Procesy

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

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

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

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

Proces uniksowy i jego środowisko

- Proces uniksowy:
  - 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 innego 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 uniksowego 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 też 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 przez otrzymanie sygnału system generuje dla procesu status o wartości 128+nr_sygnału.
- Rodzic normalnie powinien po śmierci potomka odczytać jego status 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. zapełnienia 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 (chociaż 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.

Konta użytkownika

Konta użytkownika istnieją dla zapewnienia izolacji i zabezpieczenia działających procesów. W systemach uniksowych istnieje jedno główne konto użytkownika o numerze 0 zwane *root*. Główne programy zapewniające funkcjonowanie systemu, i uruchamiane przez jądro, działają w ramach konta użytkownika root.

Można tworzyć dowolną liczbę innych kont, zarówno dla ludzi-użytkowników systemu, jak i dla określonych programów lub ich grup, których uruchamianie wymaga izolacji lub zabezpieczenia przez innymi użytkownikami.

Procesy jednego użytkownika nie mogą ingerować w procesy innego użytkownika. To znaczy, nie mogą ich zatrzymywać, wysyłać im sygnałów, tworzyć ani usuwać blokad, ustawiać ograniczeń zasobów, itp. Nie dotyczy to konta użytkownika root, który jest traktowany jako użytkownik nadrzędny, i jego procesy mogą ingerować w procesy innych użytkowników systemu.

Trzeba dodać, że ten mechanizm zabezpieczeń składający się ze zbioru zwykłych użytkowników, których procesy są przed sobą nawzajem zabezpieczone, oraz użytkownika *root*, przed którym nic nie jest zabezpieczone, jest dość ubogi i często niewystarczający. Występują sytuacje pośrednie, kiedy proces potrzebuje pewnych zwiększonych uprawnień, ale nie powinien posiadać nieograniczonej władzy. Ten problem jest rozwiązywany przez szereg dodatkowych mechanizmów.

Zasoby procesu

System operacyjny może i powinien ograniczać wielkość zasobów zużywanych przez procesy. Systemy uniksowe definiują szereg zasobów które mogą być kontrolowane. Na poziomie interpretera poleceń ograniczenia można ustawiać poleceniem `limit` (C-shell) lub `ulimit` (Bourne shell). Programowo do kontrolowania zasobów służą funkcje: `getrlimit/setrlimit`.

ulimit	limit	nazwa zasobu	opis zasobu
-c	coredumpsize	RLIMIT_CORE	plik core (zrzut obrazu pamięci) w kB
-d	datasize	RLIMIT_DATA	segment danych procesu w kB
-f	filesize	RLIMIT_FSIZE	wielkość tworzonego pliku w kB
-l	memorylocked	RLIMIT_MEMLOCK	obszar, który można zablokować w pamięci w kB
-m	memoryuse	RLIMIT_RSS	zbiór rezydentny w pamięci w kB
-n	descriptors	RLIMIT_NOFILE	liczba deskryptorów plików (otwartych plików)
-s	stacksize	RLIMIT_STACK	wielkość stosu w kB
-t	cputime	RLIMIT_CPU	wykorzystanie czasu CPU w sekundach
-u	maxproc	RLIMIT_NPROC	liczba procesów użytkownika
-v	vmemoryuse	RLIMIT_VMEM	pamięć wirtualna dostępna dla procesu w kB

Dla każdego zasobu istnieją dwa ograniczenia: miękkie i twarde. Zawsze aktualnie obowiązujące jest ograniczenie miękkie, i użytkownik może je dowolnie zmieniać, lecz tylko do maksymalnej wartości ograniczenia twardego. Ograniczenia twarde można również zmieniać, lecz procesy zwykłych użytkowników mogą je jedynie zmniejszać. Ograniczenia zasobów są dziedziczone przez podprocesy.

Priorytety procesów

W zakresie dostępu do procesora obowiązuje podstawowa zasada równości (sprawiedliwości), zapewniająca quasi-równoległe wykonywanie procesów.

Procesy uniksowe mają również **priorytety** określające kolejność ich wykonywania w procesorze. Priorytety mają znaczenie nadrzędne nad zasadą równości. To znaczy, proces o niższym priorytecie zawsze musi czekać na zakończenie lub wstrzymanie procesu o priorytecie wyższym.

Wykonywanie procesów o równych priorytetach realizowane jest zgodnie z zasadą równości.

Dynamiczna modyfikacja priorytetów

System realizuje algorytm kolejkowania wykonywania procesów, zwany **planistą** (*scheduler*). Planista uniksowy opiera się na priorytetach procesów, ale jednocześnie może je zmieniać. Ogólnie, procesy interakcyjne zyskują wyższe priorytety, ponieważ mają związek z pracą człowieka, który nie chce czekać na reakcję komputera, ale jednocześnie pracuje z przerwami, które zwykle pozwalają na wykonywanie innych procesów.

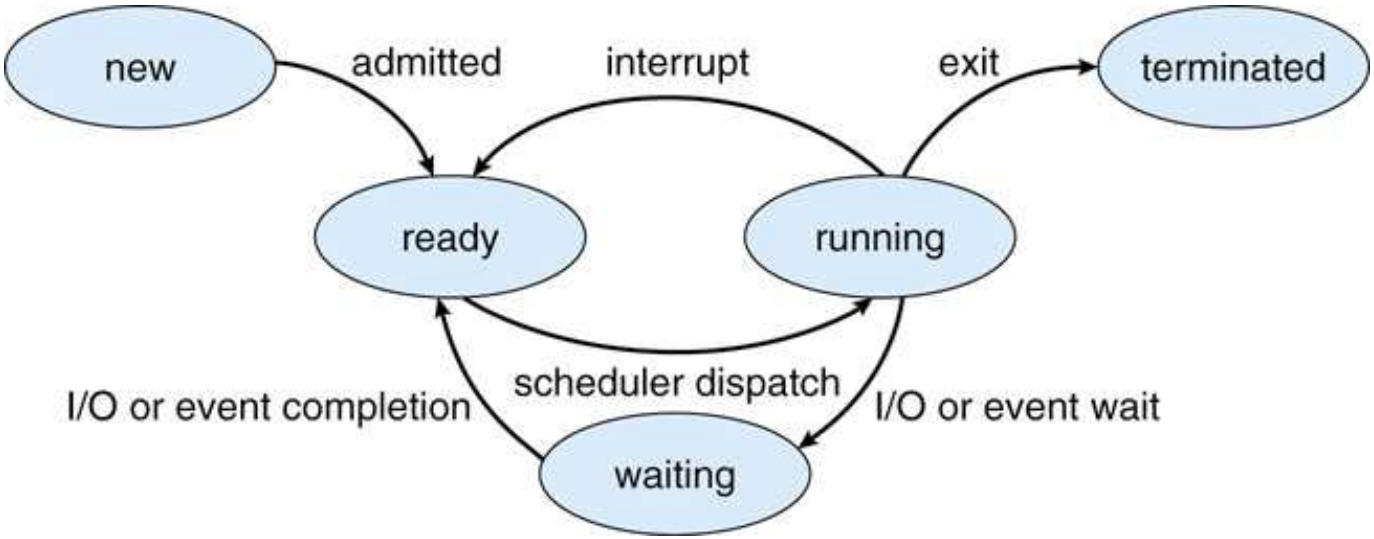
Wszelkie ingerencje w pracę planisty są zaawansowanymi operacjami, dla zwykłych użytkowników trudnymi i niebezpiecznymi. Jednak w systemach uniksowych istnieje prosty mechanizm pozwalający użytkownikowi wspomagać planistę w wyznaczaniu priorytetów procesów. Tym mechanizmem są liczby **nice** dodawane do priorytetów. Użytkownik może uruchamiać procesy z zadaną liczbą nice (służy do tego polecenie `nice`), lub zmieniać liczbę nice wykonujących się procesów (polecenie `renice`). Co prawda zwykli użytkownicy mogą tylko zwiększać domyślną wartość nice co obniża priorytet procesu. Jednak uruchamiając procesy „obliczeniowe" z priorytetem obniżonym można efektywnie jakby zwiększyć priorytet procesów „interakcyjnych".

Stany procesów

wykonywalny (runnable) — proces gotowy do wykonania, może być:

wykonywany (running) — wykonujący się na procesorze
gotowy (ready) — oczekujący w kolejce na wykonanie

oczekujący/uśpiony (waiting/sleeping) — proces na coś czeka, np. na dane do przeczytania, na sygnał, na dostęp do zasobu, lub dobrowolnie zawiesił się

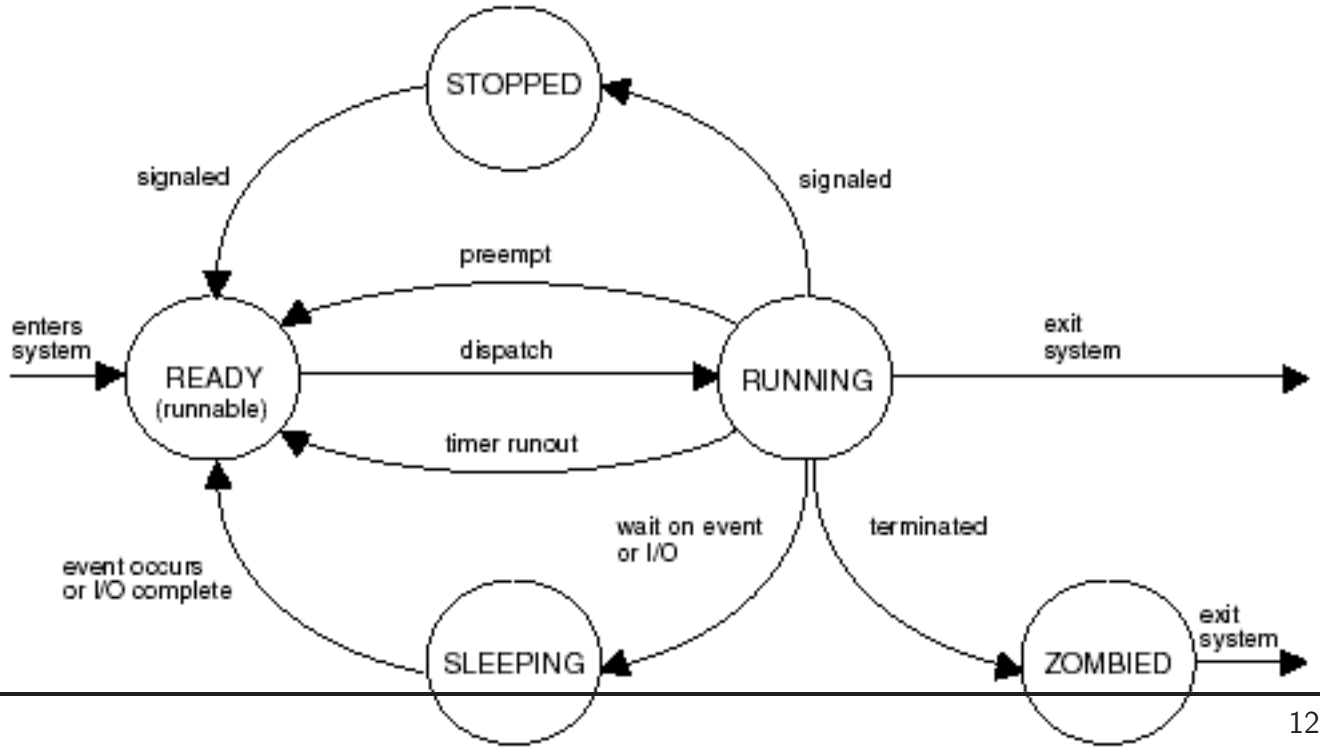


System przenosi proces w stan uśpienia jeśli nie może mu czegoś dostarczyć, np. dostępu do pliku, danych, itp. W stanie uśpienia proces nie zużywa czasu procesora, i jest *budzony* i wznawiany w sposób przez siebie niezauważony.

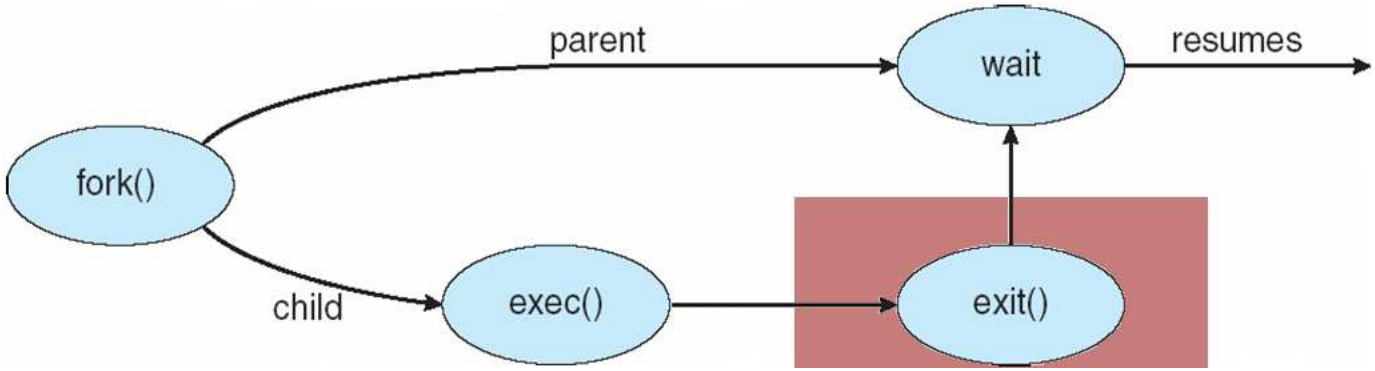
zatrzymany (stopped) — proces gotowy do wykonywania lecz zatrzymany na skutek otrzymania sygnału SIGSTOP lub SIGTSTP, może to być skutek: dobrowolnego żądania procesu, wysłania sygnału (np. przez użytkownika), lub odwołania się procesu pracującego w tle do terminala sterującego; wznowienie pracy procesu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

nieusuwalny (zombie) — proces nie może się zakończyć

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 — pozostaje on w jednym z powyższych właściwych stanów; wymiecenie procesu jest skutkiem braku dostatecznej ilości pamięci fizycznej na jednoczesne skuteczne wykonywanie wszystkich procesów wykonywalnych



Tworzenie procesów — funkcja `fork`




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

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

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

Tworzenie procesów — funkcja `fork` (cd.)

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

Własności procesu i podprocesu

Dziedziczone (podproces posiada kopię atrybutu procesu):

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

Wspólne (podproces współdzieli atrybut/zasób z procesem):

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

Różne:

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

Atrybuty procesu, które nie zmieniają się po exec:

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

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Jaka jest rola procesów w systemach operacyjnych?

2. W jaki sposób tworzone są procesy w modelu Unix/POSIX?

3. Co to jest status procesu, jak jest tworzony, i co się z nim potem dzieje?

4. Jakie związki istnieją pomiędzy procesem rodzicielskim a potomkiem?

5. W jakim celu ustawiane są ograniczenia zasobów procesu?

6. W jaki sposób na priorytet procesu wpływa ustawienie liczby nice?

7. Jakie mechanizmy lub zjawiska powodują przechodzenie procesu między stanami: wykonywalnym i uśpionym?

Sygnały

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

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

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

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

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

Mechanizmy generowania sygnałów

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

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

Reakcje na sygnał

Proces może zadeklarować jedną z następujących możliwości reakcji na sygnał:

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

W przypadku skryptów zestaw możliwych reakcji na sygnał jest bardziej ograniczony (np. Bourne shell: polecenie trap).

Dla sygnałów SIGKILL i SIGSTOP/SIGCONT proces nie może w żaden sposób zmienić domyślnej reakcji na sygnał.

Obsługa sygnałów

Jakkolwiek procedura obsługi sygnału może być napisana dowolnie i podjąć dowolne akcje, typowa funkcja handlera jest krótka i ogranicza się do wykonania minimalnych, niezbędnych w związku z powstałym zdarzeniem czynności, np. zamknięcia plików zapisywanych, skasowania plików roboczych, itp.

Po ich wykonaniu handler ma do wyboru następujące opcje:

- zakończenie procesu,
- kontynuowanie procesu od miejsca przerwania, jednak niektórych funkcji systemowych nie można kontynuować, i zostają one przedterminowo zakończone (ale proces kontynuuje),
- wznowienie procesu od określonego zapamiętanego wcześniej punktu (funkcje `setjmp()/longjmp()`); przy czym zapamiętany zostaje jedynie stos obliczeń, ale nie wartości zmiennych globalnych itp. stan (np. deskryptory plików nie cofają wykonanych operacji wejścia/wyjścia).

W przypadku kontynuowania lub wznowienia procesu istnieje możliwość przekazania informacji o fakcie obsługi sygnału przez ustawienie jakiejś zmiennej globalnej.

Sygnały — funkcje obsługi (tradycyjne)

Istnieje kilka różnych modeli generowania i obsługi sygnałów. Jednym z nich jest tzw. model tradycyjny, pochodzący z wczesnych wersji Unixa. Jego cechą charakterystyczną jest, że zadeklarowanie innej niż domyślna obsługi sygnału ma charakter jednorazowy, tj., po otrzymaniu pierwszego sygnału przywracana jest reakcja domyślna.

Funkcje tradycyjnego modelu obsługi sygnałów:

- `signal()` – deklaruje jednorazową reakcję na sygnał: ignorowanie (`SIG_IGN`), obsługa, oraz reakcja domyślna (`SIG_DFL`)
- `kill()` – wysyła sygnał do innego procesu
- `raise()` – wysyła sygnał do własnego procesu
- `pause()` – czeka na sygnał
- `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 — 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,
          "Sygnal %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);               /* zdazylismy */
  }
  else
    printf("Program wznowiony po ");
    printf("przerwaniu sygnalem %d\n", syg);
```

Obsługa sygnałów — inne modele

Tradycyjny model generowania i obsługi sygnałów jest uproszczony i nieco niewygodny (istnieje okno czasowe w czasie którego brak jest zadeklarowanego handlera). Dlatego powstały alternatywne modele funkcjonowania sygnałów. Niestety, są one wzajemnie niekompatybilne.

Model BSD nie zmienia deklaracji handlera po otrzymaniu sygnału, natomiast w trakcie wykonywania handlera kolejne sygnały tego samego typu są automatycznie blokowane. Deklaracji handlera w tym modelu dokonuje się funkcją `sigvec`.

Niezawodny model Systemu V ma podobne własności do modelu BSD, tzn. również pozwala na trwałą deklarację handlera. Wprowadza więcej funkcji i opcji obsługi sygnałów. Do deklaracji handlera w tym modelu służy funkcja `sigset()`.

Pomimo iż powyższe modele istnieją w wielu systemach uniksowych, ich stosowanie nie ma sensu w programach, które mają być przenośne. Sytuację uporządkował dopiero nowy model generowania i obsługi sygnałów — model POSIX.

Obsługa sygnałów — model POSIX

Standard POSIX zdefiniował nowy model generowania sygnałów i zestaw funkcji oferujący możliwość kompleksowego i bardziej elastycznego deklarowania ich obsługi:

- funkcja `sigaction()` – deklaruje reakcję na sygnał (trwale)
- 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);
  }
}
```

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Co to są sygnały i jak są generowane?

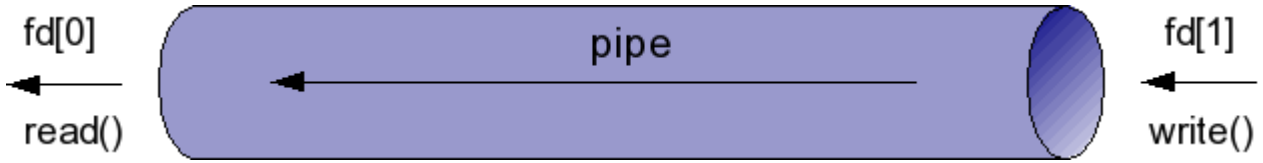
2. Jakie są możliwe reakcje procesu na otrzymanie sygnału?

3. W jaki sposób proces może przeprowadzić obsługę sygnału?

Komunikacja międzyprocesowa — potoki

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

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



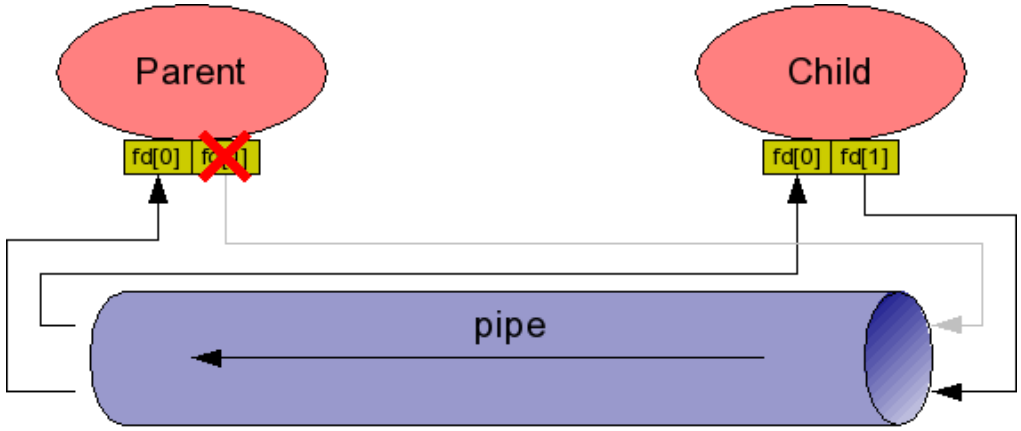
Dobrą analogią potoku jest rurka, gdzie strumień danych jest odbierany jednym końcem, a wprowadzany drugim. Gdy rurka się zapełni i dane nie są odbierane, nie można już więcej ich wprowadzić.

Potoki: funkcja `pipe`

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

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

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

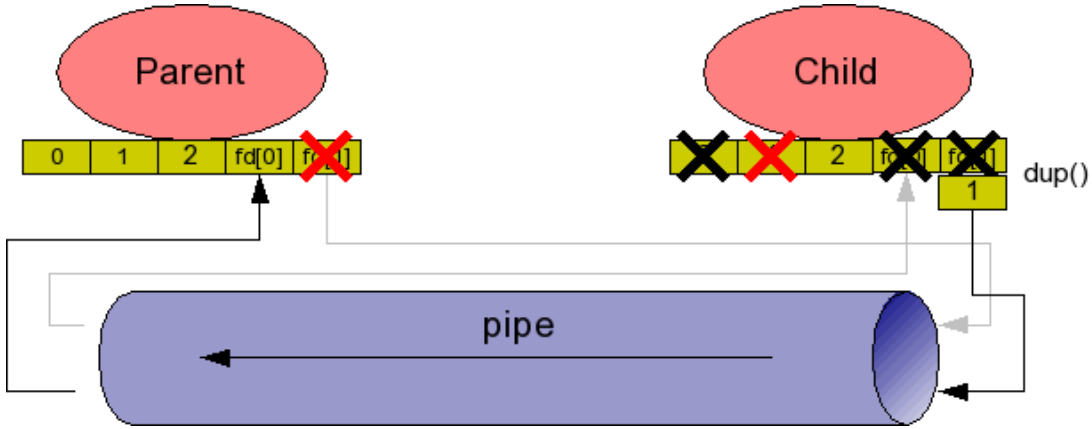


Funkcja `read()` zwraca 0 na próbie odczytu z potoku zamkniętego do zapisu, lecz jeśli jakiś proces w systemie ma ten potok otwarty do zapisu to funkcja `read()` „zawisa” na próbie odczytu.

Potoki: zasady użycia

- Potok (anonimowy) zostaje zawsze utworzony otwarty, i gotowy do zapisu i odczytu.
- Próba odczytania z potoku większej liczby bajtów, niż się w nim aktualnie znajduje, powoduje przeczytanie dostępnej liczby bajtów i zwrócenie w funkcji `read()` liczby bajtów rzeczywiście przeczytanych.
- Próba czytania z pustego potoku, którego koniec piszący jest nadal otwarty przez jakiś proces, powoduje „zawiśnięcie" funkcji `read()`, i powrót gdy jakieś dane pojawią się w potoku.
- Czytanie z potoku, którego koniec piszący został zamknięty, daje natychmiastowy powrót funkcji `read()` z wartością 0.
- Zapis do potoku odbywa się poprawnie i bez czekania pod warunkiem, że nie przekracza pojemności potoku; w przeciwnym wypadku `write()` „zawisa" aż do ukończenia operacji.
- Próba zapisu na potoku, którego koniec czytający został zamknięty, kończy się porażką i proces piszący otrzymuje sygnał SIGPIPE.
- 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`
- próba otwarcia potoku jeszcze nieotwartego w trybie komplementarnym zawisa i czeka aż do innej próby otwarcia w drugim trybie

SERWER:

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

void main() {
  int potok_fd;

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

KLIENT:

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


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

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

Operacje na FIFO

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

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

Potoki — podsumowanie

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

Podstawowe zalety potoków to:

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

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

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Przedstaw ogólne własności i działanie potoków.

2. Jaki wynik może przynieść próba odczytu danych z pustego potoku?

3. Wyjaśnij w jaki sposób komunikacja przez potok może synchronizować pracę procesów.

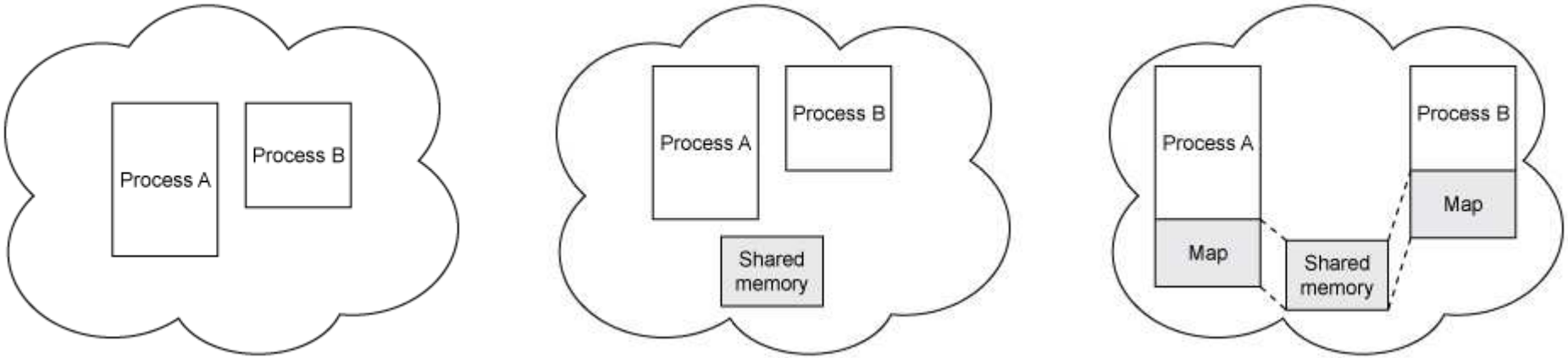
4. Wyjaśnij w jaki sposób procesy mogą nawiązać komunikację przez potok anonimowy, a w jaki przez potok nazwany.

5. Wyjaśnij co to znaczy, że potoki są mechanizmem komunikacji 1-1.

Komunikacja międzyprocesowa — obszary pamięci wspólnej

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

Komunikacja przez pamięć wspólną wymaga synchronizacji za pomocą oddzielnych mechanizmów, takich jak muteksy albo blokady zapisu i odczytu.



Obrazki zapożyczone bez zezwolenia z:

`http://www.ibm.com/developerworks/aix/library/au-spunix_sharedmemory/`

Mechanizmy komunikacji międzyprocesowej standardu POSIX Realtime opierają identyfikację 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 jakiejś wirtualnej przestrzeni nazw, natomiast większym systemom na osadzenie ich w systemie plików według dowolnej konwencji.

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.

Kroki niezbędne przy komunikacji przez pamięć współdzieloną:

1. Otwarcie/utworzenie pliku obszaru pamięci wspólnej `shm_open()`
2. Ewentualnie ustalenie rozmiaru obszaru pamięci wspólnej `ftruncate()`
3. Odwzorowanie obszaru pamięci wspólnej do obszaru w pamięci procesu `mmap()`
4. Praca: operacje wejścia/wyjścia `read()/write()`
5. Skasowanie odwzorowania `munmap()`
6. Ewentualnie skasowanie pliku obszaru pamięci wspólnej `sh_unlink()`

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;
}
```

Komunikacja międzyprocesowa — semafory i muteksy

W teorii semafor jest mechanizmem synchronizacyjnym, domyślnie kontrolującym dostęp lub przydział pewnego zasobu. Wartość semafora 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. Operacje semaforowe realizowane są przez system operacyjny, który zapewnia ich niepodzielność.

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

Semafory POSIX

```
#include <semaphore.h>

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

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

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

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Jakie właściwości ma komunikacja procesów przez pamięć wspólną?

2. Dlaczego komunikacja przez pamięć wspólną wymaga synchronizacji?

3. W jaki sposób semafor albo muteks może zostać użyty do synchronizacji komunikacji procesów przez pamięć wspólną?

Uzupełnienie: deskryptory plików

```
#include <stdio.h>        /* wersja 5: operacje IO */
#include <fcntl.h>        /* niskiego poziomu      */

main(int argc, char *argv[])
{
  char buf[1024];
  int fd1, fd2, fdin;

  fd1 = open(argv[1], O_RDONLY, 0);
  fd2 = open(argv[2], O_WRONLY|O_CREAT, 0644);
  while ((fdin = read(fd1, buf, sizeof buf)) > 0)
    write(fd2, buf, fdin);
  exit(0);
}
```

- Funkcje `open()`, `read()`, `write()`, zwane wbudowanym systemem wejścia/wyjścia, lub systemem I/O niskiego poziomu, są głównym mechanizmem operacji na plikach i urządzeniach w Uniksie.

- Odwołują się one do otwartych plików przez numery deskryptorów.

Deskryptory plików — zasady

- Deskryptory są małymi liczbami przyporządkowanymi kolejno otwieranym plikom. Program ma gwarancję uzyskania najniższego wolnego deskryptora. Deskryptory 0,1,2 reprezentują automatycznie otwierane pliki stdin, stdout, i stderr.
- Numery deskryptorów otwartych plików są lokalne dla danego procesu. Jeśli inny proces otworzy ten sam plik to otrzyma własny (choć być może ten sam) numer deskryptora, z którym związany jest kursor pliku wskazujący pozycję wykonywanych operacji zapisu/odczytu.
- Operacje odczytu pliku otwartego jednocześnie przez dwa procesy mogą poprawnie przebiegać niezależnie od siebie.
- Jeśli dwa procesy jednocześnie otworzą ten sam plik do zapisu (lub zapisu i odczytu), to ich operacje zapisu będą się na siebie nakładały, jedne dane nadpiszą drugie, i otrzymamy w pliku tzw. sieczkę.
- Jednak dwa procesy mogą współdzielić jeden deskryptor otwartego pliku, i wtedy ich operacje będą się ze sobą przeplatać. Jeśli nie będziemy tego przeplatania kontrolować to również możemy otrzymać sieczkę, aczkolwiek w tym przypadku dane nie zostaną nadpisane.
- W każdym przypadku do synchronizowania operacji wejścia/wyjścia na plikach możemy użyć blokad.

Funkcje niskiego poziomu a biblioteka `stdio`

Funkcje I/O niskiego poziomu są innym, bardziej surowym, sposobem wykonywania operacji wejścia/wyjścia na plikach niż biblioteka `stdio`.

- Operacje I/O niskiego poziomu nie mają dostępu do struktur danych biblioteki `stdio` i w odniesieniu do danego otwartego pliku nie można ich mieszać z operacjami na poziomie tej biblioteki.
- Jednak w czasie pracy z biblioteką `stdio` funkcje niskiego poziomu też pracują (na niższym poziomie), i można korzystać z niektórych z nich, jeśli się wie co się robi.
- Na przykład, numer deskryptora pliku można uzyskać z file pointera biblioteki `stdio` (`fileno(fp)`), ale nie na odwrót (dlaczego?).

Szczegóły działania funkcji `read, write`

- Przy czytaniu funkcją `read()` liczba znaków przeczytanych może nie być równa liczbie znaków żądanych ponieważ w pliku może nie być tylu znaków.
  - W szczególności przy czytaniu z terminala funkcja `read()` normalnie czyta tylko do końca linijki.
  - Wartość 0 oznacza brak danych do czytania z pliku (koniec pliku), co jednak nie oznacza, że przy kolejnej próbie czytania jakieś dane się nie pojawią (inny proces może zapisać coś do pliku), zatem nie jest to błąd.
  - Wartość ujemna oznacza błąd.
- Przy pisaniu funkcją `write()` wartość zwrócona jest normalnie równa liczbie znaków żądanych do zapisu; jeśli nie to wystąpił jakiś błąd.
- Drugi parametr funkcji `open` może przyjmować trzy wartości zadane następującymi stałymi: `O_RDONLY` (tylko czytanie), `O_WRONLY` (tylko pisanie), lub `O_RDWR` (czytanie i pisanie). Poza tym można dodać do tej wartości modyfikatory (bitowo), które zmieniają sposób działania operacji na pliku.
- Ostatni parametr funkcji `open()` zadaje 9 bitów praw dostępu dla tworzonego pliku (np. ósemkowo 0755).

Krótkie podsumowanie — pytania sprawdzające

Odpowiedz na poniższe pytania:

1. Co to są deskryptory plików?

2. W jaki sposób przydzielane są numery deskryptorów kolejno otwieranych plików?

3. Jaki jest związek deskryptorów plików z wskaźnikami plików biblioteki stdio?

4. Jakie wartości zwracają funkcje read() i write()?

5. W jakich przypadkach funkcja read() może przeczytać inną liczbę znaków niż deklarowana wielkość bufora?

6. W jakich przypadkach funkcja write() może zapisać inną liczbę znaków niż zadana argumentem?