Programowanie Współbieżne

W Linuxie/Unixie

Identyfikatory

- ***pid*** – numer identyfikacyjny procesu
  - zwykle od 0 do 32K
  - przydzielany przez system każdemu nowemu procesowi
  - uzyskać możemy go przez *int getpid()*
  - 0 numer specjalnego procesu jądra do wymiany procesów **planista** (*scheduler*)
  - **1** numer specjalnego procesu jądra inicjującego *init*
  - 2 w implementacjach Unixa przeznaczonych do pracy z pamięcią wirtualną, **demon stron**.

Identyfikatory

- ***ppid*** – numer identyfikacyjny procesu macierzystego
  - każdy proces ma swój numer macierzysty (oprócz inita)
  - można go uzyskać przez *int getppid()*
  - Przykład:

```
main()
 {
 printf(„pid = %d, ppid =
  %d\n”,getpid(),getppid());
 exit(0);
 }
```

Identyfikatory

- **identyfikator** grupy procesów
  - każdy proces jest członkiem grupy procesów
  - grupa identyfikuje się dodatnią liczbą całkowitą
  - wiele procesów może mieć ten sam numer
  - przywódca grupy procesów ma taki sam id jak grupa
  - wartość sprawdzamy przez: *int getpgrp(int pid);*
  - jeżeli pid = 0 to id grupy procesów jest dla bieżącego procesu
  - jeżeli > 0 to dla podanego
  - można ustawić id grupy: *int setpgrp(int pid, int pgrp);*
  - trzeba mieć oczywiście prawa do takiego procesu.

Identyfikatory

- **identyfikator grupy terminali oraz terminal sterujący**
  - dodatnia liczba całkowita
  - każdy proces może być członkiem grupy terminali
  - przywódca grupy procesów ma id = identyfikatorowi grupy terminali
  - jest to proces który otworzył terminal
  - terminal ma tylko jeden proces sterujący
  - jest on procesem sterującym
  - identyfikator grupy terminali identyfikuje terminal sterujący

Identyfikatory

- terminal sterujący wysyła sygnały, które są odbierane po naciśnięciu określonych klawiszy na terminalu i zakończeniu działania shella zgłoszonego.
- do terminala sterującego można się odnosić automatycznie za pośrednictwem urządzenia /dev/tty*
- aby zbadać lub ustanowić identyfikator grupy terminali dla swojego terminala sterującego użyć można funkcji `ioctl` z odpowiednią opcją (*TIOCGPGRP* lub *TIOCSPGRP*)
- można się odłączyć przez funkcję *setpgrp*

Identyfikatory

- **rzeczywisty identyfikator użytkownika**
  - dodatnia liczba całkowita przyporządkowana każdemu użytkownikowi
  - pobrać można za pomocą

```
unsigned short getuid();
```

  - odwzorowanie uidów i nazw użytkowników znajduje się w pliku /etc/passwd

Identyfikatory

- **rzeczywisty identyfikator grupy**
  - dodatnia liczba całkowita do której przyporządkowani są wszyscy użytkownicy
  - można ją odczytać przez

```
unsigned short getgid();
```

  - odwzorowanie *gidów* i nazw grup znajduje się w pliku **`/etc/group`**

Identyfikatory

- ***obowiązujący identyfikator użytkownika***
  - *każdemu procesowi jest też przyporządkowany obowiązujący identyfikator użytkownika*
  - *można go uzyskać przez* ***unsigned short geteuid();***
  - *zazwyczaj taka sama wartość jak rzeczywisty ident.*
  - *można określić specjalny 1 bitowy znacznik*
    - *jeżeli jest = 1 to podczas wykonywania tego programu obowiązującym identyfikatorem użytkownika procesu staje się identyfikator użytkownika przypisany właścicielowi pliku programu.*
    - *np. pliki root'a ustawione z e-S-ką wykonują się z prawami roota.*

Prawa dostępu

- *każdy proces ma przyporządkowane cztery rodzaje numerów identyfikacyjnych*
  - *rzeczywisty identyfikator użytkownika;*
  - *rzeczywisty id gr*
  - *obowiązujący id użytkownika*
  - *ob. id grupy*
- *Ponadto w systemie linux posługujemy się osobno uprawnieniami dla użytkownika, grupy i innych*
- *RWXRW-R-- 764*

Procesy

- ***fork*** *– jedynym sposobem utworzenia nowego procesu w Unixie jest wywołanie funkcji* ***int fork();*** *(nie dotyczy to procesu init).*
- *Wywołuje się ją w dwóch przypadkach:*
- *Gdy proces chce stworzyć swoją kopie tak aby jedna z nich mogła wykonać jakieś inne zadania*
- *Gdy proces chce wykonać drugi program wtedy kopia wykonuje exec, tak zwykle postępują programy shell.*

Procesy

- *tworzy kopię procesu wywołującego*
- *proces wywołujący **fork** nazywamy macierzystym lub przodkiem*
- *proces powstały w ten sposób nazywamy procesem potomnym lub potomkiem.*
- *Jedno wywołanie funkcji **fork** zwraca dwa razy wartość*
- *pid nowego procesu do przodka*
    - *0 do nowego procesu*
    - *– 1 gdy błąd*

Procesy

```
main ()
{
int childpid;
if ((childpid = fork()) == -1)
    {
    perror(„can't fork”);
    exit(1);
    }
else
    if (childpid == 0) /* proces potomny */
        {
        printf(„child: child pid = %d, parent pid = %d\n”,getpid(),
getppid());
        exit(0);
        }
    else   /* proces macierzysty */
        {
        printf(„parent: child pid = %d, parent pid = %d\n”,childpid,
getpid());
        exit(0);
        }
}
```

Procesy

Proces potomny kopiuje z procesu macierzystego następujące wartości:

- rzeczywisty identyfikator użytkownika,
- rzeczywisty identyfikator grupy,
- obowiązujący identyfikator użytkownika,
- obowiązujący identyfikator grupy,
- identyfikator grupy procesów,
- identyfikator grupy terminali,
- katalog główny,
- roboczy katalog bieżący,
- ustalenia dotyczące obsługi sygnałów,
- maska trybu dostępu do pliku.

Procesy

Proces potomny różni się od procesu macierzystego tym, że:

- ma nowy jednoznaczny identyfikator procesu;
- ma inny identyfikator procesu macierzystego;
- proces potomny ma własne kopie deskryptorów plików procesu macierzystego;
- czas, który pozostaje do pojawienia się sygnału budzika jest zerowany dla procesu potomnego.

exit

- ***exit*** – służy do zakończenia procesu.
- po jej wykonaniu nigdy nie następuje powrót do procesu który ją wywołał.
- proces wywołujący *exit* przekazuje do jądra liczbę całkowitą oznaczającą stan końcowy procesu.
- proces macierzysty może ją odczytać za pomocą funkcji ***wait***
- Trzeba odróżnić funkcje systemową exit od standardowej z C gdzie opróżniane są bufory We/Wy i dopiero wołana jest exit systemowa.
- Jeżeli chcemy wywołać od razu exit należy zastosować _exit

exit

- gdy proces macierzysty wywołał wait to zostaje on zawiadomiony o końcu potomka
- Po wywołaniu w potomku następuje powrót z funkcji wait u rodzica.
- jeżeli proces macierzysty nie zawołał wait to proces który ma być zakończony staje się procesem-duchem (zombie), zasoby są zwalniane przez jądro ale stan musi być utrzymany aż rodzić się o niego nie upomni.
- jeżeli jest sierotą to id macierzystego staje się 1
- jeżeli wszystkie pola identyfikatorów – procesu, grupy procesów i grupy terminali, zakończonego procesu są identyczne, to jądro wysyła sygnał zawieszenia, SIGHUP, do każdego procesu, który ma identyfikator grupy procesów taki sam jak kończony proces.

exec

- jedyny sposób by Unix wykonał jakiś program jest wywołanie exec*
- zastępuje ona program bieżącego procesu nowym programem.
- proces, który wywołał funkcję systemową exec, nazywamy procesem wywołującym
- program, który ma być wykonany nazywamy nowym programem. (nie procesem)
- funkcja exec tylko wtedy może powrócić do programu, z którego była wywołana, gdy okaże się, że wystąpił błąd.

exec

Istnieje 6 różnych wersji exec'a

- ```
  int execlp(char *filename, char *arg0, char *arg1, ...,
          char *argn, (char *) 0);
  ```

- ```
  int execl(char *pathname, char *arg0, char *arg1, ...,
          char *argn, (char *) 0);
  ```

- ```
  int execle(char *pathname, char *arg0, char *arg1, ...,
          char *argn, (char *) 0, char **envp);
  ```

- ```
  int execvp(char *filename, char **argv);
  ```
- ```
  int execv(char *pathname, char **argv);
  ```
- ```
  int execve(char *pathname, char **argv, char **envp));
  ```

exec

- część z tych funkcji ma argumenty przekazywane przez listę argumentów a cześć przez wskaźnik do tablicy stringów z argumentami
- część z nich ma *pathname* a niektóre *filename*, które jest przekształcane w *pathname* przez zmienną środowiskową *PATH*
- część z nich ma *envp* a część nie. Gdy nie ma używa się domyślnej wartości zmiennej zewnętrznej *environ*.

exec

Nowy program odziedziczy następujące cechy

- id procesu
- id procesu macierzystego,
- id grupy procesów,
- id grupy terminali,
- czas, który pozostaje do pojawienia się sygnału budzika,
- katalog główny
- roboczy katalog bieżący,
- maska trybu dostępu do pliku,
- rzeczywisty id użytkownika
- rzeczywisty id grupy,
- pliki zajęte.

exec

- Nowy proces może różnić się w:
  - obowiązującym id użytkownika
  - obowiązującym id grupy
- Jeżeli bit użytkownika dla programu, który ma być wykonany za pomocą exec ustawiony jest na 1, to obowiązującym id użytkownika staje się id użytkownika należący do właściciela pliku programu.
- podobnie z grupą
- Sygnały
  - te co były ignorowane to będą
  - te co kończyły proces to też będą tak samo reagować
  - A te co były obsłużone to już nie będą ponieważ adres funkcji obsługi sygnału jest już inny.

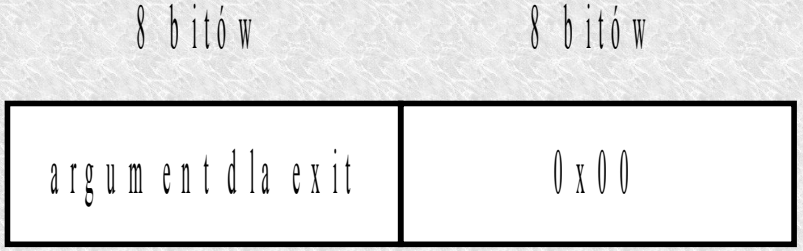
wait

powoduje że proces czeka aż jeden z jego potomków zakończył działanie

int wait(int *status);

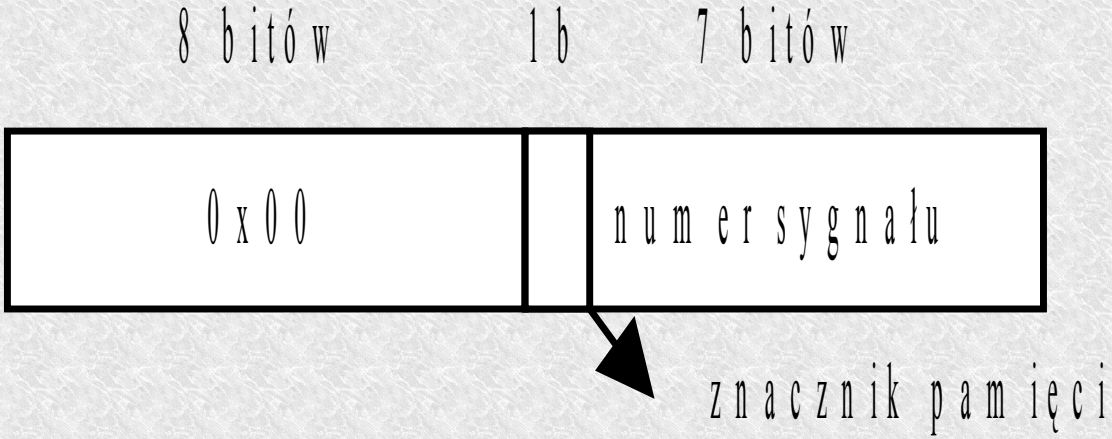
wait zwraca id procesu potomnego który został zakończony w wyniku:

1 potomek wywołał *exit* zmienna status ma



wait

2 potomek został zakończony w wyniku nadejścia sygnału
zmienna status ma:



wait

3 został zakończony podczas wykonywania go w trybie śledzenia. zmienna status ma:

wait

- jeżeli proces kończąc się wygenerował *core* to bit znacznika pamięci jest 1
- jeżeli proces nie ma żadnego potomka to funkcja od razu zwraca –1
- jeżeli proces ma potomki to zostaje on zawieszony do momentu zakończenia jednego z nich
- jeżeli status nie jest *NULL* to otrzyma on wartość zwracaną przez potomka przez funkcje *exit.*
- gdy proces potomny kończy działanie to rodzic dostaje sygnał *SIGCLD*, można napisać funkcje obsługi tego sygnału gdzie wywołamy funkcje wait
- można też zignorować sygnał *SIGCLD* (u rodzica oczywiście)

    signal(SIGCLD, SIG_IGN);

i w ten sposób jądro będzie poinformowane że rodzica nie obchodzi stan ich dzieci więc procesy *zombie* od razu będą usuwane.

wait3

różni się tym od *wait* że ma możliwość nie czekania na zakończenie procesu potomnego

```
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
int wait3(union wait *status, int options, struct
rusage *rusage);
```

- jeżeli options = WNOHANG to wait3 nie czeka na koniec potomka a zwraca 0.
- rusage informuje proces macierzysty ile o używaniu procesora przez potomka

Sygnały

- Sygnał jest informacją dla procesu że wystąpiło jakieś zdarzenie. Nazywane są też przerwaniami programowymi.
- Są wysyłane asynchronicznie
- Każdy sygnał ma nazwę opisaną w ***<signal.h>***
- Można wysyłać z jednego procesu do drugiego lub z jądra do procesu.
- Do wysyłania sygnałów służy funkcja systemowa ***kill***

Sygnały

int kill (int idproc, int sig);

- do procesu może wysłać sygnał tylko jego właściciel lub nadzorca.
- Jeżeli *pid* = 0 to sygnał wysyłany do wszystkich w grupie procesów nadawcy.
- Jeżeli *pid* = -1 a nadzorca nie jest *rootem* to sygnał będzie wysłany do wszystkich procesów o tym samym właścicielu co obowiązujący uid procesu wysyłającego.
- Jeżeli *pid* = - 1 i nadzorca to *root* to sygnał jest wysyłany do wszystkich procesów oprócz systemowych (zwykle 0 lub 1).
- Jeżeli *pid* < -1 to sygnał jest wysyłany do wszystkich procesów których id grupy jest = wartości bezwzględnej.
- Jeżeli *sig* = 0 to jest to sygnał testowy, np. by sprawdzić pid.
- Polecenie kill robi to samo tylko pobiera argumenty z wiersza poleceń.

Sygnały

- Pewne znaki z terminali powodują wysłanie sygnałów np.
  - *Ctrl*+C lub *Delete SIGINT*,
  - *Ctrl*+\ *SIGQUIT*.
  - *Ctrl+Z SIGTSTP*
- Znakiem przerwania i zakończenia można ustanowić niemal dowolny znak z terminala.
- Pewne sytuacje sprzętowe też generują sygnały
  - Przy błędzie obliczeń zmiennoprzecinkowych mamy SIGFPE
  - Odniesienie do przestrzeni adresowej spoza procesu SIGSEGV
- Pewne sytuacje wykrywane przez oprogramowanie systemowe, np. pojawienie się wysoko priorytetowych danych w gnieździe.

Sygnały

Co może z sygnałem zrobić proces?

- może dostarczyć funkcje, która będzie wywoływana za każdym razem gdy pojawi się specjalny rodzaj sygnału. (**procedura obsługi sygnału**)
- Może go zignorować, wszystkie oprócz SIGKILL I SIGSTOP
- Proces może pozwolić by wykonało się postępowanie domyślne.

Sygnały

Żeby określić jak sygnał ma być obsłużony proces wywołuje funkcje systemową:

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

- Oznacza to, że funkcja signal przekazuje wskaźnik do funkcji, która przekazuje liczbę całkowitą.
- func określa adres funkcji, która nie przekazuje niczego.
- Może mieć dwa stałe parametry
  - SIG_DFL oznacza że sygnał spowoduje czynności domyślne
  - SIG_IGN oznacza ignorowanie sygnału
- Signal przekazuje zawsze poprzednią wartość argumentu func dla danego sygnału.

Sygnały

Przykład: Chcemy by SIGUSR1 był ignorowany to piszemy

```
signal (SIGUSR1, SIG_IGN);
```

Chcemy by SIGINT wywoływał funkcje moje_przerwanie() to piszemy

```
#include <signal.h>
extern void moje_przerwanie();
...
signal(SIGINT, moje_przerwanie);
```

Gdy funkcja jest wywoływana do obsługi sygnału to jako pierwszy parametr trafia numer przerwania, można dzięki temu jedną funkcją obsługiwać wiele sygnałów.

Sygnały

- **SIGALRM** - Budzik- proces może ustawić budzik na określoną ilość sekund.
  Unsigned int alarm(unsigned int sec);Po czasie sec sekund zostanie przekazany przez jądro sygnał SIGALRM do procesu który wywołał alarm.
  unsigned int sleep(unsigned int sec); usypia proces na sec sekund. Domyślne działanie, zakończenie procesu.
- **SIGCLD** - Zakończenie procesu potomnego – Wysyłany jest do procesu macierzystego kiedy zakończy się potomek.
- **SIGHUP** - Zawieszenie – gdy terminal jest zamykany to do procesów dla których jest on terminalem sterującym wysyłany jest SIGHUP. Wysyłany też jest do procesów w grupie gdy proces przywódczy kończy pracę. Domyślnie zakończenie procesu.
- **SIGINT** - Znak przerwania – zwykle gdy użytkownik naciśnie klawisz przerwania na terminalu. Domyślnie, zakończenie
- **SIGKILL** – Bezwzględne zakończenie procesu, nie może być ignorowany ani przechwycony.

Sygnały

- **SIGPIPE** - Dane nie są odbierane z łącza komunikacyjnego – dostaje go proces piszący, gdy wysyła on dane do łącza lub kolejki gdzie nie ma kto odebrać. Domyślnie, zakończenie
- **SIGQUIT -** Znak zakończenia – gdy użytkownik naciśnie znak przerwania. Podobny jest do SIGINT ale tu generowany jest obraz pamięci.
- **SIGSEGV** - Naruszenie segmentacji. (Tzw Internal 11)
- **SIGSTOP** - bezwzględne zatrzymanie, nie można go zignorować ani przechwycić, można po tym proces reaktywować przez SIGCONT
- **SIGTERM** - Programowe zakończenie procesu – wysyłany przez inny proces do zabić procesu. Domyślnie zakończenie.
- **SIGUSR1** i **SIGUSR2** - Zdefiniowany przez użytkownika – może służyć do komunikacji między procesami, ale nie niesie za dużo informacji, oprócz tej że się pojawił.

Sygnały

```
#include <signal.h>

void obsluga2()
{
    printf("Program odebral sygnal ctrl+c\n");
}
void obsluga11()
{
    printf("Cos zajechalo pamiec\n");
    exit(-1);
}
void obsluga28()
{
    printf("Zmiana rozmiaru okna \n");
}
main()
{
char slowo[5];
signal(2,obsluga2);
signal(11,obsluga11);
signal(28,obsluga28);
strcpy(slowo,"ala ma kota jednak kot jej nie lubi");
printf("%s\n",slowo);
sleep(20);
printf("Normalny koniec\n");
}
```

Sygnały niezawodne

Wczesne implementacje sygnałów były zawodne, gdy było ich sporo dochodziło do tak zwanej sytuacji wyścigów i mogło dojść do zagubienia. W celu uzyskania sygnałów niezawodnych dodano następujące właściwości:

- Procedura obsługi sygnału zostaje nadal zainstalowana po wystąpieniu sygnału. Wcześniej zostawała ona usuwana i zanim proces wywołał ponownie funkcje signal to pewne sygnały mogły być gubione
- Proces powinien mieć możliwość wstrzymania nadejścia sygnału, nie chodzi tu o zignorowanie go tylko chwilowe wstrzymanie do momentu kiedy będzie on gotowy na obsłużenie go.
- Podczas gdy sygnał jest obsługiwany przez proces to drugi sygnał jest zapamiętywany i obsłużony wtedy gdy pierwszy sygnał zostanie obsłużony.

Maska sygnału

możemy określić maskę sygnałów za pomocą makroinstrukcji

```
#include <sygnal.h>
sigmask(int sig);
```

np. dla SIGQUIT i SIGINT maskę tworzymy tak:

```
int mask;
mask = sigmask(SIGQUIIT) | sigmask(SIGINT);
```

Implementacja maski to liczba 32 bitowa na każdy bit przypada jeden sygnał.

Blokada sygnału

Aby zablokować jeden lub więcej sygnałów, trzeba wywołać funkcje systemową

int sigblock(int mask);

Jako argument podajemy maskę mask, sygnały te będą dodane to zbioru sygnałów zablokowanych.
Wartość zwrotna to maska obowiązująca przed wywołaniem tej funkcji.

Blokada sygnału

Sygnał odblokowuje się używając:

```
int sigsetmask(int mask);
```

W argumencie *mask* nie występuje sygnał który chcemy odblokować.

Blokada sygnału

Przykład:

Jeżeli nie chcemy by pojawił się jakiś sygnał podczas wykonywania jakiegoś fragmentu programu to robimy to tak:

```
int oldmask;
oldmask = sigblock(sigmask(SIGQUIT) |
sigmask(SIGINT));
/* chroniony fragment programu */
sigsetmask(odlmask); /* przywróć starą maskę */
```

Blokada sygnału

W systemie V nie ma pojęcia maski sygnału ale można zablokować przez funkcję:

```
int sighold(int sig);
```

A do odblokowania służy

```
int sigrelse(int sig);
```