Unix: komunikacja międzyprocesowa

Witold Paluszyński
witold.paluszynski@pwr.edu.pl
http://kcir.pwr.edu.pl/~witold/

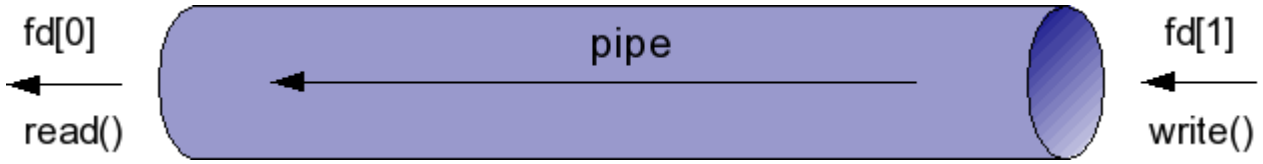
Copyright © 1999–2022 Witold Paluszyński
All rights reserved.

Niniejszy dokument zawiera materiały do wykładu na temat programowania komunikacji międzyprocesowej w systemie Unix. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Komunikacja międzyprocesowa — potoki

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

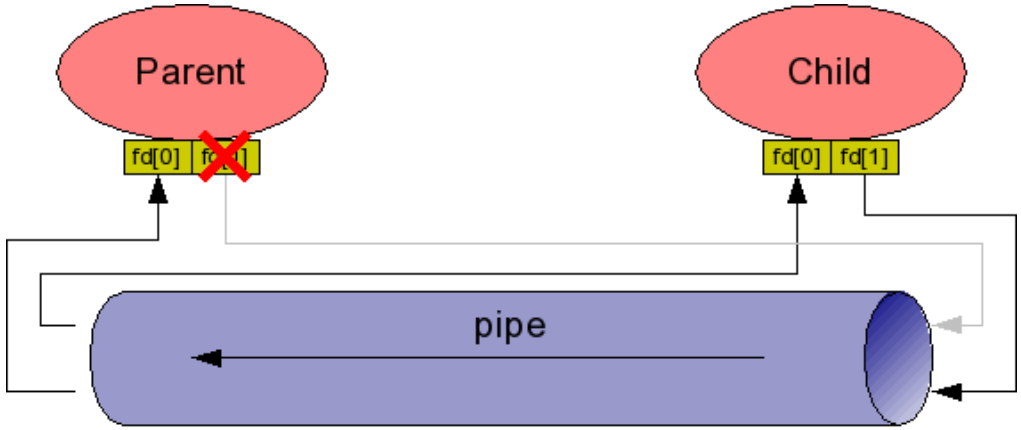
- na potoku można wykonywać tylko operacje odczytu i zapisu, funkcjami `read()` i `write()`, jak dla zwykłych plików,
- potoki są dostępne i widoczne w postaci jednego lub dwóch deskryptorów plików, oddzielnie dla końca zapisu i odczytu,
- potok ma określoną pojemność, i w granicach tej pojemności można 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:



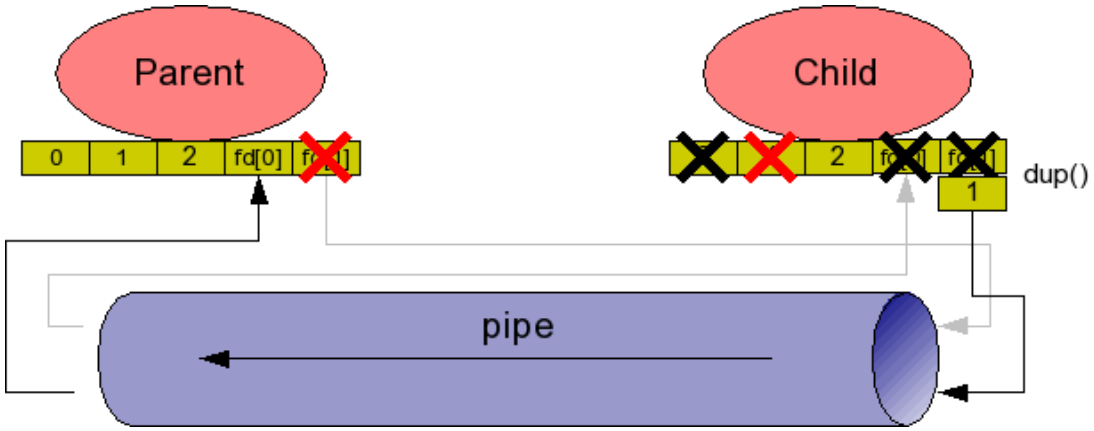

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define KOM "Komunikat dla rodzica.\n"
int main() {
  int potok_fd[2], licz, status;
  char bufor[BUFSIZ];

  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()` natychmiast zwraca 0 po próbie odczytu z pustego potoku gdy jest on zamknięty do zapisu, lecz gdy jakiś proces w systemie ma ten potok otwarty do zapisu, funkcja `read()` „zawisa” na próbie odczytu (proces przechodzi do stanu oczekiwania).

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`.
- Standard POSIX określa potoki jako jednokierunkowe. Jednak implementacja potoków większości współczesnych systemów uniksowych zapewnia komunikację dwukierunkową.

Potoki: przekierowanie standardowego wyjścia




```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int potok_fd[2], licz;  char bufor[BUFSIZ];

  pipe(potok_fd);
  if (fork() == 0) {          /* podproces tylko piszacy   */
    close(1);                 /* zamykamy stdout prawdziwy */
    dup(potok_fd[1]);         /* odzyskujemy fd 1 w potoku */
    close(potok_fd[1]);       /* dla porzadku */
    close(potok_fd[0]);       /* dla porzadku */
    close(0);                 /* dla porzadku */
    execlp("ps","ps","-fu",getenv("LOGNAME"),NULL);
  }
  close(potok_fd[1]);         /* wazne */
  while ((licz=read(potok_fd[0], bufor, BUFSIZ)) > 0)
    write(1, bufor, licz);
}
```

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.

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 domeny Unix: wprowadzenie

- 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.
- W ogólnym przypadku (wyjąwszy pary gniazdek połączonych funkcją `socketpair`) komunikowanie się za pomocą gniazdek wymaga utworzenia i wypełnienia struktury adresowej.
- W komunikacji międzyprocesowej (gniazdka domeny Unix) adresy określane są przez ścieżki plików na dysku komputera.
- Model połączeniowy (SOCK_STREAM) dla gniazdek domeny Unix działa podobnie jak komunikacja przez potoki, m.in. system synchronizuje pracę komunikujących się procesów.
- Stosowanie modelu bezpołączeniowego (SOCK_DGRAM) w komunikacji międzyprocesowej nie synchronizuje komunikujących się procesów, i nie jest odporne na przepełnienie buforów.

Gniazdka SOCK_DGRAM: serwer

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

#define GNIAZDKO_SERWERA "/tmp/gniazdko_serwera"

int main() {
  int x, y, n, sock;
  socklen_t serv_len, cli_len;
  struct sockaddr_un serv_addrstr, cli_addrstr;
  char buf[BUFSIZ];     /* bufor o domysl.rozm.*/

  /* przygotowanie gniazdka i struktury adres. */
  sock = socket(PF_UNIX, SOCK_DGRAM, 0);
  serv_addrstr.sun_family = AF_UNIX;
  strcpy(serv_addrstr.sun_path, GNIAZDKO_SERWERA);

  /* rejestracja adresu/utworzenie gniazdka */
  unlink(GNIAZDKO_SERWERA);
  serv_len = sizeof(serv_addrstr);
  if (-1==bind(sock,(struct sockaddr*)&serv_addrstr,
               serv_len)) {
    perror("blad bind");
    exit(-1);
  } /* if bind */

  /* nieskonczona petla obslugi klientow */
  printf("Serwer: rozpoczynam obsluge klientow\n");
  cli_len = sizeof(cli_addrstr);
  while(1) {
    n=recvfrom(sock, buf, sizeof(buf), 0,
               (struct sockaddr *)&cli_addrstr,
               &cli_len);
    if (n==-1) {
      perror("blad read");
      printf("Serwer: blad odczytu, kontynuuje.\n");
    } else {
      memcpy((void *)&x, (void *)buf, sizeof(int));
      y = x + 10;
      memcpy((void *)buf, (void *)&y, sizeof(int));
      sendto(sock, buf, sizeof(int), 0,
             (struct sockaddr*)&cli_addrstr,
             cli_len);
      printf("Serwer: odebral %d, wyslal: %d\n",
             x, y);
    } /* if recvfrom else */
  } /* while(1) */
} /* main() */
```

Gniazdka SOCK_DGRAM: klient

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

#define GNIAZDKO_SERWERA "/tmp/gniazdko_serwera"
#define GNIAZDKO_KLIENTA "/tmp/gniazdko_klienta_%d"

int main() {
  int i, j, n, sock, serv_len, cli_len;
  struct sockaddr_un serv_addrstr, cli_addrstr;
  char buf[BUFSIZ];     /* bufor o domysl.rozm.*/
  struct timespec timeout = {0, 100000000};

  /* przygotowanie gniazdka i struktur adres. */
  sock = socket(PF_UNIX, SOCK_DGRAM, 0);
  cli_addrstr.sun_family = AF_UNIX;
  sprintf(cli_addrstr.sun_path, GNIAZDKO_KLIENTA,
          getpid());
  unlink(cli_addrstr.sun_path);
  cli_len = sizeof(cli_addrstr);
  if (bind(sock, (struct sockaddr *)&cli_addrstr,
           cli_len)==-1) {
    perror("blad bind");
    return -1;
  } /* if bind */
  serv_addrstr.sun_family = AF_UNIX;
  strcpy(serv_addrstr.sun_path, GNIAZDKO_SERWERA);
  serv_len = sizeof(serv_addrstr);

  /* transmisja */
  for (i=0; i<100; ++i) {
    memcpy((void *)buf, (void *)&i, sizeof(int));
    sendto(sock, buf, sizeof(int), 0,
           (struct sockaddr *) &serv_addrstr,
           serv_len);
    n=recvfrom(sock, &buf, sizeof(buf), 0, 0, 0);
    if (n==-1) {
      perror("blad recvfrom");
      printf("Klient: wyslal %d, blad odcz.odpow.\n",
             i);
    } else {
      memcpy((void *)&j, (void *)buf, sizeof(int));
      printf("Klient: wyslal %d, odebral %d\n",i,j);
    } /* if recvfrom else */
    nanosleep((struct timespec *)&timeout,
              (struct timespec *)0);
  } /* for i=1..100 */
  close(sock);
  return 0;
} /* main() */
```

Gniazdka — zasady komunikacji bezpołączeniowej

serwer

```
s=socket(...); //utwórz gniazdko
struct sockaddr_un se_ad; //utw.włas.adres
bind(s,se_ad,len); //zarejestruj gniazdko serw
recvfrom(s,buf,...); //odbieraj pakiety
recv(s,buf,buflen,flags);
read(s,buf,buflen);
```

klient

```
s=socket(...); //utwórz gniazdko
//poznaj adres serwera
struct sockaddr_un se_ad;//utw.adres serw
sendto(s,buf,...,se_ad,l);//wysyłaj
```

Jeśli serwer nie potrzebuje odpowiedzieć klientowi, to nie musi poznawać jego adresu ani tworzyć żadnych struktur do tego. W takim przypadku serwer może odczytywać komunikaty z gniazdka uproszczoną funkcją `recv` a nawet zwykłą funkcją odczytu z deskryptora `read`. Jeśli jednak musi odpowiedzieć klientowi, to konieczne jest aby:

1. klient zarejestrował własny adres w systemie funkcją `bind`,
2. serwer utworzył strukturę adresową do zapisania adresu klienta,
3. serwer odebrał pakiet funkcją `recvfrom` umożliwiającą odczyt adresu, z którego ten pakiet przyszedł.

W komunikacji bezpołączeniowej każdy pakiet wysyłany jest razem z adresem odbiorcy, a jeśli adres nadawcy jest zarejestrowany w systemie funkcją `bind`, to również adresem nadawcy. Proces może się komunikować z różnymi partnerami przez to samo gniazdko.

Gniazdka — zasady komunikacji bezpołączeniowej (cd.)

W powyższych przykładach kryje się założenie, że jeden z programów (klient) zna adres serwera, i inicjuje komunikację. Serwer nie musi wcześniej znać adresu klienta, ponieważ otrzymuje go wraz z każdym otrzymanym pakietem. Ten sposób komunikacji ma szczególne znaczenie, gdy serwer jednocześnie obsługuje wielu klientów, otrzymuje od nich na przemian komunikaty, i w razie potrzeby odpowiada na nie właściwemu klientowi.

Jednak wcale nie musi tak być. Komunikacja bezpołączeniowa może być w pełni symetryczna, gdzie obydwa programy od początku znają swoje adresy. Wtedy każdy z nich może w dowolnym momencie wysłać pakiet partnerowi, albo odczytać pakiet odebrany, stosownie do ustalonego protokołu komunikacyjnego.

W przypadku takiej komunikacji dwukierunkowej programy serwera i klienta wykonują praktycznie takie same operacje i zasadniczo się od siebie nie różnią. Którego z nich nazwiemy wtedy serwerem, a którego klientem jest zatem kwestią umowną, i bardziej zależy od jego roli w protokole komunikacyjnym, niż od struktury programu.

Gniazdka SOCK_STREAM: klient

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

#define GNIAZDKO_SERWERA "/tmp/gniazdko_serwera"

int main() {
  int i, j, n, sock, addr_len;
  struct sockaddr_un addr_str;
  char buf[BUFSIZ];     /* bufor o domysl.rozm.*/
  struct timespec timeout = {0, 100000000};

  /* przygotowanie gniazdka i strukt.adresowej */
  sock = socket(PF_UNIX, SOCK_STREAM, 0);
  addr_str.sun_family = AF_UNIX;
  strcpy(addr_str.sun_path, GNIAZDKO_SERWERA);

  /* polaczenie */
  addr_len = sizeof(addr_str);
  if (connect(sock, (struct sockaddr *) &addr_str,
                    addr_len) == -1) {
    perror("blad connect");
    return -1;
  } /* if connect */

  /* transmisja */
  for (i=0; i<100; ++i) {
    memcpy((void*)buf,(void*)&i,sizeof(int));
    write(sock,(void *)buf,sizeof(int));  //wyslij
    n=read(sock,&buf,sizeof(buf));       //odbierz
    memcpy((void *)&j, (void *)buf, sizeof(int));
    printf("Klient: wyslal %d, odebral %d\n",i,j);
    nanosleep((struct timespec *) &timeout,
              (struct timespec *) 0);
  } /* for i=1..100 */
  close(sock);
  return 0;
} /* main() */
```

Gniazdka SOCK_STREAM: serwer iteracyjny

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

#define GNIAZDKO_SERWERA "/tmp/gniazdko_serwera"

int main() {
  int x, y, n, ser_sock, cli_sock, addr_len;
  struct sockaddr_un addr_str;
  char buf[BUFSIZ];     /*bufor o domysl.rozm.*/

  /* przygotowanie gniazdka i struktury adres.*/
  ser_sock = socket(PF_UNIX, SOCK_STREAM, 0);
  addr_str.sun_family = AF_UNIX;
  strcpy(addr_str.sun_path, GNIAZDKO_SERWERA);

  /* rejestracja adresu/utworzenie gniazdka */
  unlink(GNIAZDKO_SERWERA);
  addr_len = sizeof(addr_str);
  if (bind(ser_sock,(struct sockaddr*)&addr_str,
                    addr_len) == -1) {
    perror("blad bind");
    exit(-1);
  } /* if bind */

  if (listen(ser_sock,5)==-1) { /*oczek+dlug.kolej*/
    perror("blad listen");
    exit(-1);
  } /* if listen */

  /* nieskonczona petla obslugi klientow */
  while (1) {
    printf("Serwer: czeka na nowe polaczenie\n");
    cli_sock = accept(ser_sock,0,0);
    if (cli_sock==-1) {
      perror("blad accept");
      exit(-1);
    } /* if cli_sockt */
    /* petla obslugi polaczonego klienta */
    while ((n=read(cli_sock, &buf, sizeof(buf)))>0){
      memcpy((void *)&x, (void *)buf, sizeof(int));
      y = x + 10;
      memcpy((void *)buf, (void *)&y, sizeof(int));
      write(cli_sock, (void *)buf, sizeof(int));
      printf("Serwer: odebral %d, wyslal: %d\n",x,y);
    } /* while read */
    if (n==0) {
      printf("Serwer: klient zamknal polaczenie\n");
    } /* if read */
    close(cli_sock);
  } /* while(1) */
} /* main() */
```

Gniazdka SOCK_STREAM: serwer współbieżny

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

#define GNIAZDKO_SERWERA "/tmp/gniazdko_serwera"

int main() {
  int x, y, n, ser_sock, cli_sock, addr_len;
  struct sockaddr_un addr_str;
  char buf[BUFSIZ];     /*bufor o domysl.rozm.*/

  /* przygotowanie gniazdka i struktury adres.*/
  ser_sock = socket(PF_UNIX, SOCK_STREAM, 0);
  addr_str.sun_family = AF_UNIX;
  strcpy(addr_str.sun_path, GNIAZDKO_SERWERA);

  /* rejestracja adresu/utworzenie gniazdka */
  unlink(GNIAZDKO_SERWERA); /* nie moze istn. */
  addr_len = sizeof(addr_str);
  if (bind(ser_sock,(struct sockaddr*)&addr_str,
                   addr_len) == -1) {
    perror("blad bind"); exit(-1);
  } /* if bind */
  if (listen(ser_sock,5)==-1){/*ocz+dlug.kolej*/
    perror("blad listen"); exit(-1);
  } /* if listen */
  /* nieskonczona petla obslugi klientow */
  while (1) {
    printf("Serwer: czeka na nowe polaczenie\n");
    cli_sock = accept(ser_sock,0,0); /*ign.adr.kli.*/
    if (cli_sock==-1) {
      perror("blad accept"); exit(-1);
    } /* if cli_sock */
    if (fork()==0) {    /* potomek do obslugi kli. */
      close(ser_sock);  /* to jest gniazdko rodzica */
      while ((n=read(cli_sock,&buf,sizeof(buf)))>0) {
        memcpy((void *)&x, (void *)buf, sizeof(int));
        y = x + 10;
        memcpy((void *)buf, (void *)&y, sizeof(int));
        write(cli_sock, (void *)buf, sizeof(int));
        printf("Serwer: odebral %d, wyslal: %d\n",x,y);
      } /* while read */
      if (n==0) {
        printf("Serwer: klient zamknal polaczenie.\n");
      } /* if read */
      else {
        perror("blad read");
        printf("Serwer: blad odczytu, rozlacza kl.\n");
      } /* if read else */
      close(cli_sock);
      exit(0);
    } /* if fork()==0 -- koniec kodu potomka */
    close(cli_sock); /* tutaj rodzic */
  } /* while(1) */
} /* main() */
```

Gniazdka — zasady komunikacji połączeniowej

serwer	klient
`s=socket(...);` //utwórz gniazdko	`s=socket(...);` //utwórz gniazdko
`struct sockaddr_un se_ad;` //utw.włas.adres	//poznaj adres serwera
`bind(s,se_ad,len);` //zarejestruj gniazdko serw	`struct sockaddr_un se_ad;` //utw.adr.serw
`listen(s,5);` //deklaruj kolejkowanie połączeń	
`acc:`	
`s1=accept(s,...);` //oczekuj/przyjm.połączenie	`connect(s,se_ad,len);` //nawiąż połączenie
`read/write(s1,buf,len);` //komunikacja	`write/read(s,buf,buflen);` //komunikacja
`close(s1);` //zamknięcie połączenia	`close(s);` //koniec komunikacji z serwerem
`goto acc;`	`exit(0);`

W przypadku komunikacji połączeniowej, konieczność nawiązania połączenia wymusza kompletną asymetrię zachowań obu stron komunikacji. Serwer jest typowo stroną bierną, pierwszy rejestruje swój adres i oczekuje na nawiązanie z nim połączenia. Gdy to nastąpi, może rozpocząć komunikację, ale wcale nie musi znać adresu partnera/klienta, ponieważ komunikuje się z nim przez połączenie (gniazdko).

Ważne, że serwer uzyskuje połączenie za pomocą nowego gniazdka (s1), podczas gdy gniazdko oczekiwania na nowe połączenia pozostaje aktywne. Otwiera to możliwości implementowania różnych strategii obsługiwania przez serwery gniazdka oczekiwania na nowe połączenie, z istniejącym(i) gniazdkiem(gniazdkami) już nawiązanych połączeń (typowe schematy: serwer iteracyjny, serwer współbieżny wieloprocesowy).

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.

System V IPC

- Urządzenia komunikacyjne:
  - kolejki komunikatów: koniec–koniec
  - pamięć (współ)dzielona: wielu–wielu
  - semafory: liczby całkowite
- Mechanizmy System V IPC są zawarte w rozszerzeniu XSI standardu POSIX.
- Istnieją globalnie w systemie: polecenia `ipcs, ipcrm`.
- Nie są deskryptorami plików i nie wykonuje się na nich operacji I/O standardowymi funkcjami `read/write` — każde urządzenie ma swój specyficzny zbiór operacji I/O.
- Po utworzeniu danego urządzenia jest ono od razu gotowe do pracy, nie jest konieczne jego otwieranie przez każdy proces pragnący się komunikować.
  (Z wyjątkiem obszarów pamięci wspólnej, które każdy proces musi jeszcze odwzorować na swoją przestrzeń adresową.)
- Prawa dostępu do urządzeń: `RWRWRW`
- Identyfikatory urządzeń System V IPC (typu `int`) są globalne w systemie, inaczej niż deskryptory plików (i nie są kolejno generowanymi małymi liczbami). Oznacza to, że jeden proces może od razu użyć identyfikatora utworzonego przez inny proces.

  Wynika stąd możliwość „wkleszczania się”, celowego lub nie, w komunikację prowadzoną przez inne procesy.

- Klucze identyfikacyjne typu `key_t` (również `int`) stanowią wyższy poziom identyfikatorów pozwalających odwzorować dowolnie wybrany klucz liczbowy na rzeczywisty identyfikator. Wybór klucza ułatwia nieco zapewnienie poprawnego użycia urządzeń System V IPC.
- Uzyskiwanie dostępu do urządzeń:

```
int msgget(key_t key, int msgflg);
int semget(key_t key, int nsems, int semflg);
int shmget(key_t key, int size, int shmflg);
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

- Generacja kluczy funkcją `ftok` daje trzeci poziom identyfikatorów jeszcze bardziej ułatwiający wybór identyfikatora, choć nie rozwiązuje on problemów związanych z przypadkowym dostępem przez obcy proces.

```
key_t ftok(char* pathname, char proj);
```

- Konieczne jest jawne kasowanie urządzeń funkcjami kontrolnymi. Urządzenia komunikacyjne System V IPC istnieją trwale w pamięci systemu, ale nie są przechowywane na dysku. Oznacza to, że istnieją nadal po zakończeniu procesu, który je utworzył, a znikają bezpowrotnie dopiero po restarcie systemu. Przykład:

```
shmctl(shmid, IPC_RMID, /*ignorowane*/0);
```

Kolejki komunikatów System V IPC: klient

- właściwy/unikalny/prywatny identyfikator kolejki
- uzyskanie dostępu do kolejki, ew. jej utworzenie (flaga `IPC_CREAT`)
- określenie priorytetu i treści komunikatu (musi być strukturą `{long,char[]}`)
- priorytet pełni rolę typu komunikatu pozwalającego wybierać je selektywnie z kolejki
- wysłanie komunikatu z czekaniem lub bez (flaga `IPC_NOWAIT`)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

#define KEY ((key_t)987654L)
#define MSGLEN 80

int main() {
  int msqid, n;

  struct mbuf{
    long mtype;
    char mtext[MSGLEN];
  } msg = {115L, "Ala ma kota"};

  if ((msqid=msgget(KEY,0)) < 0 ) {
    printf("Brak dostepu do kolejki komunikatow!\n");
    exit(1);
  }

  if (msgsnd(msqid, (void *)&msg, MSGLEN, 0) < 0)
    printf("Blad wysylania komunikatu, errno=%d\n",
           errno);
}
```

Kolejki komunikatów System V IPC: serwer

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

#define KEY ((key_t)987654L)
#define MODES 0666
#define MSGLEN 80

void zakoncz(int syg) {
  if(msgctl(msqid, IPC_RMID,
            (struct msqid_ds *)0) < 0)
    printf("Nie moge usunac kolejki!\n");
  exit(2);
}

int main() {
  int msqid, n;
  struct mbuf {
    long mtype;
    char mtext[MSGLEN]; } msg;
  if ((msqid=msgget(KEY,MODES|IPC_CREAT))<0){
    printf("Nie moge utworzyc kolejki!\n");
    exit(1);
  }
  signal(SIGINT, zakoncz);
  while (1) {
    sleep(1);
    n = msgrcv(msqid, (void *)&msg,
               MSGLEN, 0, IPC_NOWAIT);
    if (n >= 0) printf("Komunikat: <%s>\n",
                       msg.mtext);
    else printf("Brak komunikatu: errno= %d\n",
                errno);
  }
}
```

Pamięć współdzielona System V IPC: serwer

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MEM_SIZ 4096

struct wspolna_struct {
    int klient_zapisal;
    char tekst[BUFSIZ];
};

int main() {
    int shmid;
    struct wspolna_struct *wspolna;

    shmid = shmget((key_t)1234, MEM_SIZ,
                   0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("porazka shmget");
        exit(errno);
    }

    wspolna = (struct wspolna_struct *)
              shmat(shmid, (void *)0, 0);
    if (wspolna == (struct wspolna_struct *)-1) {
        perror("porazka shmat");
        exit(errno);
    }

    wspolna->klient_zapisal = 0;
    srand((unsigned int)getpid());
    do {
        if (wspolna->klient_zapisal) {
            printf("Otrzymalem: %s", wspolna->tekst);
            sleep( rand() % 4 ); /* troche poczeka */
            wspolna->klient_zapisal = 0;
        }
        sleep(1);
    } while (strncmp(wspolna->tekst,"koniec",6) != 0);

    shmdt(wspolna);
    shmctl(shmid, IPC_RMID, 0);
    exit(0);

} /* main */
```

Pamięć współdzielona System V IPC: klient

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MEM_SIZ 4096

struct wspolna_struct {
    int klient_zapisal;
    char tekst[BUFSIZ];
};

int main() {
    struct wspolna_struct *wspolna;
    char bufor[BUFSIZ];
    int shmid;

    shmid = shmget((key_t)1234, MEM_SIZ,
                   0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("porazka shmget");
        exit(errno);
    }

    wspolna = (struct wspolna_struct *)
              shmat(shmid, (void *)0, 0);
    if (wspolna == (struct wspolna_struct *)-1) {
        perror("porazka shmat");
        exit(errno);
    }

    do {
        while(wspolna->klient_zapisal == 1) {
            sleep(1);
            printf("Czekam na odczytanie...\n");
        }
        printf("Podaj tekst do przeslania: ");
        fgets(bufor, BUFSIZ, stdin);

        strcpy(wspolna->tekst, bufor);
        wspolna->klient_zapisal = 1;
    } while (strncmp(bufor, "koniec", 6) != 0)

    shmdt((void *)wspolna);
    exit(0);

} /* main */
```

Semafory System V IPC

Semafory standardu System V IPC są kłopotliwe w użyciu i nie będą tu omawiane. Zwykle lepszym wyborem są semafory nowszego standardu POSIX.

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 muteksy, zmienne warunkowe, blokady zapisu i odczytu, itp.

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:

- dwukierunkowa komunikacja

  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

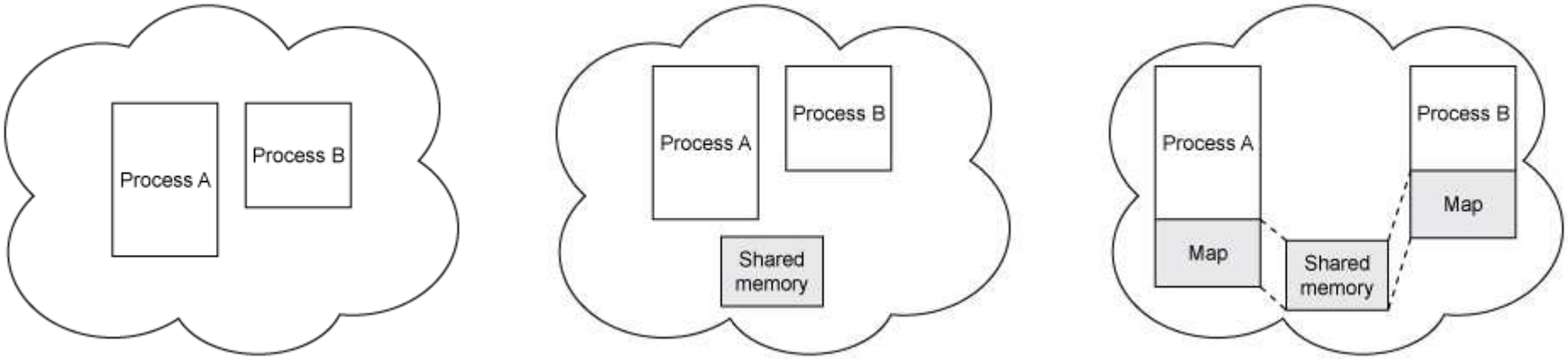
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <mqueue.h>

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

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

  printf("Probuje usunac istniejaca kolejke...\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 powyzej max: %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 POSIX

Komunikacja przez pamięć wspólną wymaga stworzenia obszaru pamięci wspólnej w systemie operacyjnym przez jeden z procesów, oraz **odwzorowania** tej pamięci do własnej przestrzeni adresowej wszystkich pragnących się komunikować procesów. Następnie komunikacja odbywa się przez zwykłe operacje na zmiennych, lub dowolną funkcją wykonującą odczyty/zapisy pamięci, np. `strcpy`, `memcpy`, itp.



Komunikacja przez pamięć wspólną jest najszybszym rodzajem komunikacji międzyprocesowej, ponieważ dane nie są nigdzie przesyłane. W momencie ich utworzenia w lokalizacji źródłowej są od razu również dostępne w lokalizacji docelowej. Jednak wymaga synchronizacji za pomocą oddzielnych mechanizmów, takich jak muteksy albo blokady zapisu i odczytu.

Operacje na obszarach pamięci wspólnej POSIX

Kroki niezbędne przy komunikacji przez pamięć współdzieloną:

1. Otwarcie/utworzenie pliku obszaru pamięci wspólnej `shm_open()`
2. Ustalenie rozmiaru obszaru pamięci wspólnej `ftruncate()`
3. Odwzorowanie obszaru pamięci wspólnej do obszaru w pamięci procesu `mmap()`
4. Praca: operacje wejścia/wyjścia `strcpy(), memcpy()`
5. Skasowanie odwzorowania `munmap()`
6. Skasowanie obszaru pamięci wspólnej `shm_unlink()`

Pamięć współdzielona POSIX: serwer

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

#define SHM_SEGMENT "/shm_segment"
#define MODES 0666

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

int main()
{
  int shmd, shared_size;
  shared_struct *segment;

  // na wszelki wypadek
  printf("Usuwam segment wspolny.\n");
  if(shm_unlink(SHM_SEGMENT) < 0)
    perror("nie moge usunac segmentu");
  else printf("Segment usuniety.\n");

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

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

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

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

Pamięć współdzielona POSIX: klient

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

#define SHM_SEGMENT "/shm_segment"
#define MODES 0666

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

int main()
{
  int shmd;
  shared_struct *segment;
  char buf[BUFSIZ];

  shmd = shm_open(SHM_SEGMENT, O_RDWR, MODES);
  if (shmd == -1) {
    perror("shm_open padlo");
    exit(errno);
  }

  segment =
    (shared_struct *)
    mmap(NULL, sizeof(shared_struct),
         PROT_READ|PROT_WRITE, MAP_SHARED, shmd, 0);
  do {
    while(segment->client_wrote == 1) {
      sleep(1);
      printf("Czekam na odczytanie...\n");
    }
    printf("Podaj tekst do przeslania: ");
    fgets(buf, BUFSIZ, stdin);
    strcpy(segment->text, buf);
    segment->client_wrote = 1;
  } while (strncmp(buf, "koniec", 6) != 0);

  munmap((char *)segment, sizeof(shared_struct));
  return 0;
}
```

Semafory: teoria

W teorii semafor jest nieujemną zmienną (unsigned int), domyślnie kontrolującą przydział pewnego zasobu. Wartość zmiennej 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 wartość dodatnią,

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

Istotna jest **niepodzielna** realizacja każdej z tych operacji, tzn. każda z operacji P, V musi zostać wykonana w całości, bez przerw na zawieszanie lub wywłaszczanie procesu. 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, zwany **muteksem** (*mutex*=*mut*ual *ex*clusion), który kontroluje dostęp do zasobu na zasadzie wyłączności. Wartość muteksu może wynosić 1 lub 0.

Semafory POSIX

Semafory występują w dwóch wariantach: i nazwane anonimowe. Semafor nazwany istnieje w systemie plików i wymaga otwarcia. Semafor anonimowy istnieje tylko jako zmienna w pamięci, i po utworzeniu przez dany proces, dostęp do niego mogą uzyskać tylko jego procesy potomne przez dziedziczenie.

Semafory nazwane:

```
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);
```

Semafory anonimowe:

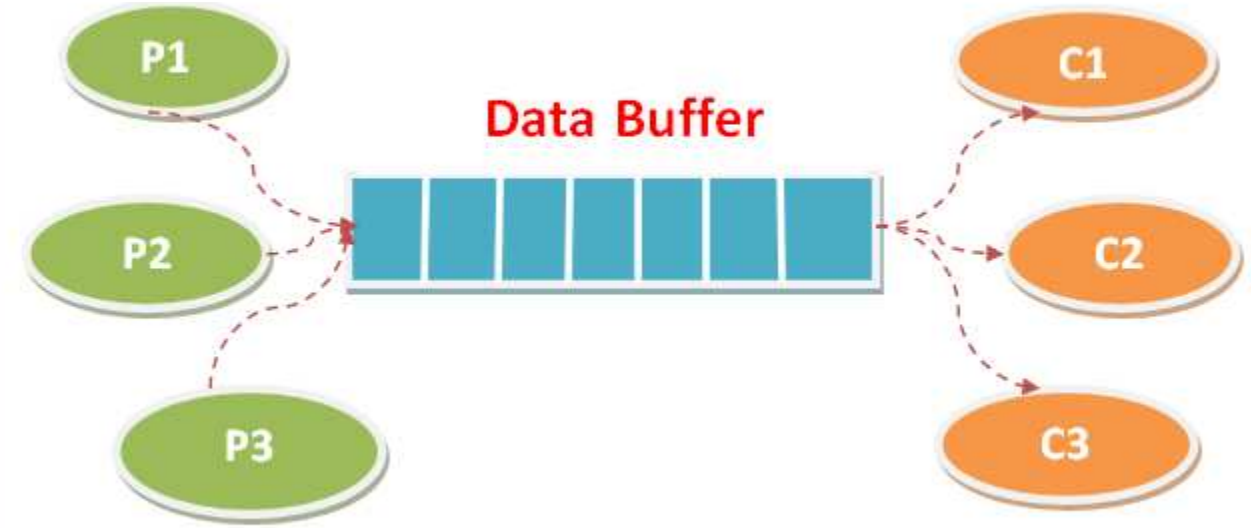
```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
```

Operacje na semaforach:

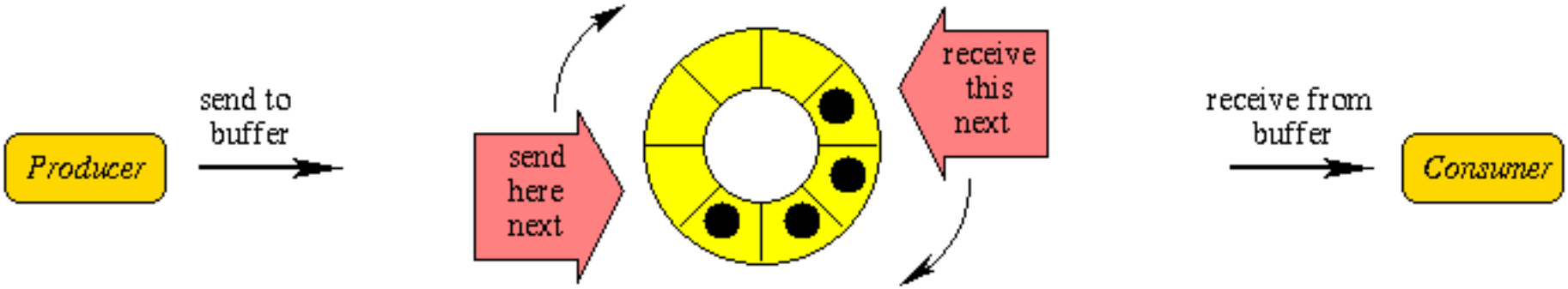
```
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
int sem_getvalue(sem_t *sem, int *sval);
```

Zagadnienie ograniczonego bufora

Modelowym przykładem zagadnienia, który można rozwiązać za pomocą komunikacji i synchronizacji, jest tzw. **problem ograniczonego bufora** (*bounded buffer problem*). Jest to wariant ogólnego zagadnienia producentów i konsumentów, w którym pewna grupa producentów produkuje jakieś dane, które muszą być następnie pobrane i dalej przetworzone („skonsumowane") przez grupę konsumentów.
W problemie ograniczonego bufora przekazywanie danych między producentami a konsumentami odbywa się za pośrednictwem bufora, którego pojemność jest dużo mniejsza niż liczba produkowanych i przekazywanych elementów. Zatem konieczna jest skuteczna koordynacja pracy wszystkich aktorów przy przekazywaniu wyprodukowanych elementów.



Zagadnienie ograniczonego bufora (cd.)




```
void producer()
{
  for(int i=0; i<REPEAT; i++) {
    //czekaj gdy brak pustych
    sem_wait(&(share->empty));
    sem_wait(&(share->mutex));
    printf("Prod: count=%d in=%d out=%d\n",
           share->count,share->in,share->out);
    sprintf(share->buf[share->in],
            "Komunikat %03d", i);
    share->count++;
    share->in = (share->in + 1) % B_SIZE;
    sem_post(&(share->mutex));
    //dodaj jeden zapelniony
    sem_post(&(share->full));
    sleep(1);
  }
  exit(0);
}
```

```
void consumer(void)
{
  for(int i=0; i<(PROD_NO*REPEAT); ++i) {
    //czekaj gdy brak gotowych
    sem_wait(&(share->full));
    sem_wait(&(share->mutex));
    printf("Cons: count=%d rcvd=%s\n",
           share->count,
           share->buf[share->out]);
    share->count--;
    share->out = (share->out + 1) % B_SIZE;
    sem_post(&(share->mutex));
    //dodaj jeden oprozniony
    sem_post(&(share->empty));
    sleep(1);
  }
  exit(0);
}
```

```
#define SHM_SEGMENT "/shm_segment"
#define MODES   0600  // prawa dostepu
#define B_SIZE  5     // rozmiar bufora
#define L_SIZE  80    // dlugosc wiersza
#define REPEAT  20    // liczba produktow
#ifndef PROD_NO
#define PROD_NO 2     // liczba producen.
#endif

typedef struct {
  char buf[B_SIZE][L_SIZE];
  int next_in;   // pierwsze wolne na prod
  int next_out;  // ost.zajete gdy count>0
  int count;     // liczba zajetych w buf.
  sem_t mutex;
  sem_t empty;
  sem_t full;
} shared_struct;

shared_struct *segment;

int main()
{
  int i, shmd;

  // utworz i zainicjalizuj segment
  shm_unlink(SHM_SEGMENT);
  shmd=shm_open(SHM_SEGMENT,
                O_RDWR|O_CREAT, MODES);
  ftruncate(shmd, B_SIZE);
  segment = (shared_struct *)
            mmap(0, B_SIZE, PROT_READ|
                 PROT_WRITE, MAP_SHARED,shmd,0);

  // inicjalizacja wartosci semaforow
  segment->count = 0;
  segment->in = 0;
  segment->out = 0;
  sem_init(&(segment->mutex), 1, 1));
  sem_init(&(segment->empty), 1, B_SIZE));
  sem_init(&(segment->full), 1, 0));

  // uruchom produkcje/konsumpcje
  for(i=1; i<=PROD_NO; ++i)
    if (fork() == 0) producer(i);
  if (fork() == 0) consumer();

  // odlacz semafory proc.glownemu
  sem_close(&(segment->mutex));
  sem_close(&(segment->empty));
  sem_close(&(segment->full));


  // czekaj na potomkow i zakoncz
  for(i=1; i<=PROD_NO; ++i) wait(NULL);
  wait(NULL);
  return 0;
}
```