klient-serwer.

Gniazdka domeny Unix: podstawowe zasady

- Gniazdka są deskryptorami plików umożliwiającymi dwukierunkową komunikację w konwencji połączeniowej (strumień bajtów) lub bezpołączeniowej (przesyłanie pakietów), w ramach jednego systemu lub przez sieć komputerową, np. protokołami Internetu.
- Model połączeniowy dla gniazdek domeny Unix działa podobnie jak komunikacja przez potoki, m.in. system synchronizuje pracę komunikujących się procesów.
- Stosowanie modelu bezpołączeniowego w komunikacji międzyprocesowej nie jest odporne na przepełnienie buforów w przypadku procesów pracujących z różną szybkością; w takim przypadku lepsze jest zastosowanie modelu połączeniowego, w którym komunikacja automatycznie synchronizuje procesy.
- W komunikacji międzyprocesowej (gniazdka domeny Unix) adresy określane są przez ścieżki plików na dysku komputera, co umożliwia ogłoszenie adresu serwera.
- W ogólnym przypadku (wyjąwszy pary gniazdek połączonych funkcją `socketpair`) komunikowanie się za pomocą gniazdek wymaga utworzenia i wypełnienia struktury adresowej.

Gniazdka domeny Unix: funkcja `socketpair`

W najprostszym przypadku gniazdka domeny Unix pozwalają na realizację komunikacji podobnej do anonimowych potoków. Funkcja `socketpair` tworzy parę gniazdek połączonych podobnie jak funkcja `pipe` tworzy potok. Jednak gniazdka zawsze zapewniają komunikację dwukierunkową:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
  int gniazdka[2];  char buf[BUFSIZ];
  socketpair(PF_UNIX, SOCK_STREAM, 0, gniazdka)
  if (fork() == 0) {                    /* potomek */
    close(gniazdka[1]);
    write(gniazdka[0], "Uszanowanie dla rodzica", 23);
    read(gniazdka[0], buf, BUFSIZ)
    printf("Potomek odczytal: %s\n", buf);
    close(gniazdka[0]);
  } else {                                /* rodzic */
    close(gniazdka[0]);
    read(gniazdka[1], buf, BUFSIZ)
    printf("Rodzic odczytal %s\n", buf);
    write(gniazdka[1], "Pozdrowienia dla potomka", 24);
    close(gniazdka[1]);
  }
}
```

Gniazdka SOCK_STREAM: serwer

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/un.h>
#include <sys/socket.h>

int main() {
  int serv_sock, cli_sock;
  struct sockaddr_un addr_str;
  serv_sock = socket(PF_UNIX, SOCK_STREAM, 0);
  addr_str.sun_family = AF_UNIX;
  strcpy(addr_str.sun_path, "gniazdko_serwera");
  unlink("gniazdko_serwera");
  bind(serv_sock, (struct sockaddr *) &addr_str, sizeof(addr_str));
  listen(serv_sock, 5);                          /* kolejkowanie polaczen */
  while(1) {                                     /* petla oczek.na polaczenie */
    char buf;
    cli_sock = accept(serv_sock,0,0);            /* polacz.zamiast adr.klienta*/
    read(cli_sock, &buf, 1);                     /* obsluga klienta: */
    buf++;                                       /* ... generujemy odpowiedz */
    write(cli_sock, &buf, 1);                    /* ... wysylamy */
    close(cli_sock);                             /* ... koniec */
  }
}
```

Gniazdka SOCK_STREAM: klient

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/un.h>
#include <sys/socket.h>

int main() {
  int sock;
  struct sockaddr_un addr_str;
  char buf = 'A';

  sock = socket(PF_UNIX, SOCK_STREAM, 0);
  addr_str.sun_family = AF_UNIX;
  strcpy(addr_str.sun_path, "gniazdko_serwera");
  if (connect(sock, (struct sockaddr *) &addr_str, sizeof(addr_str)) == -1){
    perror("blad connect");
    return -1;
  }
  write(sock, &buf, 1);
  read(sock, &buf, 1);
  printf("znak od serwera = %c\n", buf);
  close(sock);
  return 0;
}
```

Gniazdka SOCK_DGRAM: klient

```
#include <unistd.h>
#include <string.h>
#include <sys/un.h>
#include <sys/socket.h>
int main() {
  int sock, serv_len, cli_len;
  struct sockaddr_un serv_addrstr, cli_addrstr;
  char ch = 'A';
  unlink("gniazdko_klienta");
  sock = socket(PF_UNIX, SOCK_DGRAM, 0);
  cli_addrstr.sun_family = AF_UNIX;
  strcpy(cli_addrstr.sun_path, "gniazdko_klienta");
  cli_len = sizeof(cli_addrstr);
  bind(sock, (struct sockaddr *) &cli_addrstr, cli_len);
  serv_addrstr.sun_family = AF_UNIX;
  strcpy(serv_addrstr.sun_path, "gniazdko_serwera");
  serv_len = sizeof(serv_addrstr);
  sendto(sock, &ch, 1, 0, (struct sockaddr *) &serv_addrstr, serv_len);
  if (recvfrom(sock, &ch, 1, 0, 0, 0) == -1)
    perror("blad recvfrom");
  else
    printf("znak od serwera = %c\n", ch);
  return 0;
}
```

Gniazdka SOCK_DGRAM: serwer

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/un.h>
#include <sys/socket.h>

int main() {
  int sock, serv_len, cli_len;
  struct sockaddr_un serv_addrstr, cli_addrstr;

  sock = socket(PF_UNIX, SOCK_DGRAM, 0);
  serv_addrstr.sun_family = AF_UNIX;
  strcpy(serv_addrstr.sun_path,"gniazdko_serwera");
  unlink("gniazdko_serwera");
  serv_len = sizeof(serv_addrstr);
  bind(sock,(struct sockaddr *)&serv_addrstr,serv_len);
  while(1) {                                  /* petla oczek.na pakiet */
    char ch;
    cli_len = sizeof(cli_addrstr);
    recvfrom(sock, &ch, 1, 0, (struct sockaddr *) &cli_addrstr, &cli_len);
    ch++;
    sendto(sock, &ch, 1, 0, (struct sockaddr *) &cli_addrstr, cli_len);
  }
}
```

Gniazdka — inne warianty komunikacji

- W przypadku komunikacji połączeniowej (gniazdka typu `SOCK_STREAM`) umożliwia to serwerowi związanie gniazdka z jakimś rozpoznawalnym adresem (funkcja `bind`), dzięki czemu serwer może zadeklarować swój adres i oczekiwać na połączenia, a klient może podejmować próby nawiązania połączenia z serwerem (funkcja `connect`).

- W przypadku komunikacji bezpołączeniowej (gniazdka typu `SOCK_DGRAM`) pozwala to skierować pakiet we właściwym kierunku (adres odbiorcy w funkcji `sendto`), jak również związać gniazdko procesu z jego adresem (`bind`), co ma skutek opatrzenia każdego wysyłanego pakietu adresem nadawcy.

- Możliwe jest również użycie funkcji `connect` w komunikacji bezpołączeniowej, co jest interpretowane jako zapamiętanie w gniazdku adresu odbiorcy i kierowanie do niego całej komunikacji zapisywanej do gniazdka, ale nie powoduje żadnego nawiązywania połączenia.

- Wywołanie funkcji `close` na gniazdku połączonym powoduje rozwiązanie połączenia, a w przypadku komunikacji bezpołączeniowej rozwiązanie związku adresu z gniazdkiem.

Mechanizmy komunikacji standardu POSIX Realtime

Istnieją mechanizmy komunikacji międzyprocesowej, analogiczne bądź podobne do System V IPC, wprowadzone w rozszerzeniu „realtime" standardu POSIX rozszerzenia Realtime IEEE 1003.1. Są to:

- kolejki komunikatów,
- pamięć współdzielona,
- semafory.

Pomimo iż ich funkcjonalność jest podobna do starszych i bardzo dobrze utrwalonych mechanizmów System V IPC, te nowe posiadają istotne zalety, przydatne w aplikacjach czasu rzeczywistego ale nie tylko w takich. Dlatego zostaną one tu przedstawione. Należy zwrócić uwagę, że nie wszystkie mechanizmy czasu rzeczywistego wprowadzone w standardzie POSIX są tu omówione, np. nie będą omawiane sygnały czasu rzeczywistego, timery, ani mechanizmy związane z wątkami, takie jak mutexy, zmienne warunkowe, ani blokady zapisu i odczytu.

Wszystkie mechanizmy komunikacji międzyprocesowej tu opisywane, opierają identyfikację wykorzystywanych urządzeń komunikacji na deskryptorach plików, do których dostęp można uzyskać przez identyfikatory zbudowane identycznie jak nazwy plików. Nazwy plików muszą zaczynać się od slash-a „/" (co podkreśla fakt, że mają charakter globalny), jednak standard nie określa, czy te pliki muszą istnieć/być tworzone w systemie, a jeśli tak to w jakiej lokalizacji. Takie rozwiązanie pozwala systemom, które mogą nie posiadać systemu plików (jak np. systemy wbudowane) tworzyć urządzenia komunikacyjne w swojej własnej wirtualnej przestrzeni nazw, natomiast większym systemom komputerowym na osadzenie ich w systemie plików według dowolnie wybranej konwencji.

Dodatkowo, semafory mogą występować w dwóch wariantach: anonimowe i nazwane. Jest to analogiczne do anonimowych i nazwanych potoków. Semafor anonimowy nie istnieje w sposób trwały, i po jego utworzeniu przez dany proces, dostęp do niego mogą uzyskać tylko jego procesy potomne przez dziedziczenie. Dostęp do semaforów nazwanych uzyskuje się przez nazwy plików, podobnie jak dla pozostałych urządzeń.

Urządzenia oparte o konkretną nazwę pliku zachowują swój stan (np. kolejka komunikatów swoją zawartość, a semafor wartość) po ich zamknięciu przez wszystkie procesy z nich korzystające, i ponownym otwarciu. Standard nie określa jednak, czy ten stan ma być również zachowany po restarcie systemu.

Kolejki komunikatów POSIX

Kolejki komunikatów standardu POSIX mają następujące własności:

- kolejka jest dwukierunkowym urządzeniem komunikacyjnym

  Kolejka może być otwarta w jednym z trybów: `O_RDONLY`, `O_WRONLY`, `O_RDWR`.

- stały rozmiar komunikatu

  Podobnie jak kolejki komunikatów System V IPC, a odmiennie niż potoki (anonimowe i FIFO), które są strumieniami bajtów, kolejki przekazują komunikaty jako całe jednostki.

- priorytety komunikatów

  Podobnie jak komunikaty System V IPC, komunikaty POSIX posiadają priorytety, które są jednak inaczej wykorzystywane. Nie ma możliwości odebrania komunikatu o dowolnie określonym priorytecie, natomiast zawsze odbierany jest najstarszy komunikat o najwyższym priorytecie.

  Taka funkcjonalność pozwala, między innymi, uniknąć typowego zjawiska inwersji priorytetów, gdzie komunikat o wysokim priorytecie może znajdować się w kolejce za komunikatem/ami o niższym priorytecie.

- blokujące lub nieblokujące odczyty

  Podobnie jak kolejki komunikatów System V IPC, kolejki POSIX posiadają zdolność blokowania procesu w oczekiwaniu na komunikat gdy kolejka jest pusta, lub natychmiastowego powrotu z kodem sygnalizującym brak komunikatu. Jednak w odróżnieniu od kolejek System V IPC, ta funkcjonalność jest dostępna dla kolejki jako takiej (wymaga jej otwarcia w trybie `O_NONBLOCK`) a nie dla konkretnych odczytów.

- powiadamianie asynchroniczne

  Kolejki komunikatów POSIX posiadają dodatkową funkcję pozwalającą zażądać asynchronicznego powiadomienia o nadejściu komunikatu do kolejki. Dzięki temu proces może zajmować się czymś innym, a w momencie nadejścia komunikatu może zostać powiadomiony przez:

  – doręczenie sygnału
  – uruchomienie określonej funkcji jako nowego wątku

  Rejestracja asynchronicznego powiadomienia jest dopuszczalna tylko dla jednego procesu, i ma charakter jednorazowy, to znaczy, po doręczeniu pojedynczego powiadomienia wygasa (ale może być ponownie uruchomiona). W przypadku gdy jakiś proces w systemie oczekiwał już na komunikat w danej kolejce, asynchroniczne powiadomienie nie jest generowane.

Kolejki komunikatów POSIX: mq_receive

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <mqueue.h>

#define MQ_TESTQUEUE "/mq_testqueue"
#define MODES 0666
#define MSGLEN 65536

int main() {
  mqd_t mqd;
  int len;
  unsigned int pri;
  char msg[MSGLEN];

  // na wszelki wypadek
  printf("Probuje usunac istniejaca kolejke komunikatow...\n");
  if(mq_unlink(MQ_TESTQUEUE) < 0)
    perror("nie moge usunac kolejki");
  else printf("Kolejka usunieta.\n");
```

```
  mqd = mq_open(MQ_TESTQUEUE, O_RDONLY|O_CREAT|O_NONBLOCK, MODES, 0);
  if (mqd == (mqd_t)-1) {
    perror("mq_open");
    exit(-1);
  }
  else printf("Kolejka komunikatow mqd = %d\n", mqd);

  printf("Czekam na dane ...\n");
  do {
    sleep(1);
    len = mq_receive(mqd, msg, MSGLEN, &pri);
    if (len >= 0)
      printf("Odebrany komunikat dlugosc %d: <%d,%s>\n", len, pri, msg);
    else perror("brak komunikatu");
  } while (0!=strncmp(msg, "koniec", MSGLEN));

  mq_close(mqd);
  return 0;
}
```

Kolejki komunikatów POSIX: mq_send

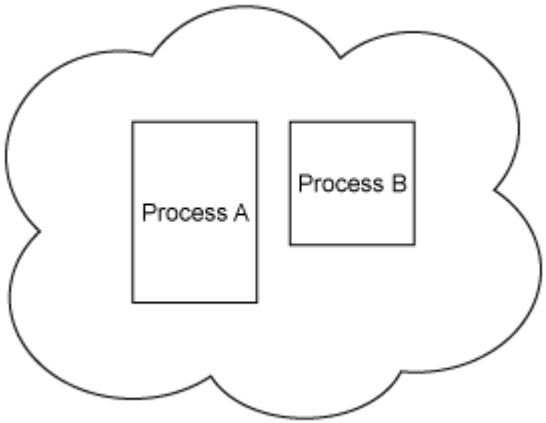
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <mqueue.h>
#include <sys/unistd.h>
#ifndef MQ_PRIO_MAX
 #define MQ_PRIO_MAX _SC_MQ_PRIO_MAX
#endif

#define MQ_TESTQUEUE "/mq_testqueue"
#define MODES 0666
#define MSGLEN 65536

int main() {
  mqd_t mqd;
  unsigned int pri;
  char msg[MSGLEN], buf[BUFSIZ], *charptr;

  mqd = mq_open(MQ_TESTQUEUE, O_WRONLY|O_CREAT|O_NONBLOCK, MODES, 0);
  if (mqd == (mqd_t)-1) { perror("mq_open"); exit(-1); }
  else printf("Kolejka komunikatow mqd = %d\n", mqd);
```

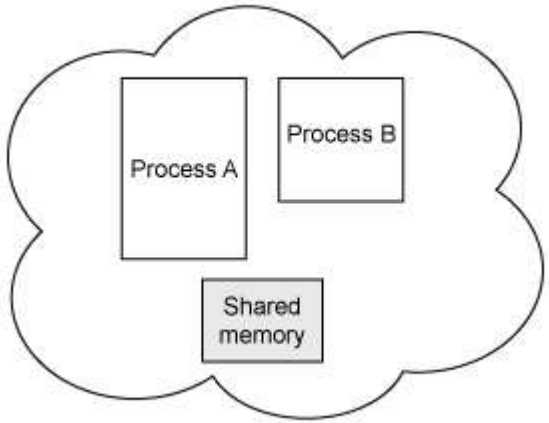
```
  do {
    printf("Podaj tresc komunikatu: ");
    fflush(stdout);
    fgets(msg, MSGLEN, stdin);
    charptr = strchr(msg, '\n');
    if (NULL!=charptr)
      *charptr = 0;
    printf("Podaj priorytet komunikatu: ");
    fflush(stdout);
    fgets(buf, BUFSIZ, stdin);
    sscanf(buf, "%i", &pri);
    if (pri<0) pri = 0;
    if (pri>MQ_PRIO_MAX) {
      printf("Wartosc priorytetu przekracza maksimum: %d\n", MQ_PRIO_MAX);
      pri = MQ_PRIO_MAX;
    }
    printf("Wysylany komunikat: <%d,%s>\n", pri, msg);

    if (mq_send(mqd, msg, strlen(msg)+1, pri) < 0)
      perror("blad mq_send");
    else printf("Poszlo mq_send.\n");
  } while (0!=strncmp(msg, "koniec", MSGLEN));
  mq_close(mqd);
  return 0;
}
```

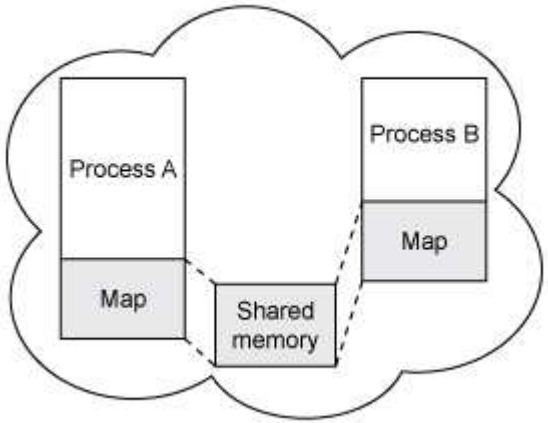
Pamięć współdzielona

Komunikacja przez pamięć wspólną jest najszybszym rodzajem komunikacji międzyprocesowej. Wymaga stworzenia obszaru pamięci wspólnej w systemie operacyjnym przez jeden z procesów, a następnie **odwzorowania** tej pamięci do własnej przestrzeni adresowej wszystkich pragnących się komunikować procesów.

Komunikacja w tym trybie wymaga synchronizacji za pomocą oddzielnych mechanizmów, takich jak muteksy albo blokady zapisu i odczytu.







Obrazki zapożyczone bez zezwolenia z:
`http://www.ibm.com/developerworks/aix/library/au-spunix_sharedmemory/`

Pamięć współdzielona: serwer

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>

#define POSIX_SOURCE
#define SHM_TESTMEM "/shm_testmem"
#define MODES 0666

struct shared_struct {
  int client_wrote;
  char text[BUFSIZ];
};

int main()
{
  struct shared_struct *shared_mem;
  int shmd, shared_size;

  // na wszelki wypadek
  printf("Probuje usunac istniejacy obszar wspolny...\n");
  if(shm_unlink(SHM_TESTMEM) < 0)
    perror("nie moge usunac obszaru pamieci");
  else printf("Obszar pamieci wspolnej usuniety.\n");
```

```
  shmd = shm_open(SHM_TESTMEM, O_RDWR|O_CREAT, MODES);
  if (shmd == -1) {
    perror("shm_open padlo");
    exit(errno);
  }

  shared_size = sizeof(struct shared_struct);
  ftruncate(shmd, shared_size);
  shared_mem = (struct shared_struct *)
               mmap(NULL,shared_size,PROT_READ|PROT_WRITE,MAP_SHARED,shmd,0);

  srand((unsigned int)getpid());
  shared_mem->client_wrote = 0;
  do {
    printf("Czekam na dane ...\n");
    sleep( rand() % 4 );        /* troche czekamy */
    if (shared_mem->client_wrote) {
      printf("Otrzymalem: \"%s\"\n", shared_mem->text);
      sleep( rand() % 4 );      /* znow troche poczekajmy */
      shared_mem->client_wrote = 0;
    }
  } while (strncmp(shared_mem->text, "koniec", 6) != 0);

  munmap((char *)shared_mem, shared_size);
  return 0;
}
```

Pamięć współdzielona: klient

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>

#define POSIX_SOURCE
#define SHM_TESTMEM "/shm_testmem"
#define MODES 0666

struct shared_struct {
    int client_wrote;
    char text[BUFSIZ];
};

int main()
{
  struct shared_struct *shared_mem;
  char buf[BUFSIZ], *charptr;
  int shmd, shared_size;

  shmd = shm_open(SHM_TESTMEM, O_RDWR|O_CREAT, MODES);
  if (shmd == -1) {
    perror("shm_open padlo");
    exit(errno);
```

```
  }

  shared_size = sizeof(struct shared_struct);
  ftruncate(shmd, shared_size);
  shared_mem = (struct shared_struct *)
               mmap(NULL,shared_size,PROT_READ|PROT_WRITE,MAP_SHARED,shmd,0);
  do {
    while(shared_mem->client_wrote == 1) {
      sleep(1);
      printf("Czekam na odczytanie...\n");
    }
    printf("Podaj text do przeslania: ");
    fgets(buf, BUFSIZ, stdin);
    charptr = strchr(buf, '\n');
    if (NULL!=charptr)
      *charptr = 0;

    strcpy(shared_mem->text, buf);
    shared_mem->client_wrote = 1;
  } while (strncmp(buf, "koniec", 6) != 0);

  munmap((char *)shared_mem, shared_size);
  return 0;
}
```

Semafory: teoria

W teorii semafor jest nieujemną zmienną (sem), domyślnie kontrolującą przydział pewnego zasobu. Wartość zmiennej sem oznacza liczbę dostępnych jednostek zasobu. Określone są następujące operacje na semaforze:

P(sem) — oznacza zajęcie zasobu sygnalizowane zmniejszeniem wartości semafora o 1, a jeśli jego aktualna wartość jest 0 to oczekiwanie na jej zwiększenie,

V(sem) — oznacza zwolnienie zasobu sygnalizowane zwiększeniem wartości semafora o 1, a jeśli istnieje(ą) proces(y) oczekujący(e) na semaforze to, zamiast zwiększać wartość semafora, wznawiany jest jeden z tych procesów.

Istotna jest niepodzielna realizacja każdej z tych operacji, tzn. każda z operacji P, V może albo zostać wykonana w całości, albo w ogóle nie zostać wykonana. Z tego powodu niemożliwa jest prywatna implementacja operacji semaforowych przy użyciu zmiennej globalnej przez proces pracujący w warunkach przełączania procesów.

Przydatnym przypadkiem szczególnym jest semafor binarny, który kontroluje dostęp do zasobu na zasadzie wyłączności. Wartość takiego semafora może wynosić 1 lub 0. Semafory binarne zwane są również muteksami (ang. *mutex* = *mut*ual *ex*clusion).

Semafory POSIX

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode,
                unsigned int value);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);

int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem,
                  const struct timespec *abs_timeout);

int sem_getvalue(sem_t *sem, int *sval);
```