

Unix: programowanie procesów

Witold Paluszyński

witoldp@pwr.wroc.pl

<http://sequoia.ict.pwr.wroc.pl/~witold/>

Copyright © 1999–2006 Witold Paluszyński

All rights reserved.

Niniejszy dokument zawiera materiały do wykładu na temat programowania procesów i 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.

Środowisko procesu unixowego

- Proces unixowy:
 - kod programu w pamięci (obszar programu i danych)
 - **środowisko** (zestaw zmiennych i przypisanych im wartości)
 - zestaw dalszych informacji utrzymywanych przez jądro Unixa o wszystkich istniejących procesach, m.in. tablicę otwartych plików, maskę sygnałów, priorytet, i in.
- Funkcja `main()` wywoływana przez procedurę startową z następującymi argumentami:
 - liczbą argumentów wywołania: `int argc`
 - wektorem argumentów wywołania: `char *argv[]`
 - środowiskiem: `char **environ`
rzadko używanym ponieważ dostęp do środowiska możliwy jest również poprzez funkcje: `getenv()` / `putenv()`
- Proces charakteryzują: `pid`, `ppid`, `pgrp`, `uid`, `euid`, `gid`, `egid`.
Wartości te można uzyskać funkcjami `get...()`

Tworzenie procesów

- Procesy tworzone są przez klonowanie istniejącego procesu funkcją `fork()`, zatem każdy proces jest podprocesem (*child process*) jakiegoś procesu nadrzędnego, albo inaczej rodzicielskiego (*parent process*).
- Jedynym wyjątkiem jest proces numer 1 — **init** — tworzony przez jądro Unixa w chwili startu systemu. Wszystkie inne procesy w systemie są bliższymi lub dalszymi potomkami inita.
- Podproces dziedziczy i/lub współdzieli pewne atrybuty i zasoby procesu nadrzędnego, a gdy kończy pracę, Unix zatrzymuje go do czasu odebrania przez proces nadrzędny statusu podprocesu (wartości zakończenia).

Kończenie pracy procesów

- Zakończenie procesu unixowego może być **normalne**, wywołane przez zakończenie funkcji `main()`, lub funkcję `exit()`. Wtedy:
 - następuje wywołanie wszystkich handlerów zarejestrowanych przez funkcję `atexit()`,
 - następuje zakończenie wszystkich operacji wejścia/wyjścia procesu i zamknięcie otwartych plików,
 - można spowodować normalne zakończenie procesu bez kończenia operacji wejścia/wyjścia ani wywoływania handlerów `atexit`, przez wywołanie funkcji `_exit()`.
- Zakończenie procesu może być **anormalne** (ang. *abnormal*), przez wywołanie funkcji `abort`, lub otrzymanie sygnału (funkcje `kill()`, `raise()`).

Status procesu

- W chwili zakończenia pracy proces generuje kod zakończenia, tzw. **status**, który jest wartością zwróconą z funkcji `main()` albo `exit()`. W przypadku śmierci z powodu otrzymania sygnału kodem zakończenia jest wartość $128 + \text{nr_sygnału}$.
- Rodzic normalnie powinien po śmierci potomka odczytać jego kod zakończenia wywołując funkcję systemową `wait()` (lub `waitpid()`). Aby to ułatwić (w nowszych systemach) w chwili śmierci potomka jego rodzic otrzymuje sygnał `SIGCLD` (domyślnie ignorowany).
- Gdy proces nadrzędny żyje w chwili zakończenia pracy potomka, i nie wywołuje funkcji `wait()`, to potomek pozostaje (na czas nieograniczony) w stanie zwanym **zombie**. Może to być przyczyną wyczerpania jakichś zasobów systemu, np. zapęłnienia tablicy procesów, otwartych plików, itp.
- Istnieje mechanizm adopcji polegający na tym, że procesy, których rodzic zginął przed nimi (sieroty), zostają adoptowane przez proces numer 1, `init`. `Init` jest dobrym rodzicem (choć przybranym) i wywołuje okresowo funkcję `wait()` aby umożliwić poprawne zakończenie swoich potomków.

Grupy procesów

- **Grupa procesów:** wszystkie podprocesy uruchomione przez jeden proces nadrzędny. Każdy proces w chwili utworzenia automatycznie należy do grupy procesów swojego rodzica.
- Pod pewnymi względami przynależność do grupy procesów jest podobna do przynależności do partii politycznych. Każdy proces może:
 - założyć nową grupę procesów (o numerze równym swojemu PID),
 - wstąpić do dowolnej innej grupy procesów,
 - włączyć dowolny ze swoich podprocesów do dowolnej grupy procesów (ale tylko dopóki podproces wykonuje kod rodzica).
- Grupy procesów mają głównie znaczenie przy wysyłaniu sygnałów.

Zasoby procesu

- Ograniczenia zasobów procesu: `getrlimit/setrlimit`: `RLIMIT_CORE`, `RLIMIT_CPU`, `RLIMIT_DATA`, `RLIMIT_FSIZE`, `RLIMIT_NOFILE`, i inne, zależnie od systemu

Sesja i terminal

Sesję stanowi zbiór grup procesów o wspólnym numerze sesji. Zwykle potoki poleceń uruchamiane w interpreterze poleceń stanowią oddzielne grupy procesów, lecz wszystkie należą do jednej sesji (interpretera poleceń).

Sesja może posiadać tzw. terminal sterujący. Wtedy w sesji istnieje jedna grupa procesów pierwszoplanowych (*foreground*) i dowolna liczba grup procesów drugoplanowych, czyli pracujących w tle (*background*). Procesy z grupy pierwszoplanowej otrzymują sygnały i dane z terminala.

Pojęcie terminala pochodzi od ekranowych terminali alfanumerycznych służących użytkownikom do łączenia się z komputerem. Przy połączeniach przez sieć, lub z systemu okienkowego Unix stosuje się pojęcie *pseudo-terminala* stanowiącego urządzenie komunikacyjne składające się z dwóch części: części użytkownika do której połączenie sieciowe wysyła dane użytkownika, i części programowej, którą widzi program na zdalnym komputerze, np. interpreter poleceń.

Mogą istnieć procesy nie posiadające terminala sterującego. Często przydatne jest by procesy pracujące ciągle w tle nie miały terminala sterującego. Takie procesy nazywamy **demonami** (ang. *daemon*).

Stany procesów

wykonywalny (runnable) — proces w kolejce procesów gotowych do wykonania

uśpiony (sleeping) — proces czeka na coś (np. na dane do przeczytania, na sygnał, na dostęp do zasobu, lub dobrowolnie śpi na określony okres)

zatrzymany (stopped) — proces gotowy do wykonywania lecz zatrzymany na skutek otrzymania sygnału SIGSTOP lub SIGTSTP, może to być skutkiem dobrowolnego żądania procesu, celowego wysłania sygnału (np. przez użytkownika), lub odwołania się procesu pracującego w tle do terminala sterującego; wznowienie pracy procesu następuje po otrzymaniu sygnału SIGCONT

wymieciony (swapped out) — proces usunięty okresowo z kolejki procesów gotowych do wykonywania wskutek działania algorytmu obsługi pamięci wirtualnej; stan wymiecenia nie jest właściwym stanem procesu — może on być w jednym z trzech poprzednich stanów; wymiecenie procesu wykonywalnego oznacza brak dostatecznej ilości pamięci fizycznej na jednoczesne skuteczne wykonywanie wszystkich procesów wykonywalnych

zombie — proces czeka na zakończenie

Tworzenie procesów

- `system()` – tworzy podprocesy interpretera komend i programu
- `popen()` – również tworzy dwa podprocesy
- `fork/exec` – rozdzielone tworzenie podprocesu i wykonanie programu, bez dodatkowego interpretera komend

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

int main() {
    char komenda[BUFSIZ];

    sprintf(komenda, "ps -fu %s", getenv("LOGNAME"));
    system(komenda);
    return(0);
}
```

- funkcja `system` zwraca wartość statusu zwróconą przez interpreter poleceń, która zwykle jest statusem wykonanego polecenia

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

    case 0:
        /* jestem potomkiem, no to znikam... */
        execlp("program", "program", NULL);
        fprintf(stderr, "Blad, nieudane execlp, errno=%d\n", errno);
        _exit(0);

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

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 handlers
- 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 — mechanizmy generowania

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

Sygnały — funkcje obsługi (tradycyjne)

- `kill` – wysyła sygnał do innego procesu
- `raise` – wysyła sygnał do własnego procesu
- `pause` – czeka na sygnał
- `signal` – deklaruje jednorazową reakcję na sygnał: ignorowanie, blokowanie, obsługa, oraz reakcja domyślna (zakończenie)
- `sigset/sighold/sigrelse/sigignore/sigpause` — inne funkcje tradycyjnego modelu obsługi sygnałów
- `alarm` – uruchamia „budzik”, który przyśle sygnał `SIGALRM`

Handlery sygnałów

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

Zwróćmy uwagę:

- deklaracja handlera: funkcja typu void z pojedynczym argumentem int
- powrót z handlera — wznowienie pracy programu

Handlery sygnałów (cd.)

```
void handler(int signal) {
    signal(signal, SIG_IGN);
    /* wlasciwe akcje handlera, np: */
    unlink(tmp_file);
    exit(1);
}

int main() {
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, handler);
    ...
}
```

- sprawdzenie w programie, czy dyspozycją sygnału nie jest ignorowanie — obsługa takich sygnałów nie powinna być zmieniana
- ignorowanie sygnału na wejściu do handlera
- zakończenie pracy handlera:
 - koniec procesu
 - kontynuacja
 - kontynuacja z przekazaniem informacji
 - kontynuacja od określonego punktu
- ponowne uzbrojenie handlera w przypadku kontynuacji

Handlery sygnałów — alarm

```
#include <signal.h>
#include <setjmp.h>

jmp_buf stan_pocz;
int obudzony = 0;

void obudz_sie(int syg) {
    signal(syg, SIG_HOLD);
    fprintf(stderr, "Przerwane sygnałem %d\n", syg);
    obudzony = 1;
    longjmp(stan_pocz, syg);
}

int main() {
    int syg;

    ...
    syg = setjmp(stan_pocz); /* np.poczatek petli */
    signal(SIGTERM, obudz_sie);
    signal(SIGINT, obudz_sie);
    signal(SIGALRM, obudz_sie);
    if (obudzony == 0) {
        alarm(30);           /* limit 30 sekund */
    }
}
```

```
DlugieObliczenia();  
alarm(0);           /* zdazyliśmy */  
}  
else  
    printf("Program wznowiony po przerwaniu sygnałem %d\n", syg);
```

Sygnały — podejście ulepszone

Nowy zestaw funkcji oferujący możliwość kompleksowego i trwałego deklarowania obsługi sygnałów:

- `sigaction()` – trwale deklaruje reakcję na sygnał
 - automatyczne blokowanie na czas obsługi obsługiwanego sygnału i dowolnego zbioru innych sygnałów
 - dodatkowe opcje obsługi sygnałów, np. handler sygnału może otrzymać dodatkowe argumenty: strukturę informującą o okolicznościach wysłania sygnału i drugą strukturę informującą o kontekście przez odebraniem sygnału
- rozszerzony prototyp handlera:

```
void handler(int sig, siginfo_t *sip, ucontext_t *uap);
```

- `sigprocmask()/sigfillset()/sigaddset()` – operacje na zbiorach sygnałów

```

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void joj(int sig, siginfo_t *sip, ucontext_t *uap) {
    printf("Dostalem signal numer %d\n", sig);
    if (sip->si_code <= 0)
        printf("Od procesu %d\n", sip->si_pid);
    printf("Kod sygnalu %d\n", sip->si_code);
}

void main() {
    struct sigaction act;

    act.sa_handler = joj;
    sigemptyset(&act.sa_mask);
    act.sa_flags = SA_SIGINFO;

    sigaction(SIGINT, &act, 0);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}

```

Potoki: funkcja popen

utworzenie podprocesu i komunikacja przez potok

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

int main() {
    FILE *proc_fp;
    char komenda[BUFSIZ], bufor[BUFSIZ];

    sprintf(komenda, "ps -fu %s", getenv("LOGNAME"));
    proc_fp = popen(komenda, "r");
    while (fgets(bufor, BUFSIZ, proc_fp) != NULL)
        fputs(bufor, stdout);
    return(pclose(proc_fp));
}
```

Potoki: funkcja pipe

```
#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 zamkniętego deskryptora, lecz jeśli jakiś proces ma jeszcze ten deskryptor otwarty to funkcja read „zawisa” na próbie odczytu

Potoki: zasady użycia

- potok (nienazwany) 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 pisać jest nadal otwarty przez jakiś proces, powoduje „zawiśnięcie” funkcji `read()`, i powrót gdy jakieś dane pojawią się w potoku
- czytanie z potoku, którego koniec pisać 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 pisać otrzymuje sygnał SIGPIPE

Potoki: manipulacja deskryptorami

```
#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 piszący */
        close(1);           /* zamykamy stdout prawdziwy */
        dup(potok_fd[1]);    /* odzyskujemy fd 1 w potoku */
        close(potok_fd[1]);  /* dla porządku */
        close(potok_fd[0]);  /* dla porządku */
        close(0);           /* dla porządku */
        execlp("ps", "ps", "-fu", getenv("LOGNAME"), NULL);
    }
    close(potok_fd[1]);     /* ważne */
    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

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 ($\geq \text{PIPE_BUF}$ bajtów) mogą mieszać się z zapisami z innych procesów.

System V IPC

- Urządzenia komunikacyjne:
 - kolejki komunikatów: koniec–koniec
 - semafony: jednobitowe flagi
 - pamięć (współ)dzielona: wielu–wielu
- 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” procesu 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.

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);
    }
}

```

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 dostępu do kolejki komunikatow!\n");
        exit(1);
    }

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


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 wspolnastruct {
    int klient_zapisal;
    char tekst[BUFSIZ];
};

int main() {
    int pracuj = 1;
    void *pamiec_wspolna = (void *)0;
    struct wspolnastruct *wspolna;
    int shmid;

    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("Otrzymałem: %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);
}

```


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 wspolnastruct {
    int klient_zapisal;
    char tekst[BUFSIZ];
};

int main() {
    int pracuj = 1;
    void *pamiec_wspolna = (void *)0;
    struct wspolnastruct *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);
}

```

Semafor: 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(a) proces(y) oczekujący(e) na semaforze to, zamiast zwiększać wartość semafora, wznowiany 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 wyłączości. Wartość takiego semafora może wynosić 1 lub 0.

Semafore w Unixie

- mamy zbiory semaforów: `semget(key, nsems, semflg)`
- operacje na semaforach: `semop(semid, sembufops, nops)`, gdzie `sembufops` jest wskaźnikiem do tablicy `nops` struktur `sembuf` opisujących serię operacji niepodzielnych

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

- `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)

- operacje na semaforach: `semctl(semid, semnum, cmd, ...)`
pozwała ustawić określoną (początkową) wartość semafora (`cmd=SETVAL`), albo usunąć go z systemu (`cmd=IPC_RMID`)
- mechanizm UNDO: podanie flagi `SEM_UNDO` w operacjach semaforowych spowoduje „odkręcenie” operacji semaforowych dokonanych przez dany proces, w przypadku jego śmierci

Semafore w Unixie: 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[0], 2) < 0)  
        perror("semop lock error");  
}
```

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


Gniazdka domeny Unix: przykład

```
#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(wniazdka[1]);
        write(wniazdka[0], "Uszanowanie dla rodzica", 23);
        read(wniazdka[0], buf, BUFSIZ)
        printf("Potomek odczytal: %s\n", buf);
        close(wniazdka[0]);
    }
    else {                                             /* rodzic */
        close(wniazdka[0]);
        read(wniazdka[1], buf, BUFSIZ)
        printf("Rodzic odczytal %s\n", buf);
        write(wniazdka[1], "Pozdrowienia dla potomka", 24);
        close(wniazdka[1]);
    }
}
```

Gniazdko domeny Unix: wprowadzenie

- Gniazdko są deskryptorami plików umożliwiającymi dwukierunkową komunikację w konwencji połączeniowej (strumień bajtów) lub bezpołączeniowej (przesyłanie pakietów), w ramach jednego systemu Unixa lub przez sieć komputerową 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 (gniazdko domeny Unix) adresy określone są przez ścieżki plików na dysku komputera, co umożliwia ogłoszenie adresu serwera.
- W ogólnym przypadku (wyjąwszy pary gniazdek połączonych) komunikowanie się za pomocą gniazdek wymaga utworzenia i wypełnienia struktury adresowej.

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;
}
```

Gniazdko 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);                /* kolejgowanie 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_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 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 — 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 gniezdku adresu odbiorcy i kierowanie do niego całej komunikacji zapisywanej do gniazdka, ale nie powoduje żadnego nawiązywania połączenia.
- Wywołanie funkcji `close` na gniezdku połączonym powoduje rozwiązanie połączenia, a w przypadku komunikacji bezpołączeniowej rozwiązanie związku adresu z gniazdkiem.

Przykład: serwer wieloprocessowy

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

#define GNIAZDKO_SERWERA "/tmp/gniazdko_serwera"

void obsluz_klienta(int sock);

int main() {
    int serv_sock, cli_sock;
    int addr_len;
    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);
    addr_len = sizeof(addr_str);
    unlink(GNIAZDKO_SERWERA);                /* nie moze istniec */
    bind(serv_sock, (struct sockaddr *)&addr_str, addr_len); /* rejestracja */
    listen(serv_sock, 5);                     /* kolejgowanie polaczen */
}
```

```

while (1) {
    printf("Rodzic pid=%d, czekam na polaczenie\n", (int)getpid());
    cli_sock = accept(serv_sock,0,0);    /* polacz.zamiast adr.klienta*/
    if (fork() == 0) {                  /* jestem potomkiem */
        close(serv_sock);                /* to gniazdko niepotrzebne */
        obsluz_klienta(cli_sock);        /* to musze obsluzyc */
        return 0;                        /* skonczylem prace */
    }

    /* jestem rodzicem, odpowiadam tylko za odbieranie polaczen */
    close(cli_sock);
    waitpid((pid_t)-1, 0, WNOHANG);      /* obowiazki rodzicielskie */
    /* opoznienie sleep niepotrzebne, bede czekal w funkcji accept */
}
}

```

„Czyszczenie” podprocesów: ciąg dalszy

W serwerach pracujących i tworzących podprocesy w sposób ciągły istotnym staje się problem zczytywania statusu kończących pracę potomków.

Możliwe rozwiązania tego problemu:

- wywoływanie funkcji `wait` natychmiast po utworzeniu podprocesu — tak można pisać programy, jednak nie są one w żadnym stopniu współbieżne,
- okresowe wywoływanie funkcji `waitpid/wait3/wait4` z flagą `WNOHANG` — to działa poprawnie o ile te wywołania będą wystarczająco częste, np. będą równie częste jak tworzenie podprocesów, i nie będzie nadmiernych opóźnień,
- zadeklarowanie handlera sygnału `SIGCHLD` (lub `SIGCLD`) funkcją `signal` lub `sigset` i wywoływanie w nim funkcji `wait` — wtedy mamy pewność, że istnieje potomek w stanie zombie i funkcja `wait` wróci natychmiast,

```
#include <stdio.h>      /* basic I/O routines.    */
#include <unistd.h>      /* define fork(), etc.    */
#include <sys/types.h>   /* define pid_t, etc.     */
#include <sys/wait.h>    /* define wait(), etc.    */
#include <signal.h>      /* define signal(), etc.  */
```

```

void child_wait(int sig) {
    int child_status;
    signal(SIGCLD, child_wait);
    wait(&child_status);
}

main() {

    signal(SIGCHLD, child_wait);

    switch (fork()) {
        case 0:                /* inside child process */
            exit(0);

        default:                /* inside parent process */
            sleep(5);
            break;
    }
}

```

(Uwaga: jeśli zamiast zadeklarujemy handler funkcją `sigaction` to sygnał będzie generowany nie tylko przy śmierci potomka, ale również przy jego zastopowaniu i wznowieniu. Temu z kolei można zapobiec ustawiając flagę

SA_NOCLDSTOP w strukturze sigaction.)

- ustawienie flagi SA_NOCLDWAIT w strukturze sigaction przy deklarowaniu obsługi sygnału SIGCHLD (w tym przypadku sygnał może być po prostu ignorowany) — powoduje to nietworzenie procesów zombie przy śmierci potomków, i całkowity brak konieczności wywoływania funkcji wait:

```
struct sigaction sa;
sa.sa_handler = SIG_IGN;
#ifdef SA_NOCLDWAIT
    sa.sa_flags = SA_NOCLDWAIT;
#else
    sa.sa_flags = 0;
#endif
sigemptyset(&sa.sa_mask);
sigaction(SIGCHLD, &sa, NULL);
```