Unix: programowanie procesów

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

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

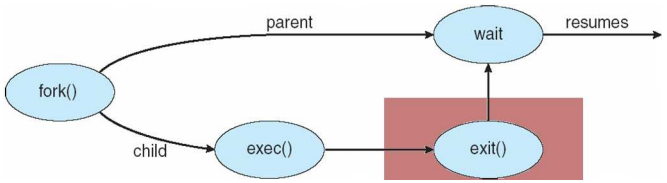
Tworzenie procesów — funkcja `system`

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

int main() {
  char komenda[BUFSIZ];

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

- Funkcja `system` powoduje wykonanie polecenia w podprocesie.
- Polecenie jest interpretowane i wykonywane przez interpreter poleceń, i może wykorzystywać wszystkie mechanizmy: skierowania, potoki, itd.
- Podproces współdzieli z procesem wywołującym dostęp do plików, w tym do terminala (`stdin, stdout, stderr`).
- W czasie wykonywania podprocesu proces wywołujący czeka.
- Funkcja `system` zwraca wartość statusu zwróconą przez interpreter poleceń, która zwykle jest statusem wykonanego polecenia.

Tworzenie procesów — funkcja `fork`

```
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 fork, pid potomka = %d\n", pid);
}
```



Tworzenie procesów — funkcja `popen`

```
#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));
}
```

- Funkcja `popen` powoduje wykonanie w podprocesie polecenia interpretowanego przez interpreter poleceń, podobnie jak `system`.
- Funkcja `popen` tworzy jednokierunkowy potok komunikacyjny skierowany od podprocesu do procesu wywołującego (argument `"r"`), lub od procesu wywołującego do podprocesu (argument `"w"`).
- Proces wywołujący posiada dodatkowy dostęp do potoku, gdy podproces ma potok przypisany jako swój plik `stdout`, lub `stdin`, odpowiednio.
- Proces wywołujący nie czeka tutaj na zakończenie podprocesu, tylko oba procesy pracują równolegle.

- Funkcja `fork` tworzy podproces, który jest kopią (klonem) procesu macierzystego.
- Po sklonowaniu się podproces może wywołać jedną z funkcji grupy `exec` i rozpocząć w ten sposób wykonywanie innego programu.
- W tym przypadku nie ma polecenia do wykonania, ani interpretera poleceń; jest tylko program i jego wektor argumentów.
- Po sklonowaniu się oba procesy współdzielą dostęp do plików otwartych przed sklonowaniem.
- Funkcja `fork` jest podstawową (i jedyną) metodą tworzenia procesów.

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

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+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. 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.

Stany procesów

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

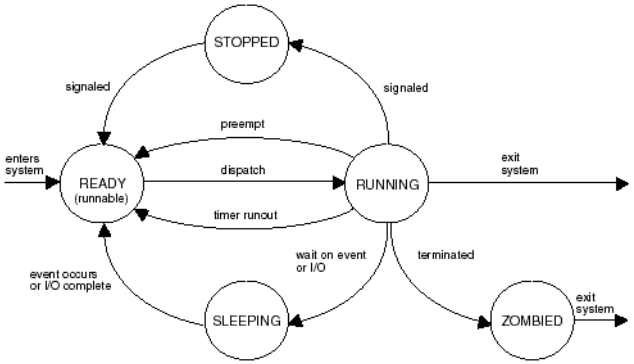
wykonywany (*running*) — wykonujący się na procesorze
gotowy (*ready*) — oczekujący w kolejce na przydział procesora

oczekujący/uśpiony (*waiting/sleeping*) — proces na coś czeka, np. na dane do przeczytania, na dostęp do zasobu, na sygnał, lub na upływ określonego czasu; system przenosi proces w stan uśpienia jeśli nie może mu czegoś dostarczyć, np. dostępu do danych, zasobów, 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 arbitralnie zatrzymany na skutek otrzymania sygnału `SIGSTOP` lub `SIGTSTP`, lub odwołania się procesu pracującego w tle do terminala sterującego; wznowienie pracy procesu następuje po otrzymaniu sygnału `SIGCONT`

nieusuwalny (*zombie*) — proces zakończony, czeka na odczytanie statusu

---



Zasoby procesu

System operacyjny może i powinien ograniczać wielkość zasobów zużywanych przez procesy, w celu realizacji swoich funkcji zapewnienia sprawnej i efektywnej pracy.
Systemy uniksowe definiują szereg zasobów które mogą być kontrolowane.
Na poziomie interpretera poleceń ograniczenia można ustawiać poleceniem `ulimit`.
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.

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.

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 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*).

Sygnały

Sygnały są mechanizmem asynchronicznego powiadamiania procesów o zdarzeniach. Zostały początkowo przystosowane do powiadamiania o błędach, ale następnie rozszerzone do obsługi wielu innych zdarzeń.

Zdarzenia generujące sygnały:

- błędy i 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)
- funkcja `kill`
- mechanizmy software-owe (SIGALRM)

Reakcją procesu na sygnał może być: ignorowanie, wywołanie wybranej funkcji obsługi (handlera), lub natychmiastowe zakończenie (śmierć) procesu.

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

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 sygnałow:

- `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 (cd.)

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

int main() {
  if (signal(SIGINT, SIGN_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

Obsługa sygnałów — inne modele

Tradycyjny model generowania i obsługi sygnałów jest uproszczony i nieco niewygodny (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.

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 sygnalem %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 przerwaniu sygnalem %d\n", syg);
```

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);
  }
}
```

Przykład: serwer wieloprocesowy

```
#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);                 /* kolejkowanie 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);                 /* gniazdko niepotrzebne */
      serve_client(cli_sock);           /* gniazdko obslugi klienta */
      return 0;                         /* skonczylem prace */
    }

    /* jestem rodzicem, odpowiadam tylko za odbieranie polaczen */
    close(cli_sock);                    /* gniazdko niepotrzebne */
    waitpid((pid_t)-1,(int *)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:

```
while (1) {
  int status;

  cli_sock = accept(serv_sock,0,0);
  if (fork() == 0) {                    /* jestem potomkiem */
    close(serv_sock);                   /* gniazdko niepotrzebne */
    serve_client(cli_sock);             /* gniazdko obslugi klienta */
    return 0;                           /* skonczylem prace */
  }
  close(cli_sock);                      /* jestem rodzicem */
  wait(&status);                        /* obowiazki rodzicielskie */
}
```

  Ten sposób jest poprawny z punktu widzenia obsługi potomków, jednak tak napisany program nie będzie w żadnym stopniu współbieżny.

- Okresowe wywoływanie jednej z pozostałych funkcji z grupy `wait*` (`waitpid`, `waitid`, `wait3`, `wait4`, ...) z flagą `WNOHANG`, zapobiegającą oczekiwaniu na zakończenie podprocesu:

```
while (1) {
  cli_sock = accept(serv_sock,0,0);
  if (fork() == 0) {                    /* jestem potomkiem */
    close(serv_sock);                   /* gniazdko niepotrzebne */
    serve_client(cli_sock);             /* gniazdko obslugi klienta */
    return 0;                           /* skonczylem prace */
  }
  close(cli_sock);                      /* jestem rodzicem */
  while (waitpid((pid_t)-1,(int *)0,WNOHANG)>0)   /* obowiazki */
    ;                                             /* rodzicielskie */
}
```

  Ten sposób pozwala zachować współbieżność i będzie skutecznie likwidować *zombie* o ile takie wywołania funkcji `wait*` będą wystarczająco częste, i nie będzie nadmiernych opóźnień.

  (Zauważmy, że w tym przypadku funkcja `waitpid` ani nie pobiera statusu potomka, ani nie oczekuje jego zakończenia, ale jest potrzebna dla poprawnej obsługi protokołu zakończenia podprocesu.)

- 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:

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

  /* main() */
  signal(SIGCHLD, child_wait);
  while (1) {
    cli_sock = accept(serv_sock,0,0);
    if (fork() == 0) {                  /* jestem potomkiem */
      close(serv_sock); serve_client(cli_sock); return 0;
    }
    close(cli_sock);                    /* jestem rodzicem */
  }
```

  Uwaga: jeśli 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 (co więcej, 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);
```

  Ten sposób wydaje się prosty i skuteczny, jednak nie wszystkie systemy uniksowe posiadają ten mechanizm.
