# Unix: komunikacja międzyprocesowa

Witold Paluszyński
witold.paluszynski@pwr.wroc.pl
http://sequoia.ict.pwr.wroc.pl/~witold/

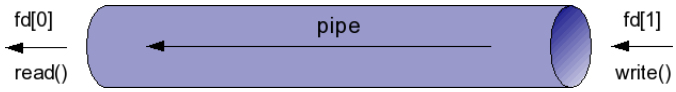


Niniejszy dokument zawiera materiały do wykładu na temat programowania komunikacji międzyprocesowej w systemie Unix. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

## Komunikacja międzyprocesowa — potoki

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

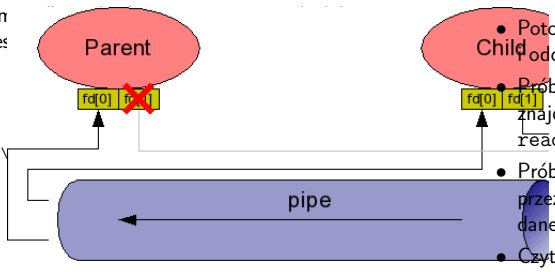
- na potoku można wykonywać tylko operacje odczytu i zapisu, funkcjami `read` i `write`, jak dla zwykłych plików,
- potoki są dostępne i widoczne w postaci jednego lub dwóch deskryptorów plików, oddzielnie dla końca zapisu i odczytu,
- potok ma określoną pojemność, i w granicach tej pojemności można 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. „anonim
otwartych i gotowych do pracy des



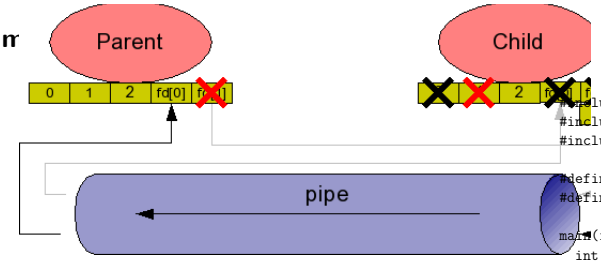

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define KOM "Komunikat dla rodzica.\
int main() {
  int potok_fd[2], licz, status;
  char bufor[BUFSIZ];

  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: m




```
#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: komunikacja dwukierunkowa — rodzic

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

#define OK 0
#define BLAD -1

main(int argc, char *argv[]) {
  int pipein[2], pipeout[2];
  int liczba, wynik, ntest, i;
  char znak, line[40];

  printf("Uwaga, rodzic startuje...\n");
  if (pipe(pipein) == BLAD || pipe(pipeout) == BLAD) {
    fprintf(stderr, "*** Blad pipe\n");
    exit(1);
  }

  switch (fork()) {
  case BLAD:
    fprintf(stderr, "*** Blad fork\n");
    exit(1);

  case  OK:
    if (close(0) == BLAD) {
      fprintf(stderr, "*** Blad close(0)\n");
      exit(1);
    }
    if (dup(pipeout[0]) == BLAD) {
      fprintf(stderr, "*** Blad dup(pipeout[0]\n");
      exit(1);
    }
    if (close(1) == BLAD) {
      fprintf(stderr, "*** Blad close(1)\n");
      exit(1);
    }
    if (dup(pipein[1]) == BLAD) {
      fprintf(stderr, "*** Blad dup(pipein[1]\n");
      exit(1);
    }
    if ((close(pipeout[0]) == BLAD) ||
        (close(pipeout[1]) == BLAD) ||
        (close(pipein[0]) == BLAD) ||
        (close(pipein[1]) == BLAD)) {
      fprintf(stderr, "*** Blad close(pipein/out[0/1])\n");
      exit(1);
    }

    execlp("slave", "slave", NULL);
    fprintf(stderr, "*** Blad execlp(slave)\n");
    exit(1);
  }

  printf("No, potomek splodzony, rodzic kontynuuje...\n");
  if ((close(pipeout[0]) == BLAD) ||
      (close(pipein[1]) == BLAD)) {
    fprintf(stderr, "*** Blad close(pipein[0/1])\n");
    exit(1);
  }
  srand(1994);
  for (ntest=0; ntest<100; ntest++) {
    liczba = rand();
    sprintf(line, "%38d", liczba);
    line[38] = '\n';
    line[39] = '\0';
    printf("Wysylamy komunikat: %s\n", line);
    if (write(pipeout[1], line, 39) == BLAD) {
      fprintf(stderr, "*** Blad write(pipeout[1]): %s\n", line);
      exit(1);
    }
    printf("Dane wyslane, teraz czekamy na wyniki\n");
    for (i=0;i<40;i++)
      line[i]=' ';
    if (read(pipein[0],line,40) == BLAD) {
      fprintf(stderr, "*** Blad read(pipein[0]): %s\n", line);
      exit(1);
    }
    sscanf(line, "%d", &wynik);
    printf("Wartosc otrzymana z podprocesu: %d\n", wynik);
  }

  if (close(pipeout[1]) == BLAD || close(pipein[0]) == BLAD) {
    fprintf(stderr, "*** Blad close(pipeout[1]/pipein[0])\n");
    exit(1);
  }
  return(1);
}
```

## Potoki: komunikacja dwukierunkowa — potomek

```
#include <stdio.h>

main()
{
  char line[81];
  int  xliczba;

  while (gets(line) != NULL) {
    sscanf(line,"%d\n",&xliczba);
    printf("%d\n",xliczba*xliczba);
  }
}
```

## 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.
- 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 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.

## System V IPC

- Urządzenia komunikacyjne:
  - kolejki komunikatów: koniec–koniec
  - pamięć (współ)dzielona: wielu–wielu
  - semafory: liczby całkowite
- Istnieją globalnie w systemie: polecenia `ipcs, ipcrm`.
- Nie są deskryptorami plików i nie wykonuje się na nich operacji I/O standardowymi funkcjami `read/write`, lecz 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.)
- Identyfikatory urządzeń System V IPC (typu `int`) są globalne w systemie, odmiennie niż deskryptory plików (także nie są kolejno generowanymi małymi liczbami). Oznacza to, że jeden proces może użyć identyfikatora utworzonego przez inny proces.

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

- Klucze identyfikacyjne typu `key_t` (również `int`) stanowią drugi 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 shmget(key_t key, int size, int shmflg);
int semget(key_t key, int nsems, int semflg);
```

- 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);
```

- Prawa dostępu do urządzeń: `RWRWRW`
- 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, ale znikają bezpowrotnie przy restarcie systemu.
- Mechanizmy komunikacji System V IPC są zawarte w rozszerzeniu XSI standardu POSIX.

## System V IPC — kolejki komunikatów: serwer

- właściwy/unikalny/prywatny identyfikator kolejki
- uzyskanie dostępu do kolejki, ew. jej utworzenie
- określenie priorytetu i treści komunikatu
- wysłanie komunikatu z czekaniem lub bez

```
#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 komunikatow!\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 komunikatow!\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);
  }
}
```

## System V IPC — kolejki komunikatów: klient

```
#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);
}
```

## System V IPC — pamięć współdzielona: 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, pracuj = 1;
    void *pamiec_wspolna = (void *)0;
    struct wspolna_struct *wspolna;

    srand((unsigned int)getpid());

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

```
    pamiec_wspolna = shmat(shmid, (void *)0, 0);
    if (pamiec_wspolna == (void *)-1) {
        perror("shmat padlo");
        exit(errno);
    }

    wspolna = (struct wspolna_struct *)pamiec_wspolna;
    wspolna->klient_zapisal = 0;
    while(pracuj) {
        if (wspolna->klient_zapisal) {
            printf("Otrzymalem: %s", wspolna->tekst);
            sleep( rand() % 4 ); /* niech troche poczeka */
            wspolna->klient_zapisal = 0;
            if (strncmp(wspolna->tekst, "koniec", 6) == 0)
                pracuj = 0;
        }
        sleep(1);
    }

    if (shmdt(pamiec_wspolna) == -1) {
        perror("shmdt padlo");
        exit(errno);
    }
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("shmctl(IPC_RMID) padlo");
        exit(errno);
    }
    exit(0);
}
```

## System V IPC — pamięć współdzielona: 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() {
    int pracuj = 1;
    void *pamiec_wspolna = (void *)0;
    struct wspolna_struct *wspolna;
    char bufor[BUFSIZ];
    int shmid;

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

```
    pamiec_wspolna = shmat(shmid, (void *)0, 0);
    if (pamiec_wspolna == (void *)-1) {
        perror("shmat padlo");
        exit(errno);
    }

    wspolna = (struct wspolna_struct *)pamiec_wspolna;
    while(pracuj) {
        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;

        if (strncmp(bufor, "koniec", 6) == 0) {
              pracuj = 0;
        }
    }

    if (shmdt(pamiec_wspolna) == -1) {
        perror("shmdt padlo");
        exit(errno);
    }
    exit(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 System V IPC

Implementacja semaforów System V IPC opiera się na następujących zasadach:

- Semafory tworzone są jako zbiory semaforów; w typowych przypadkach użycia wykorzystuje się zbiory składające się z pojedynczego semafora:

  ```
  semget(key, nsems, semflg);
  ```

- Zamiast funkcji implementujących podstawowe operacje semaforowe, wykorzystuje się sekwencje operacji elementarnych na poszczególnych semaforach ze zbioru. Te sekwencje system realizuje w sposób atomowy, tzn. cała sekwencja jest zawsze wykonana niepodzielnie. Sekwencje operacji tworzone są przez program jako tablice struktur:

  ```
  struct sembuf { short sem_num;
                  short sem_op;
                  short sem_flg; }
  ```

- Istnieje dodatkowy mechanizm UNDO — ustawienie flagi `SEM_UNDO` przez proces w operacjach semaforowych spowoduje „odkręcenie" tych operacji w przypadku śmierci procesu. Zapewnia to zdolność kontynuacji pracy z semaforem systemu, w którym jakiś proces zginął, lub zwyczajnie zakończył pracę, ale nie „zbilansował" wcześniej do zera swoich operacji na semaforze.

## Semafory System V IPC — operacje

Operacje na semaforach realizuje funkcja `semop(semid,sembufops,nops)`, gdzie `sembufops` jest wskaźnikiem do tablicy `nops` struktur `sembuf` opisujących sekwencję operacji semaforowych z kodami:

- `sem_op > 0`
  wartość `sem_op` dodawana jest do wartości semafora, co odpowiada zwolnieniu pewnej ilości zasobu
  (przy $+1$ otrzymujemy operację V)
- `sem_op < 0`
  wartość `sem_op` odejmowana jest od wartości semafora o ile $|$`sem_op`$|$ jest mniejsza od wartości semafora; w przeciwnym wypadku operacja czeka na zwiększenie wartości semafora, co odpowiada próbie zajęcia zasobu
  (przy $-1$ otrzymujemy operację P)
- `sem_op == 0`
  operacja czeka na wyzerowanie się semafora bez zmiany jego wartości, co odpowiada wykrywaniu wyczerpania zasobu („zagłodzenia" się systemu), w celu podjęcia jakiejś akcji, na przykład przydzielenia większej ilości zasobu
  (ta operacja nie ma odpowiednika dla semaforów teoretycznych)

## Semafory System V IPC: przykład użycia

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 123456L/*klucz semafora do blokad*/
#define PERMS  0666

static struct sembuf op_lock[2] = {
    0, 0, 0,       /*czekaj az sem nr 0 bedzie 0*/
    0, 1, SEM_UNDO /*wtedy zwieksz sem nr 0 o 1 */
};

static struct sembuf op_unlock[1] = {
    0, -1, (IPC_NOWAIT | SEM_UNDO)
                   /*zmn.sem0 o 1,tzn.ustaw na 0*/
};

int semid = -1;    /*bedzie identyfikatorem sem.*/

my_lock(int fd) {
  if (semid < 0) {
    if((semid=semget(SEMKEY,1,IPC_CREAT|PERMS))<0)
      perror("semget error");
    }
    if (semop(semid, op_lock, 2) < 0)
      perror("semop lock error");
}
```

```
my_unlock(int fd) {
  if(semop(semid,op_unlock,1)<0)
    perror("semop unlock error");
}
```

## Semafory System V IPC — operacje kontrolne

Poza „zwykłymi" operacjami semaforowymi wykonywanymi funkcją `semop` System V IPC udostępnia dodatkowe operacje, zwane „kontrolnymi", wykonywanymi funkcją `semctl`.

Do operacji wykonywanych tą funkcją należą m.in.:

- usunnięcie całego zbioru semaforów z systemu (`cmnd=IPC_RMID`),
- ustawienie określonej (np. początkowej) wartości semafora o danym numerze w zbiorze (`cmnd=SETVAL/SETALL`),
- pobieranie aktualnych wartości semaforów ze zbioru (`cmnd=GETVAL/GETALL`),
- pobieranie informacji o procesach oczekujących na danym semaforze (`cmnd=GETNCNT/GETZCNT`),
- jeszcze inne.

## Semafory System V IPC — uwagi

Semafory standardu System V IPC są kłopotliwe w użyciu. Zwykle lepszym wyborem są semafory nowszego standardu Posix.

## Synchronizacja dostępu do pamięci semaforami: 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>
#include <sys/sem.h>

#define SHM_KEY ( (key_t) 987654L )
#define R_SEM_KEY ( (key_t) 987654L )
#define W_SEM_KEY ( (key_t) 987653L )
#define MODES 0666

static struct sembuf sem_wait = {0, -1, 0},
                     sem_post = {0, 1, 0};

static union semun {
    int             val;
    struct semid_ds *buf;
    unsigned short  *array;
} sem_arg ;

int main()
{
  char *shared_mem = (char *)0;
  char buf[BUFSIZ], *charptr;
```

```
  int shmid, r_semid, w_semid;

  r_semid = semget(R_SEM_KEY, 1, IPC_CREAT|MODES);
  if (r_semid == -1) {
    perror("semget(\"R\") padlo");
    exit(errno);
  }

  w_semid = semget(W_SEM_KEY, 1, IPC_CREAT|MODES);
  if (w_semid == -1) {
    perror("semget(\"W\") padlo");
    exit(errno);
  }

  shmid = shmget(SHM_KEY, BUFSIZ, IPC_CREAT|MODES);
  if (shmid == -1) {
    perror("shmget padlo");
    exit(errno);
  }

  shared_mem = (char *)shmat(shmid, (void *)0, 0);
  if (shared_mem == (char *)-1) {
    perror("shmat padlo");
    exit(errno);
  }

  srand((unsigned int)getpid());
  sem_arg.val = 0; /* inicjalizacja semafora */
  semctl(r_semid, 0, SETVAL, sem_arg);
  do {
```

```
    printf("Czekam na dane ...\n");
    if (0!=semop(r_semid, &sem_post, 1)) { /* otworz semafor gotowosci */
      perror("semop(\"R\",\"post\")");
    }
    sleep( rand() % 6 ); /* troche czekamy */
    if (0!=semop(w_semid, &sem_wait, 1)) { /* czekaj na zapis, potwierdz */
      perror("semop(\"W\",\"wait\")");
    } else {
      printf("Otrzymalem: \"%s\"\n", shared_mem);
      sleep( rand() % 6 ); /* znow troche poczekajmy */
    }
  } while (strncmp(shared_mem, "koniec", 6) != 0);

  if (shmdt(shared_mem) == -1) {
    perror("shmdt padlo");
    exit(errno);
  }

  if (shmctl(shmid, IPC_RMID, 0) == -1) {
    perror("shmctl(IPC_RMID) padlo");
    exit(errno);
  }

  if (semctl(r_semid, 0, IPC_RMID) == -1) {
    perror("semctl(\"R\",IPC_RMID) padlo");
    exit(errno);
  }

  if (semctl(w_semid, 0, IPC_RMID) == -1) {
    perror("semctl(\"W\",IPC_RMID) padlo");
```

```
    exit(errno);
  }

  return 0;
}
```

## Synchronizacja dostępu do pamięci semaforami: 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>
#include <sys/sem.h>

#define SHM_KEY ( (key_t) 987654L )
#define R_SEM_KEY ( (key_t) 987654L )
#define W_SEM_KEY ( (key_t) 987653L )
#define MODES 0666

static struct sembuf sem_nowait = {0, -1, IPC_NOWAIT},
                     sem_post = {0, 1, 0};

const static struct timespec timeout = { 1, 0 }; /* 1sec */

int main()
{
  char *shared_mem = (char *)0;
  char buf[BUFSIZ], *charptr;
  int shmid, r_semid, w_semid;

  r_semid = semget(R_SEM_KEY, 1, IPC_CREAT|MODES);
  if (r_semid == -1) {
```

```
  perror("semget(\"R\") padlo");
  exit(errno);
}

w_semid = semget(W_SEM_KEY, 1, IPC_CREAT|MODES);
if (w_semid == -1) {
  perror("semget(\"W\") padlo");
  exit(errno);
}

shmid = shmget(SHM_KEY, BUFSIZ, IPC_CREAT|MODES);
if (shmid == -1) {
  perror("shmget padlo");
  exit(errno);
}

shared_mem = (char *)shmat(shmid, (void *)0, 0);
if (shared_mem == (char *)-1) {
  perror("shmat padlo");
  exit(errno);
}

do {
  while(0!=semop(r_semid, &sem_nowait, 1)) {
    printf("Czekam na odczyt/gotowosc serwera.\n");
    sleep(1);
  }
  printf("Podaj text do przeslania: ");
  fgets(buf, BUFSIZ, stdin);
  charptr = strchr(buf, '\n');
  if (NULL!=charptr)
    *charptr = 0;
  strcpy(shared_mem, buf);
  if (0!=semop(w_semid, &sem_post, 1)) { /* sygnalizuj zapis gotowy */
    perror("semop(\"w\",\"post\")");
  }
} while (strncmp(buf, "koniec", 6) != 0);

if (shmdt(shared_mem) == -1) {
  perror("shmdt padlo");
  exit(errno);
}

return 0;
}
```

## 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: 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;
}
```

## Synchronizacja dostępu do pamięci semaforami: serwer

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

#define SEM_READSEM "/sem_readsem"
#define SEM_WRITESEM "/sem_writesem"
#define SHM_TESTMEM "/shm_testmem"
#define SHM_SIZE BUFSIZ
#define MODES 0666

int main()
{
  char *shared_mem;
  int shmd;
  sem_t *semr, *semw;

  // na wszelki wypadek
  printf("Probuje usunac istniejacy semafor R ...\n");
  if(sem_unlink(SEM_READSEM) < 0)
    perror("nie moge usunac semafora R");
  else printf("Semafor R usuniety.\n");
  printf("Probuje usunac istniejacy semafor W ...\n");
```

```
  if(sem_unlink(SEM_WRITESEM) < 0)
    perror("nie moge usunac semafora W");
  else printf("Semafor W usuniety.\n");
  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");

  semr = sem_open(SEM_READSEM, O_CREAT, MODES, 0); /* wstepnie zamkniety */
  if (semr == SEM_FAILED) {
    perror("sem_open R padlo");
    exit(errno);
  }

  semw = sem_open(SEM_WRITESEM, O_CREAT, MODES, 1); /* wstepnie otwarty */
  if (semw == SEM_FAILED) {
    perror("sem_open W padlo");
    exit(errno);
  }

  shmd = shm_open(SHM_TESTMEM, O_RDWR|O_CREAT, MODES);
  if (shmd == -1) {
    perror("shm_open padlo");
    exit(errno);
  }
  ftruncate(shmd, SHM_SIZE);
  shared_mem = (char*) mmap(NULL, SHM_SIZE, PROT_READ, MAP_SHARED, shmd, 0);

  srand((unsigned int)getpid());
  do {
```

```
    printf("Czekam na dane ...\n");
    sleep( rand() % 4 );        /* troche czekamy */
    if (0!=sem_trywait(semr)) { /* proba odczytu */
      perror("sem_wait R");
    } else {
      printf("Otrzymalem: \"%s\"\n", shared_mem);
      sleep( rand() % 4 );      /* znow troche poczekajmy */
      if (0!=sem_post(semw)) {  /* zezwala na ponowny zapis */
        perror("sem_post W");
      }
    }
  } while (strncmp(shared_mem, "koniec", 6) != 0);

  munmap((char *)shared_mem, SHM_SIZE);
  sem_close(semr);
  sem_close(semw);
  return 0;
}
```

## Synchronizacja dostępu do pamięci semaforami: klient

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

#define SEM_READSEM "/sem_readsem"
#define SEM_WRITESEM "/sem_writesem"
#define SHM_TESTMEM "/shm_testmem"
#define SHM_SIZE BUFSIZ
#define MODES 0666

int main()
{
  char *shared_mem, buf[SHM_SIZE], *charptr;
  int shmd;
  sem_t *semr, *semw;

  semr = sem_open(SEM_READSEM, O_CREAT, MODES, 0); /* wstepnie zamkniety */
  if (semr == SEM_FAILED) {
    perror("sem_open R padlo");
    exit(errno);
  }
```

```
  munmap((char *)shared_mem, SHM_SIZE);
  sem_close(semr);
  sem_close(semw);
  return 0;
}
```

```
semw = sem_open(SEM_WRITESEM, O_CREAT, MODES, 1); /* wstepnie otwarty */
if (semw == SEM_FAILED) {
  perror("sem_open W padlo");
  exit(errno);
}

shmd = shm_open(SHM_TESTMEM, O_RDWR|O_CREAT, MODES);
if (shmd == -1) {
  perror("shm_open padlo");
  exit(errno);
}
ftruncate(shmd, SHM_SIZE);
shared_mem = (char*) mmap(NULL, SHM_SIZE, PROT_WRITE, MAP_SHARED, shmd, 0);

do {
  while(0!=sem_trywait(semw)) { /* proba wejscia w zapis */
    printf("Czekam na odczytanie...\n");
    sleep(1);
  }
  printf("Podaj text do przeslania: ");
  fgets(buf, SHM_SIZE, stdin);
  charptr = strchr(buf, '\n');
  if (NULL!=charptr)
    *charptr = 0;
  strcpy(shared_mem, buf);
  if (0!=sem_post(semr)) {    /* zezwala na odczyt */
    perror("sem_post R");
  }
} while (strncmp(buf, "koniec", 6) != 0);
```