Procesy i sygnały

Procesy

- ***Program*** w systemie UNIX jest plikiem zawierającym gotowy do wykonania przez komputer ciąg instrukcji oraz zestaw danych zainicjowanych przez programistę.
- ***Proces*** jest podstawową aktywną jednostką pracy zarządzaną przez system. Jest to aktualnie wykonywany program wraz z całym *kontekstem*. (Definicja tradycyjna, stworzona wtedy, kiedy nie brano pod uwagę wątków). W myśl tej definicji procesem jest każdy działający program, do którego przydzielono zasoby. Wiele procesów może wykonywać ten sam program. Życie procesu rozpoczyna się w momencie jego utworzenia.
- ***Kontekst procesu*** to stan procesu, który definiują jego instrukcje, wartości zmiennych i rejestrów oraz struktury systemowe znajdujące się w jądrze związane z wykonywaniem procesu.
- Procesy mogą być dynamicznie tworzone i usuwane. Proces tworzący nazywa się procesem ***macierzystym*** (ang. *parent process*) lub rodzicem (ang. *parent*) a utworzone przez niego nowe procesy nazywane są jego ***potomkami*** (ang. *children*). Każdy nowy proces może tworzyć kolejne procesy, wskutek czego powstaje drzewo procesów.
- Każdy proces otrzymuje unikatowy numer - ***identyfikator procesu PID*** (ang. *process identifier*), który jednoznacznie określa działający proces.
- Każdy proces ma również ***identyfikator procesu macierzystego PPID*** (ang. *parent process identifier*).
- Proces macierzysty może powołać do życia wiele procesów potomnych, ale każdy potomek ma tylko jednego przodka.
- Kiedy Unix rozpoczyna pracę uruchamia pojedynczy program (`init`), którego proces otrzymuje PID równy 1.

Podstawowe atrybuty procesu

- Każdy proces charakteryzuje się pewnymi atrybutami. Należą do nich:
  - Identyfikator procesu PID
  - Identyfikator procesu macierzystego PPID
  - Rzeczywisty identyfikator właściciela procesu
  - Rzeczywisty identyfikator grupy procesu
  - Efektywny identyfikator właściciela procesu
  - Efektywny identyfikator grupy procesu
  - Katalog bieżący i katalog główny
  - Maska tworzenia pliku
  - Identyfikator sesji
  - Terminal sterujący
  - Deskryptory otwartych plików
  - Ustalenia dotyczące obsługi sygnałów
  - Ustawienia zmiennych środowiskowych
  - Ograniczenia zasobów

Sygnały

- *Sygnał* jest to informacja dla procesu, że wystąpiło jakieś zdarzenie.
- Sygnały mogą być wysyłane:
  - z procesu do innego procesu (grupy procesów)
  - z procesu do siebie samego
  - z jądra do procesu
- Sygnały są wysyłane:
  - za pomocą funkcji systemowej `kill`
  - za pomocą polecenia `kill`
  - za pomocą klawiatury - tylko wybrane sygnały
  - przez pewne sytuacje wyjątkowe wykrywane przez oprogramowanie systemowe
  - przez pewne sytuacje wyjątkowe wykrywane przez sprzęt
- Proces może wysłać sygnał do innego procesu tylko wtedy, kiedy ten proces ma takim sam efektywny identyfikator co nadawca sygnału. Wyjątkiem jest proces działający z EUID równym 0.
- Sygnał jest *dostarczony* (ang. *delivered*) do procesu, gdy proces podejmuje akcję obsługi sygnału.
- W przypadku odebrania sygnału proces może:
  - zezwolić na domyślną obsługę sygnału,
  - zignorować sygnał (są sygnały, których nie można ignorować - `SIGKILL` i `SIGSTOP`)
  - obsłużyć sygnał samodzielnie
- Działanie domyślne – czynności podejmowane przez jądro, gdy pojawi się sygnał. Są to:
  - zakończenie (ang. *termination*)
  - ignorowanie (ang. *ignoring*)
  - zrzut pamięci (ang. *core dump*)
  - zatrzymanie(ang. *stopped*)

- Każdy sygnał ma nazwę oraz numer. Są one opisane w pliku nagłówkowym `<sys/signal.h>`.
- Przykładowe sygnały (numeracja dotyczy systemu Linux), pełny zestaw: `man 7 signal`

Nazwa	Numer	Znaczenie	Czynność domyślna
SIGHUP	1	Zerwanie łączności z terminalem	Zakończenie
SIGINT	2	Przerwanie (może być generowane z klawiatury)	Zakończenie
SIGQUIT	3	Zakończenie (może być generowane z klawiatury)	Zrzut pamięci i zakończenie
SIGILL	4	Nielegalna instrukcja sprzętowa	Zrzut pamięci i zakończenie
SIGABRT	6	Wysyłany przez funkcję `abort()`	Zrzut pamięci i zakończenie
SIGFPE	8	Wyjątek arytmetyczny (np. dzielenie przez 0)	Zrzut pamięci i zakończenie
SIGKILL	9	Zakończenie (nie da się przechwycić ani zignorować)	Zakończenie
SIGSEGV	11	Niepoprawne wskazanie pamięci	Zrzut pamięci i zakończenie
SIGPIPE	13	Zapis do potoku zamkniętego z jednej strony (nikt nie czyta)	Ignorowany
SIGALRM	14	Pobudka (upłynął czas ustawiony funkcją `alarm()`)	Ignorowany
SIGTERM	15	Zakończenie programowe (domyślny sygnał polecenia `kill`)	Zakończenie
SIGCHLD	17	Zakończenie procesu potomnego	Ignorowany
SIGSTOP	19	Stop (nie da się przechwycić ani zignorować)	Zatrzymanie
SIGCONT	18	Kontynuacja wstrzymanego procesu	Ignorowany
SIGTSTP	20	Stop (dla klawiatury)	Zatrzymanie
SIGTTIN	21	Czytanie z terminala przez proces drugoplanowy	Zatrzymanie
SIFTTOU	22	Pisanie do terminala przez proces drugoplanowy	Zatrzymanie

Identyfikatory procesu

```
#include <unistd.h>
#include <sys/types.h>
```

identyfikator procesu (PID) – nadawany przez jądro systemu	`pid_t getpid(void)`
identyfikator procesu macierzystego (PPID) – nadawany przez jądro systemu	`pid_t getppid(void)`

- Przykład:

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

int main() {
  printf("Jestem procesem %ld\n", (long)getpid());
  printf("Moim rodzicem jest %ld\n", (long)getppid());
  return 0;
}
```

Wynik działania:

```
$ ./info_o_procesie
Jestem procesem 17357
Moim rodzicem jest 16805

$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
adam     16805 16804  0 14:53 pts/1    00:00:00 -bash
adam     17357 16805  0 16:12 pts/1    00:00:00 ./info_o_procesie
```

- Domyślna maksymalna wartość PID w Linuksie

```
$ cat /proc/sys/kernel/pid_max
32768
```

Oglądanie aktywności procesów

- Aktualnie działające procesy można obejrzeć za pomocą polecenia:
  - `ps` – tablica zawierająca wszystkie procesy aktualnie istniejące, bez względu na stan (wszystkie systemy Unix)
  - `top` – najbardziej aktywne procesy (większość systemów Unixowych)
  - `pstree` – wykonywane procesy w postaci drzewa zależności (system Linux)

- Obowiązuje hierarchia. Wszystkie procesy są procesami potomnymi procesu `init`, który ma PID równy 1.

- Przykłady

```
$ ps
  PID TTY          TIME CMD
 1333 pts/2    00:00:00 bash
 1369 pts/2    00:00:00 ps

$ ps -e -o pid,ppid,command | more
  PID  PPID COMMAND
    1     0 init [3]
  ...
  954     1 syslogd -m 0
  958     1 klogd -x
  ...
 1107     1 xinetd -stayalive -pidfile /var/run/xinetd.pid
 1128     1 /usr/sbin/vsftpd /etc/vsftpd/vsftpd.conf
 1174     1 /usr/sbin/httpd
 1183     1 crond
  ...
 1332  1329 sshd: adam@pts/2
 1333  1332 -bash
 1374  1333 ps -e -o pid,ppid,command

$ ps -f
  UID        PID  PPID  C STIME TTY          TIME CMD
  adam      1333  1332  0 17:12 pts/2    00:00:00 -bash
  adam      1389  1333  0 17:20 pts/2    00:00:00 ps -f
```

Powoływanie do życia nowych procesów – funkcja fork()

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);
```

- Funkcja `fork` tworzy nowy proces. Nowy proces jest dokładną kopią procesu, w którym wywołana była funkcja `fork`. Proces wywołujący funkcje `fork` nazywa się *procesem macierzystym*, zaś nowy proces – *procesem potomnym*. Proces potomny otrzymuje nowy identyfikator PID. Proces macierzysty dalej wykonuje program od miejsca, a którym wywołano `fork`. Proces potomny wykonuje ten sam program od tego samego miejsca.

- Proces potomny dziedziczy większość z atrybutów procesu macierzystego. Ma:
  - Własny identyfikator procesu PID
  - Własny identyfikator procesu macierzystego PPID
  - Własne kopie deskryptorów otwartych plików

- Funkcja `fork` zwraca:
  - jeśli utworzenie nowego procesu się powiedzie
    - w procesie macierzystym zwracany jest identyfikator nowo utworzonego procesu PID,
    - w procesie potomka zwracana jest wartość 0.
  - jeśli utworzenie nowego procesu się nie powiedzie
    - w procesie macierzystym zwracana jest wartość –1, zaś zmiennej `errno` przypisywany jest kod błędu,
- Oba procesy kontynuują działanie *od instrukcji występującej po* `fork`. *Wykonują się asynchronicznie*.

Przykłady użycia funkcji `fork`

Program tworzy proces, który jest dokładną kopią procesu macierzystego.

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

int main()
{
  int i;
  int suma=0;
  fork(); /* utwórz nowy proces */
  for (i=0; i<3; i++)
  {
    printf("pid %d ppid %d i=%d\n",getpid(),getppid(),i);
    fflush(stdout);
    suma +=i;
  }
  printf("pid %d ppid %d  suma=%d\n",getpid(),getppid(),suma);
  exit(0);
}
```

- Przykładowy wynik działania:

```
pid 1360 ppid 1325   i=0
pid 1361 ppid 1360   i=0
pid 1361 ppid 1360   i=1
pid 1361 ppid 1360   i=2
pid 1361 ppid 1360   i=3
pid 1361 ppid 1360   suma=3
pid 1360 ppid 1325   i=1
pid 1360 ppid 1325   i=2
pid 1360 ppid 1325   i=3
pid 1360 ppid 1325   suma=4
```

Kiedy używamy funkcji `fork()`?

- Chcemy rozdzielić realizację zadania na dwa lub więcej procesów
- Chcemy uruchomić nowy program

Rozdzielanie zadań na procesy

Wersja a)

```
pid_t pid;
pid=fork();
if (pid == -1) {
   /* błąd - nie udało się powołać procesu potomnego */
}
else if (pid == 0) {
   /* kod wykonywany w procesie potomnym */
}
else {
   /* kod wykonywany w procesie macierzystym */
}
```

Wersja b)

```
pid_t pid;
pid=fork();
switch (pid) {
  case -1:
           /* błąd - nie udało się powołać procesu potomnego */
           break;
  case 0:
           /* kod wykonywany w procesie potomnym */
           break;
  default:
           /* kod wykonywany w procesie macierzystym */
           break;
}
```

- Przykład:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
  pid_t pid;
  switch( pid=fork())
  {
    case -1:
      printf("Nie można utworzyc procesu potomnego\n");
      exit(1);
    case 0:                /* proces nowo utworzony */
      printf("To wykonuje potomek\n");
      break;
    default:               /* proces pierwotny */
      printf("To wykonuje proces macierzysty\n");
      break;
   }
  exit(0);
}
```

Tworzenie grupy procesów

- Wersja a

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

int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   if (argc != 2){
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if ((childpid = fork()))
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

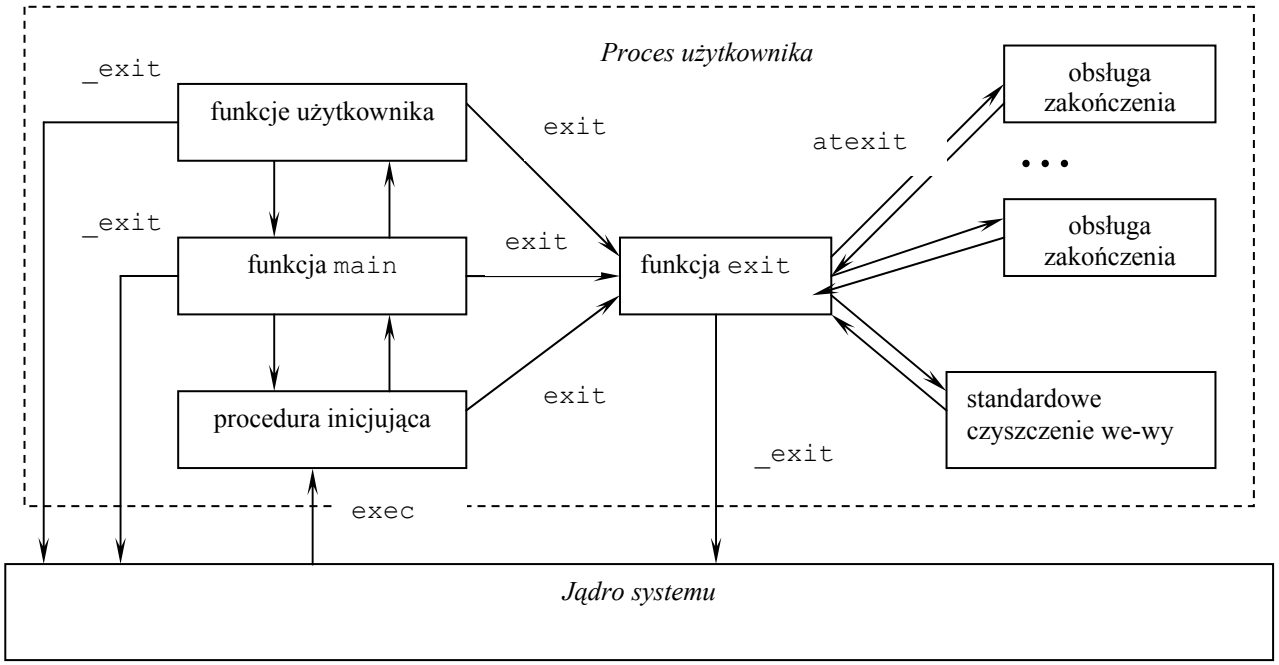
- Wersja b

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

int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   if (argc != 2){
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if ((childpid = fork()) <= 0)
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

Zadanie: narysuj schemat procesów tworzonych w przypadku a) oraz b).

Kończenie procesu

- Proces może być zakończony normalnie za pomocą:
  - zakończenia funkcji `main()`
  - wywołania `return` w funkcji `main()`
  - wywołania funkcji `exit()` w dowolnej funkcji
  - wywołania funkcji `_exit()` w dowolnej funkcji
- Proces może być zakończony przedwcześnie w wyniku:
  - wywołania funkcji `abort()`
  - otrzymania sygnału (wygenerowanego przez system, wysłanego za pomocą polecenia `kill` lub funkcji `kill()` z innego procesu, wygenerowanego z klawiatury), więcej na temat sygnałów – patrz temat Sygnały.



W. Richard Stevens: Programowanie w środowisku systemu UNIX, str. 206

- Funkcje kończące proces:

```
#include <stdlib.h>
void exit(int status);
```

Kończy proces i zwraca kod zakończenia do procesu macierzystego. Wywołuje funkcje `atexit()`. Strumienie we-wy są zamykane i czyszczone.

```
#include <unistd.h>
void _exit(int status);
```

Kończy natychmiast proces. Deskryptory otwartych plików są zamykane, do procesu macierzystego jest wysyłany sygnał SIGCHLD. Wszystkie procesy potomne kończonego procesu są przejmowane przez proces, którego identyfikator PID jest równy 1 (`init`).

```
#include <stdlib.h>
void abort();
```

Wysyła do procesu sygnał `SIGABRT`

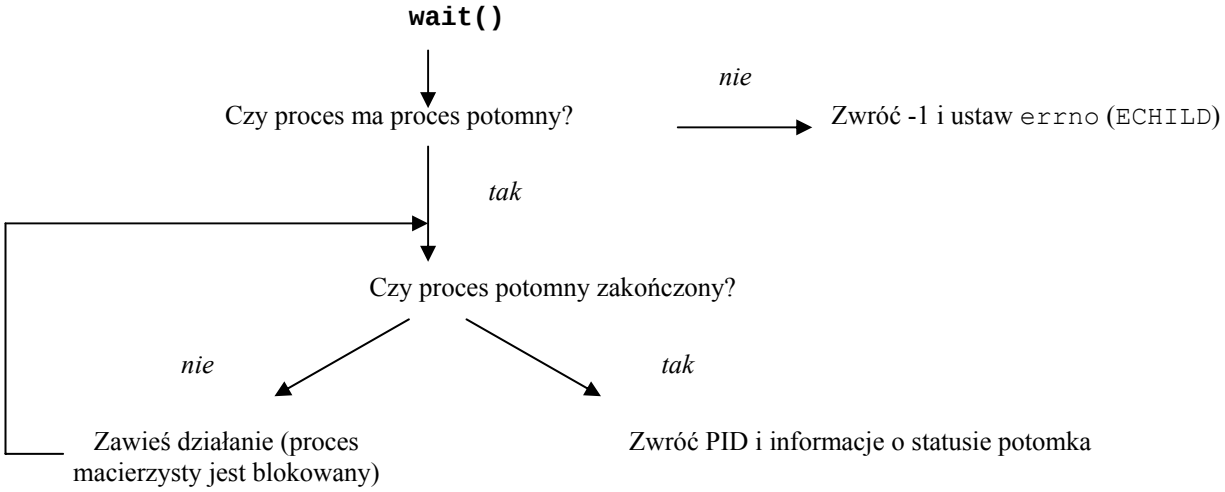
```
#include <signal.h>
int kill(pid_t pid, int sig);
```

Wysyła do procesu wskazanego za pomocą `pid` sygnał `sig`

Synchronizacja procesu macierzystego i potomnego

- Kiedy powoływany jest nowy proces, wtedy proces macierzysty może:
  - zawiesić działanie w oczekiwaniu na zakończenie działań niektórych lub wszystkich swoich procesów potomnych
  - kontynuować działanie współbieżnie ze swoimi potomkami, o zakończeniu procesu potomnego zostanie zawiadomiony za pomocą sygnału `SIGCHLD`
  - zakończyć się, pozostawiając działającego potomka.

Czekanie na zakończenie procesu potomnego



Proces zombi

- Gdy proces potomny kończy się, to do procesu macierzystego wysyłany jest sygnał `SIGCHLD` o zamiarze zakończenia procesu. Domyślnie jest on przez proces macierzysty ignorowany (patrz: `man 7 signal`).
- Kod wyjścia procesu (tworzony na przykład przez `exit()` w procesie potomnym) jest przechowywany przez jądro w tablicy procesów tak długo, aż poprosi o niego proces macierzysty.
- Proces macierzysty odbiera kod wyjścia za pomocą jednej z funkcji `wait`.
- Jeżeli proces macierzysty nie wywoła funkcji `wait`, a proces potomny zakończy się, to system zwalnia co prawda zasobów zajmowane przez potomka ale zapis w tablicy procesów pozostaje. Potomek staje się procesem-duchem (ang. *zombie*).
- Zombie będzie dopiero wtedy usunięty z tablicy procesów, kiedy zostanie odczytany jego kod wyjścia.

Proces sierota

- Jeśli proces macierzysty kończy działanie, to proces potomny staje się sierotą (ang. *orphan*).
- Jest on wtedy adoptowany przez proces `init`.
- Adoptujący rodzic `init` odbierze wartości kodów wyjścia osieroconych procesów z tablicy procesów.

Funkcje systemowe `wait`

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status)
pid_t waitpid(pid_t pid, int *status, int options)
```

- Funkcja `wait` zawiesza wykonywanie procesu macierzystego do momentu otrzymania informacji o zakończeniu działania *jednego* z jego potomków (sygnał `SIGCHLD`). Funkcja zwraca PID tego procesu potomnego.
- Funkcja `waitpid` ma opcje, które pozwalają określić czy należy czekać na zakończenie potomka. Można również określić na jaki proces funkcja ma czekać.

Argumenty

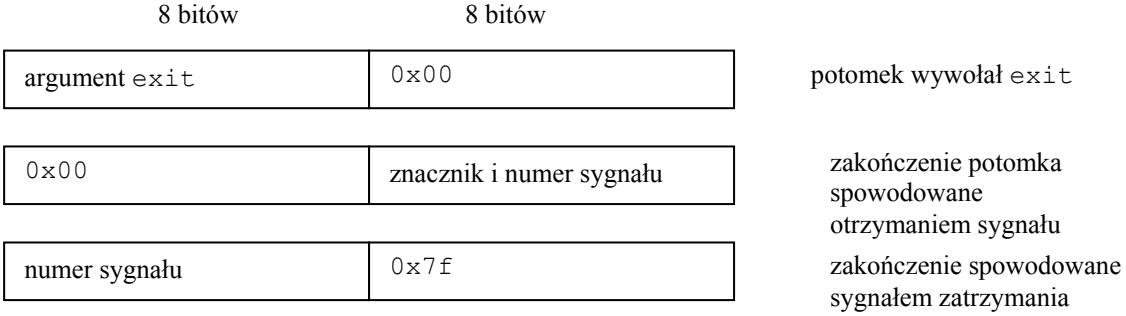
status	wskaźnik do zmiennej, w której zapisana jest informacja o stanie końcowym procesu potomnego; NULL oznacza, że nie interesuje nas status zakończonego procesu potomnego
pid	identyfikator PID procesu, na który funkcja czeka; -1 oznacza dowolny proces; 0 oznacza proces, które identyfikator grupy jest równy identyfikatorowi grupy procesu wywołującego
options	określenie sposobu zachowania funkcji, 0 oznacza pominięcie opcji, jeśli tym argumentem będzie `WNOHANG` - funkcja nie będzie blokowana, nawet jeżeli nie można pobrać informacji o stanie wskazanego potomka

Wartość zwracana

- Wartością zwracaną jest identyfikator procesu potomnego, który został zakończony. W przypadku błędu przekazują wartość –1 i ustawiają zmienną `errno`.
- Funkcja `waitpid` zwraca 0, jeśli użyta zostanie opcja `WNOHANG` i system stwierdzi, że istnieją działające procesy potomne.

Sprawdzanie statusu potomka

- Sposób wypełnienia zmiennej `status` zależy od implementacji. Przykład:



- Status potomka można sprawdzać za pomocą odpowiednich makrodefinicji zawartych w pliku nagłówkowym `sys/wait.h`. Argumentem makra jest zmienna ze statusem końcowym procesu.

Makro	Działanie
`WIFEXITED(`*`status`*`)`	Zwraca wartość niezerową (prawda), jeśli proces zakończył się normalnie.
`WEXITSTATUS(`*`status`*`)`	Jeśli proces zakończył się normalnie, to zwraca kod wyjścia procesu potomnego.
`WIFSIGNALED(`*`status`*`)`	Zwraca wartość niezerową (prawda), jeśli proces zakończył się z powodu otrzymania sygnału.
`WTERMSIG(`*`status`*`)`	Zwraca numer sygnału, który spowodował zakończenie
`WIFSTOPPED(`*`status`*`)`	Zwraca wartość niezerową (prawda), jeśli proces został zatrzymany z powodu otrzymania sygnału.
`WSTOPSIG(`*`status`*`)`	Zwraca numer sygnału, który spowodował zatrzymanie

Przykład

A. Czekamy na zakończenie potomka, nie interesuje nas jego status zakończenia

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

int main() {
   switch( fork() ) {
   case 0:  /* Proces potomny */
      printf("pid potomka: %d ppid: %d\n",getpid(),getppid());
      exit(0);
   case -1:
      printf("Blad funkcji fork\n");
      exit (1);
    default: /* Proces macierzysty */
      printf("pid przodka: %d ppid: %d\n",getpid(),getppid());
  /* Oczekiwanie na zakończenie potomka */
       wait(NULL);
       printf("proces potomny zakonczony\n");
       exit(0);
  }
}
```

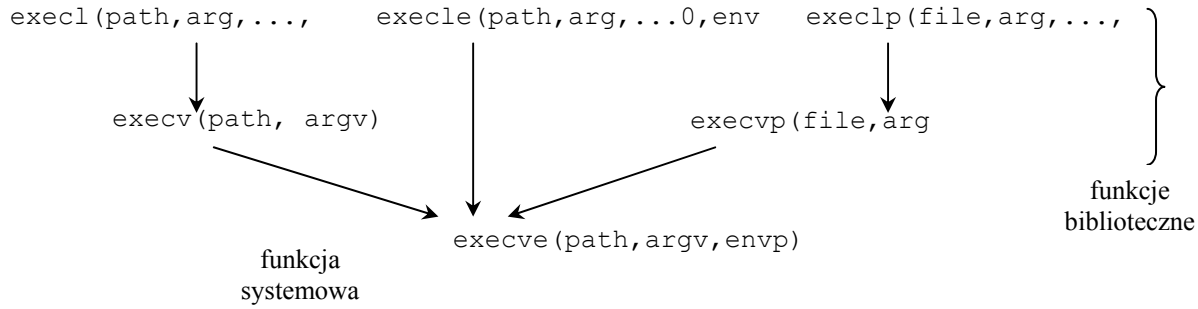
B. Czekamy na zakończenie potomka, interesuje nas jego status zakończenia

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

int main() {
  switch( fork() ) {
   case 0:  /* Proces potomny */
     printf("PID potomka: %d PPID: %d\n",getpid(),getppid());
     exit(0);
   case -1:
     printf("Blad funkcji fork\n");
     exit (1);
   default: /* Proces macierzysty */
     printf("PID przodka: %d PPID: %d\n",getpid(),getppid());
  /* Oczekiwanie na zakończenie potomka */
     {
       int status;
       pid_t pid_potomka;
       pid_potomka = wait(&status);
       printf("Potomek zakończony: PID = %d\n", pid_potomka);
       if(WIFEXITED(status))
         printf("Potomek zakonczyl się kodem %d\n",
                 WEXITSTATUS(status));
       else
         printf("Potomek zakonczony w wyniku otrzymania sygnalu\n");
     }
     exit(0);
   }
                                        }
```

Uruchamianie nowego programu

- Uruchamianie programu realizowane jest w dwóch krokach:
  - tworzony jest nowy proces za pomocą funkcji `fork()`
  - kod procesu (kopia kodu procesu macierzystego) zastępowany jest nowym kodem za pomocą funkcji z rodziny `exec`

Rodzina funkcji exec




```
#include <unistd.h>
int execl(const char *path, const char *arg,...,(char *)0);
int execv(const char *path, char *const argv[]);
int execlp(const char *file,const char *arg,...,(char *)0);
int execvp(const char *file, char *const argv[]);
int execle(const char *path, const char *arg,...,(char *)0,
                                          char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

- Nie ma powrotu z pomyślnie wykonanej funkcji `exec`. Jeśli funkcja zwróci wartość –1, oznacza to, że uruchomienie nowego programu nie powiodło się, zaś zmiennej `errno` przypisywany jest kod błędu.

- Przykład: uruchomienie w procesie potomnym polecenia `ls`

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

int main() {
  pid_t potomek;
  int status;
  if ((potomek=fork()) == -1) {
    perror("Nie uruchomiono procesu potomnego");
    exit(1);
  }
  else if (potomek==0) {
    if (execl("/bin/ls","ls","-l",NULL) < 0) {
      perror("Proces potomny nie uruchomil programu /bin/ls");
      exit(1);
      }
    }
  else if (potomek != wait(NULL))
    perror("Blad w procesie macierzystym");
  exit(0);
}
```

- Przykład: uruchomienie w procesie potomnym polecenia wczytanego w wierszu wywołania

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

int main(int argc, char *argv[]) {
   pid_t childpid;

   if (argc < 2){
      fprintf (stderr, "Uzycie: %s polecenie arg1 arg2 ...\n", argv[0]);
      exit(1);
   }
   childpid = fork();
   if (childpid == -1) {
      perror("Nie uruchomiono procesu potomnego");
      exit(1);
   }
   if (childpid == 0) {
      execvp(argv[1], &argv[1]);
      perror("Proces potomny nie uruchomil programu");
      exit(1);
   }
   if (childpid != wait(NULL)) {
      perror("Blad w procesie macierzystym");
      exit(1);
   }
   exit(0);
}
```

Przykład – prosty shell

- Schemat działania:

```
while (TRUE) {
  type_prompt();
  read_command(command,params);
  pid=fork();
  if (pid < 0) {
    printf("Unable to fork");
    continue;
  }
  if (pid != 0) {
    wait(&status);
  }
  else {
    execve(commmand,params,0);
  }
}
```

- Przykładowy kod

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

const int MAX    =256;
const int CMD_MAX=10;
char *valid_cmds = " ls  ps  df ";
int
main( ){
  char  line_input[MAX], the_cmd[CMD_MAX];
  char  *new_args[CMD_MAX], *cp;
  int   i,n;
  while (1) {
    printf("cmd> ");
    if (fgets(line_input, MAX, stdin) != NULL) {
      n=strlen(line_input);
      if (line_input[n-1]=='\n') line_input[n-1]='\0';
      cp = line_input;
      i  = 0;
      if ((new_args[i] = strtok(cp, " ")) != NULL) {
        strcpy(the_cmd, new_args[i]);
        strcat(the_cmd, " ");
        if ((strstr(valid_cmds, the_cmd) - valid_cmds) % 4 == 1) {
          do {
            cp = NULL;
            new_args[++i] = strtok(cp, " ");
          } while (i < CMD_MAX && new_args[i] != NULL);
          new_args[i] = NULL;
          switch (fork( )) {
          case 0:
            execvp(new_args[0], new_args);
            perror("exec failure");
            exit(1);
          case -1:
            perror("fork failure");
            exit(2);
```

```
          default:
            // In the parent we should be waiting for
            // the child to finish
            wait(NULL);
            ;
          }
        } else
          printf("?\n");
      }
    }
  }
}
```

Przykład działania:

```
$ ./program
cmd> ps
  PID TTY          TIME CMD
 1061 pts/2    00:00:00 bash
 1653 pts/2    00:00:00 p27
 1654 pts/2    00:00:00 ps
cmd> df
System plików      bl.  1K B        użyte dostępne %uż. zamont. na
/dev/sda3               988244    106324    831720  12% /
/dev/sda2               101105      8816     92289   9% /boot
/dev/sda7             27595143   4269761  23325382  16% /home
/dev/sda5              3945128   2074820   1870308  53% /usr
/dev/sda6              1976492    270452   1706040  14% /var
cmd>
```

Uruchamianie programu dla niecierpliwych – funkcja `system()`

```
#include <stdlib.h>
int system(const char * cmdstring);
```

- Działanie funkcji `system()` jest ściśle związane z shellem. Funkcja tworzy proces potomny, który w celu wykonania polecenia (parametr `string`) uruchamia shella, na przykład `/bin/sh`.
- Przykład:

```
#include <stdlib.h>

int main ()
{
  int wynik;
  wynik = system ("ls -l");
  return wynik;
}
```

Funkcja `system()` zwraca:

`-1` – jeśli polecenie nie mogło być wykonane
`127` – jeśli nie może uruchomić shella
lub kod zakończenia polecenia przesłanego do wykonania.

- Przykład:

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

int main() {
  struct passwd *pw=0;
  char cmd[256];
  int wynik;
  if ( !(pw=getpwuid(getuid()))) {
    fprintf(stderr, "%s: nieznany UID\n", strerror(errno));
    exit(1);
  }
  sprintf(cmd,"ps -l | mail -s 'Uruchomione procesy %ld' %s",
         (long) getpid(), pw->pw_name);
  errno=0;
  wynik=system(cmd);

  if (wynik == 127 && errno !=0) {
    fprintf(stderr, "%s: nie wykonalo sie polecenie %s\n",
            strerror(errno),cmd);
  }
  else if (wynik == -1) {
    fprintf(stderr, "%s: nie wykonalo sie polecenie %s\n",
            strerror(errno),cmd);
  }
  else {
    printf("Polecenie '%s'\n zwrocilo kod %d\n", cmd,wynik);
    printf("Przeczytaj poczte.\n");
  }
exit(0);
}
```

Sesja i grupa procesów

Grupa procesów

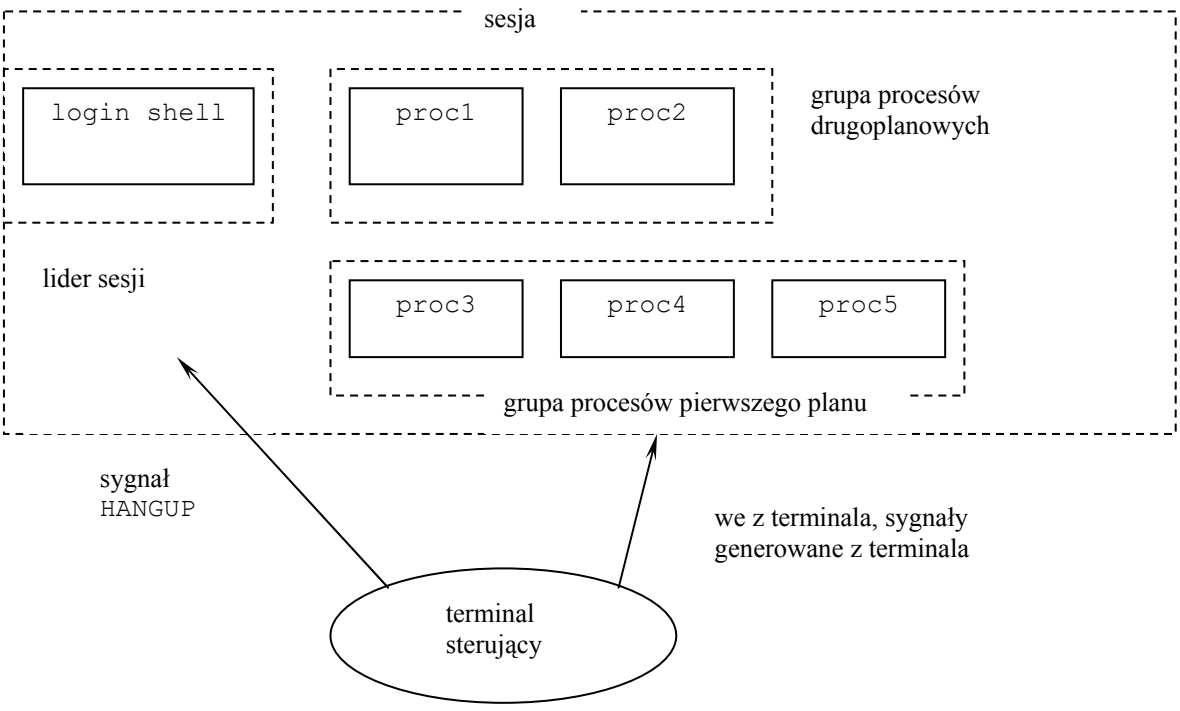
- *Grupa procesów*: zbiór powiązanych ze sobą procesów mających wspólnego przodka. Każda grupa procesów ma swój identyfikator.
- Każdy proces jest członkiem jakiejś grupy procesów. Proces, którego identyfikator PID jest taki sam jak identyfikator grupy nazywany jest *przywódcą grupy procesów* (ang. *process group leader*).
- Grupa procesów istnieje dopóty, dopóki jest w niej co najmniej jeden proces, niezależnie od tego, czy przywódca grupy się już zakończył czy nie.
- Proces dziedziczy grupę ze swojego procesu macierzystego. Proces może zmienić identyfikator grupy, do której należy.
- Zastosowanie: Sygnał można wysyłać jednocześnie do wszystkich procesów grupy.

Sesja, terminal sterujący

- *Sesja*: zbiór zawierający jedną lub więcej grup procesów, które mogą być powiązane z terminalem. Proces, który utworzył sesję nazywany jest *przywódcą sesji* (ang. *session leader*).
- Zastosowanie: obsługa procesów przez terminal sterujący.
- Grupy sesji tworzą:
  - jedną grupę pierwszego planu (ang. *foreground*) – można czytać z i pisać do terminala, do tej grupy przesyłane są sygnały generowane z klawiatury.
  - jedną lub więcej grup drugoplanowych (ang. *background*) –nie można czytać z i pisać do terminala, próba takiego działania kończy się przesłaniem do nich sygnału, który je zatrzymuje.
- Przykład:

  W shellu zostały wydane polecenia:

```
$ proc1 | proc2 &
$ proc3 | proc4 | proc5
```



- Synonimem dla terminala sterującego sesji jest plik `/dev/tty`.

Identyfikacja grupy procesów i sesji

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

zwrócenie identyfikatora grupy procesów	`pid_t getpgrp(void)`
zmiana identyfikatora grupy procesów procesu o identyfikatorze *pid* (0 oznacza dla bieżącego procesu)	`int setpgid(pid_t pid, pid_t pgid)`

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

Utworzenie nowej sesji: funkcja zwraca identyfikator sesji wywołującego procesu.	`pid_t setsid(void);`
Identyfikator sesji procesu *pid* (0 oznacza bieżącego procesu)	`pid_t getsid(pid_t pid)`
Uwaga: funkcja nie należy do standardu POSIX	

- Proces tworzący nową sesję *nie może być liderem grupy*. Gdy tworzona jest nowa sesja:
  - proces wywołujący staje się liderem nowej sesji, jest to jedyny proces w tej sesji
  - proces wywołujący staje się liderem nowej grupy procesów, identyfikator tej grupy jest równy identyfikatorowi PID procesu wywołującego
  - proces wywołujący pozbawiany jest terminala sterującego.

Obsługa sygnałów

- Do informowania jądra systemu, w jaki sposób ma obsługiwać dany sygnał służą funkcje:
  `signal()` – ANSI C
  `sigaction()` – POSIX

Zawodna obsługa sygnałów (ang. unreliable signals)

- Klasyczny schemat programu z funkcją `signal()`:

```
/* procedura obslugi sygnalu SIGUSR1 */
void obslugaUSR1(){
  /* przetwarzaj sygnał */
    ...
  /* ponowne zainstaluj funkcję obsługi */
  signal(SIGUSR1, obslugaUSR1);
}
main() {
  /* zainstalowanie obslugi sygnalu SIGINT */
  signal(SIGUSR1, obslugaUSR1);
    ...
}
```

- *Obecnie działanie funkcji `signal` jest zależne od wersji systemu!*

Niezawodna obsługa sygnałów (ang. reliable signals)

- Funkcje niezawodnej obsługi sygnałów muszą się charakteryzować:
  - stałą (ang. *persistent*) obsługą sygnałów;
  - blokowaniem tego samego sygnału podczas jego obsługi;
  - jednokrotnym dostarczeniem sygnału do procesu po odblokowaniu
  - możliwością maskowania (blokowania) sygnałów
- Funkcją, która ma zapewniać niezawodną obsługę sygnałów jest `sigaction`.

Przerwane funkcje systemowe

- Błąd `EINTR` oznacza, że wskutek dostarczenia sygnału przerwana została funkcja systemowa.
- Przerwana funkcja systemowa może:
  - być automatycznie wznawiana - zależy to od systemu
  - zwracać kod błędu `EINTR` - wtedy użytkownik chcąc ją wznowić musi zapewnić obsługę tego błędu
- Przykład:

```
if (read(fd,buf,size) < 0)
  if (errno==EINTR) {
    // na przykład continue
    ...
  }
  ...
}
```

Funkcja `signal`

```
#include <signal.h>
void (*signal (int sig, void (*func)(int)))(int);

typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

gdzie:
`sig` – określa numer sygnału, dla którego definiowana jest obsługa.
`handler` - nazwa funkcji obsługi sygnału zdefiniowanej przez użytkownika. Może on również przyjmować jedną z wartości:

- `SIG_IGN` - oznacza, że sygnał będzie ignorowany
- `SIG_DFL` - sygnał będzie obsługiwany w sposób domyślny, zdefiniowany w systemie.

Funkcja zwraca `SIG_ERR` jeśli wystąpił błąd, lub adres poprzedniej funkcji obsługi sygnału.

- Przykład 1:

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

/* procedura obslugi sygnalu SIGINT */
void obslugaINT(int signum){
  printf("Obsluga sygnalu SIGINT\n";
}
main() {
  /* zarejestrowanie obslugi sygnalu SIGINT */
  signal(SIGINT, obslugaINT)
  /* nieskonczona petla */
  while(1)
    sleep(10);
    ;
}
```

- Przykład 2:

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

main() {
  /* zarejestrowanie obslugi sygnalu SIGINT */
  signal(SIGINT, SIG_IGN)
  /* nieskonczona petla */
  while(1)
    sleep(10);
    ;
}
```

Funkcja `sigaction`

```
#include <signal.h>
int sigaction (int sig, const struct sigaction *action,
               struct sigaction *oldAction);
```

Argument *sig* określa numer sygnału, dla którego definiowana jest obsługa.
Argument *action* określa nowy sposób obsługi sygnału. Jeśli ma wartość NULL, to obsługa sygnału nie będzie zmieniona.
Jeśli argument *oldAction* jest różny od NULL, to po wykonaniu funkcji będzie wskazywał obsługę sygnału sprzed wywołania funkcji.
Funkcja zwraca 0, jeśli pomyślnie się wykona, w przeciwnym wypadku -1.

- Struktura `sigaction` określa sposób obsługi sygnału przez jądro.

```
/* tradycyjna struktura sigaction */
struct sigaction {
 void (*sa_handler)(int); /* Funkcja obsługi sygnału */
 sigset_t sa_mask;        /* Maska sygnałów – czyli sygnały blokowane
                             podczas obsługi bieżącego sygnału,
                          sygnał przetwarzany jest blokowany domyślnie */
 int sa_flags;            /* Nadzoruje obsługę sygnału przez jądro */
}

/* nowa struktura sigaction */
struct sigaction {
/* należy użyć jednego z dwóch poniższych */
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void);
}
```

- Podobnie jak w przypadku funkcji `signal`, składowej `sa_handler` zamiast adresu funkcji obsługi sygnału można przypisać jedną z wartości:
  - `SIG_IGN` - wtedy, kiedy sygnał ma być ignorowany
  - `SIG_DFL` - wtedy, kiedy sygnał ma być obsługiwany w sposób domyślny.
- Sygnał obsługiwany jest domyślnie blokowany, niezależnie od tego co zawiera składowa `sa_mask`.
- Składowa `sa_flags` pozwala użytkownikowi uszczegółowić obsługę sygnałów:

Wartość	Opis
`SA_RESETHAND`	Przywróć domyślną obsługę sygnału po jego obsłużeniu
`SA_NODEFER`	Wyłącz automatyczne blokowanie sygnału, gdy jest on obsługiwany
`SA_RESTART`	Automatycznie restartuj funkcję systemową
`SA_SIGINFO`	Użyj `sa_sigaction` dla funkcji obsługi sygnału. Można wtedy uzyskać dodatkowe informacje o sygnale.

Zbiór sygnałów

```
#include <signal.h>
sigset_t set;
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

- Zmienna typu `sigset_t` służy do przechowywania zbiór sygnałów.
- Funkcja `sigemptyset` inicjalizuje pusty zbiór sygnałów.
- Funkcja `sigfillset` inicjalizuje pełny zbiór sygnałów.
- Funkcja `sigaddset` dodaje sygnał `signum` do zbioru.
- Funkcja `sigdelset` usuwa sygnał `signum` z zbioru.
- Wszystkie te funkcje w razie powodzenia zwracają 0, a w razie niepowodzenia zwracają –1.
- Funkcja `sigismember` sprawdza, czy w zestawie znajduje się sygnał `signum`. Jeśli tak, zwraca 1, w przeciwnym wypadku zwraca 0. Jeśli w systemie nie ma sygnału o podanym numerze, funkcja zwraca –1.

Przykłady:

```
/* utworzenie zbioru dwóch sygnałów SIGINT i SIGQUIT */
sigset_t twosigs;

sigemptyset(&twosigs);
sigaddset(&twosigs, SIGINT);
sigaddset(&twosigs, SIGQUIT);
```

- Przykłady

A. Wszystkie sygnały są blokowane podczas obsługi sygnału `INT`

```
#include <stdio.h>              /* dla printf() */
#include <signal.h>             /* dla sigaction() */
#include <unistd.h>             /* dla pause() */

void Zakoncz(char *komunikat);
void ObslugaSygalow(int typSygnalu);

int main(int argc, char *argv[]){
  struct sigaction sygnaly;
  sygnaly.sa_handler = ObslugaSygalow;
  /* Maska blokująca wszystkie sygnały */
  if (sigfillset(&sygnaly.sa_mask) < 0)
    Zakoncz("sigfillset()");
  /* Brak opcji */
  sygnaly.sa_flags = 0;
  /* Ustaw obsługę sygnału przerwania INT */
  if (sigaction(SIGINT, &sygnaly, 0) < 0)
     Zakoncz("sigaction()");
  for(;;)
    pause(); /*zawies program do otrzymania sygnału*/
  exit(0);
}

void ObslugaSygalow (int typSygnalu) {
  printf("Otrzymano sygnal przerwania. Koniec programu.\n");
  exit(1);
}
```

B. Obsługa sygnału `SIGUSR1` zarezerwowanego na użytek aplikacji.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

sig_atomic_t licznikUSR1 = 0; // zmienna globalna używana w funkcji
                     // obsługi sygnału powinna być specjalnego typu

void ObslugaSygnalu (int numerSygnalu){
  ++licznikUSR1;
}

int main () {
  struct sigaction sygnaly;
  memset (&sygnaly, 0, sizeof (sygnaly));
  sygnaly.sa_handler = &ObslugaSygnalu;
  sigaction (SIGUSR1, &sygnaly, NULL);

  /* ...  */

  printf ("SIGUSR1 został wysłany %d razy\n", (int)licznikUSR1);
  return 0;
}
```

C. Czekanie na zakończenie procesu potomnego

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

sig_atomic_t status_potomka;

void CzyszczeniePoPotomku (int signal_number){
  int status;
  wait (&status);
  status_potomka = status;
}

int main () {
  struct sigaction obslugaSigchld;
  int i;
  memset (&obslugaSigchld, 0, sizeof (obslugaSigchld));
  obslugaSigchld.sa_handler = &CzyszczeniePoPotomkach;
  sigaction (SIGCHLD, &obslugaSigchld, NULL);

  /* ...  */

  return 0;
}
```

D. Czekanie na zakończenie procesu potomnego

```
void CzyszczeniePoPotomku (int signal_number){
/* wait nieblokujące */
   while (waitpid(-1,NULL,WNOHANG)>0)
   ;
}
```

Blokowanie sygnałów na poziomie procesu

- Proces może zablokować dostarczenie sygnału.

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- Funkcja `sigprocmask` zmienia zestaw aktualnie blokowanych sygnałów. Argument *how* określa sposób zmiany:

Wartość argumentu *how*	Nowy zestaw blokowanych sygnałów
SIG_BLOCK	Dodaj zbiór z argumentu *set* do zestawu aktualnie blokowanych sygnałów
SIG_UNBLOCK	Usuń zbiór z argumentu *set* z zestawu aktualnie blokowanych sygnałów
SIG_SETMASK	Ustaw zgodnie z zestawem podanym jako argument *set*

- Przykład:

```
#include <signal.h>
sigset_t newmask; /* sygnały do blokowania */
sigset_t oldmask; /* aktualna maska sygnałów */
sigemptyset(&newmask);         /* wyczyść zbiór blokowanych sygnałów */
sigaddset(&newmask, SIGINT);   /* dodaj SIGINT do zbioru */
/* Dodaj do zbioru sygnałów zablokowanych */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
   perror("Nie udało się zablokować sygnału");
/* tutaj chroniony kod */
if (sigprocmask(SIG_SETMASK, &newmask, NULL) < 0)
   perror("Nie udało się przywrócić maski sygnałów");
```

- Przykład:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
sigset_t oldmask, blockmask;
pid_t mychild;
sigfillset(&blockmask);
if (sigprocmask(SIG_SETMASK, &blockmask, &oldmask) == -1) {
   perror("Nie udało się zablokować wszystkich sygnałów");
   exit(1);
}
if ((mychild = fork()) == -1) {
   perror("Nie powołano procesu potomnego");
   exit(1);
} else if (mychild == 0) {
   if (sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1){
      perror("Proces potomny nie odtworzył maski sygnałów");
      exit(1);
   }
   /* .....kod procesu potomnego ..... */

} else {
   if (sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1){
      perror("Proces macierzysty nie odtworzył maski sygnałów ");
      exit(1);
   }
   /* ..... kod procesu macierzystego..... */   }
```

Czekanie na sygnały

Funkcja pause()

- Funkcja `pause` zawiesza wywołujący ją proces do czasu dostarczenia sygnału. Z funkcji wraca się po obsłużeniu sygnału.

```
#include <unistd.h>
int pause(void);
```

Funkcja zawsze zwraca wartość –1 i ustawia zmienną `errno` na `EINTR`.

Funkcja sleep()

- Funkcja `sleep` usypia wywołujący ją proces na określoną w argumencie liczbę sekund. Z funkcji wraca się po upływie określonej liczby sekund lub po otrzymaniu sygnału, który nie jest ignorowany.

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

Funkcja zwraca wartość 0, jeśli upłynął czas lub liczbę sekund, które jeszcze zostały w przypadku przerwania spowodowanego sygnałem.

Funkcja nanosleep()

- Funkcja `nanosleep` usypia wywołujący ją proces na czas określony w argumencie *`req`*. Z funkcji wraca się po upływie określonego czasu lub po otrzymaniu sygnału, który nie jest ignorowany przez proces.

```
#include <time.h>
int nanosleep(const struct timespec *req, struct timespec *rem);

struct timespec {
  time_t tv_sec;        /* seconds */
  long tv_nsec;         /* nanoseconds */
};
```

Funkcja zwraca -1, jeśli nastąpiło przerwanie w wyniku dostarczonego sygnału. W tym przypadku czas pozostały umieszczany jest w argumencie *`rem`*.

Zadanie

Chcemy czekać do momentu otrzymania określonego sygnału. Dlaczego poniższe programy nie będą poprawne?

```
// wersja A
// zmienna globalna ustawiana w funkcji obsługi sygnału
static volatile sig_atomic_t sygnalOtrz=0;
...
while(sygnalOtrz==0)
  pause();

================================================================
// wersja B
static volatile sig_atomic_t sygnalOtrz=0
...
int signum=SIGUSR1; // na ten sygnał czekamy
sigset_t zbiorSyg;
...
sigemptyset(&zbiorSyg);
sigaddset(&zbiorSyg,signum);
siprocmask(SIG_BLOCK,&zbiorSyg,NULL);
while(sygnalOtrz==0)
  pause();
```

Funkcja `sigpending()`

- Sygnały, które pojawiły się w czasie blokady są sygnałami oczekującymi (ang. *pending signals*).

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Funkcja `sigpending` zwraca w zmiennej *`set`* zestaw sygnałów oczekujących.

Funkcja `sigsuspend()`

- Aby odebrać sygnał oczekujący, możemy posłużyć się funkcją `sigsuspend`.

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

Funkcja `sigsuspend` *tymczasowo* zastępuje maskę sygnałów procesu maską podaną w argumencie funkcji. Jeśli nowa maska pozwoli na obsługę sygnału już oczekującego, zostanie on obsłużony natychmiast. W przeciwnym wypadku proces zostaje zawieszony do momentu, kiedy nadejdzie odblokowany sygnał. Po obsłudze sygnału ponownie jest ustawiana maska sprzed wywołania `sigsuspend`.

Zadanie

Napisz poprawną wersję programu, który czeka do momentu otrzymania określonego sygnału. Wykorzystaj funkcję `sigprocmask` i `sigsuspend`.

Wysyłanie sygnałów

Funkcja `kill()`

- Wysyłanie sygnału do działającego procesu realizowane jest przy pomocy funkcji systemowej `kill`:

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

Argument `pid` określa identyfikator procesu-odbiorcy sygnału, natomiast `sig` jest numerem wysyłanego sygnału. Jeśli:

`pid > 0`	sygnał jest wysyłany do procesu o identyfikatorze `pid`
`pid = 0`	sygnał jest wysyłany do wszystkich procesów w grupie procesu wysyłającego sygnał
`pid = -1`	sygnał jest wysyłany do wszystkich procesów w systemie, z wyjątkiem procesów specjalnych, na przykład procesu `init`; nadal obowiązują ograniczenia związane z prawami
`pid < -1`	sygnał jest wysyłany do wszystkich procesów we wskazanej grupie `-pid`

- Funkcja `kill` zwraca 0, jeśli sygnał został pomyślnie wysłany, w przeciwnym wypadku zwraca -1 i ustawia kod błędu w zmiennej `errno`.

Funkcja `alarm()`

- Funkcja `alarm` pozwala ustawić czas pobudki.

```
#include <unistd.h>
unsigned int alarm(unisgned int sec);
```

Funkcja ustawia licznik czasu, jeśli *sec* > 0. Po upływie tego czasu do procesu jest wysyłany sygnał `SIGALRM`.
Wartość 0 sekund kasuje istniejący licznik.

Funkcja `raise()`

- Wysyłanie dowolnego sygnału do samego siebie jest realizowane za pomocą funkcji `raise`.

```
#include <signal.h>
int raise (int sig);
```

Funkcja `abort()`

- Wywołanie funkcji `abort()` powoduje wysłanie do siebie sygnału `SIGABRT`.

```
#include <stdlib.h>
void abort(void);
```

Uwagi dodatkowe

- Zasady, których powinno się przestrzegać:
  - jeśli w procedurze obsługi sygnału przypisywana jest wartość zmiennej globalnej, to zmienna ta powinna być typu `sig_atomic_t` (jest to typ całkowity, gwarantujemy, że przypisanie będzie dokonane przez jedną instrukcję maszynową)
  - można zadeklarować zmienną globalną jako `volatile` - jest to informacja dla kompilatora, że nie należy tej zmiennej optymalizować, jest zmieniana poza zwykłą ścieżką wykonania programu

Funkcje wielowejściowe

- W funkcjach obsługi sygnałów można wywoływać tylko funkcje wielowejściowe (ang. *reentrant*, wielobieżne). Kod wielowejściowy to taki kod, w którym nie jest przechowywana informacja o stanie, ani lokalnie, ani globalnie. Wszystko na czym funkcja operuje jest dostarczane przez użytkownika.
  Przykłady funkcji wielowejściowych: `_exit`, `alarm`, `chmod`, `kill`, `read`, `write`
  Przykład funkcji nie wielowejściowej: `malloc`.
- W przypadku konieczności użycia funkcji nie wielowejściowej, musimy zabezpieczyć pełne jej wykonanie, na przykład za pomocą zablokowania sygnałów (`sipgprocmask`), okresowego sprawdzania, czy sygnał nie oczekuje (`sigpending`), wywołania w bezpiecznym miejscu programu `sigsuspend` w celu obsługi sygnału.
- Wiele z funkcji określanych mianem wielowejściowych zmienia wartość zmiennej `errno` – czyli nie jest w pełni wielowejściowa. Aby po powrocie z funkcji obsługi sygnału mieć niezmienioną wartość `errno`, trzeba w funkcji obsługi sygnału odtworzyć wartość `errno` poprzedzającą wywołanie tej procedury.

Dziedziczenie i syganały

- Po wykonaniu funkcji `fork` proces potomny dziedziczy po swoim przodku wartości maski sygnałów i ustalenia dotyczące obsługi sygnałów.
- Nie obsłużone sygnały procesu macierzystego są czyszczone.

- Po wykonaniu funkcji `exec` maska obsługi sygnałów i nieobsłużone sygnały są takie same jak w procesie, w którym wywołano funkcję `exec`.