

Instrukcja do laboratorium Systemów Operacyjnych
Ćwiczenie 6

Temat: Zarządzanie procesami w systemie Linux

Wprowadzenie

I. Pojęcie procesu

Procesem nazywamy instancję programu w trakcie wykonywania. Każdy proces zajmuje własną przestrzeń adresową i wykonuje w niej określone w swoim kodzie instrukcje.

Proces składa się z:

- kodu programu (sekcja tekstu)
- licznika rozkazów
- zawartości rejestrów procesora
- stosu procesu (przechowuje dane tymczasowe)
- sekcji danych (przechowuje zmienne globalne)

Proces może znaleźć się w następujących stanach

- nowy – proces utworzony
- aktywny – trwa wykonywanie instrukcji
- gotowy – czeka na przydział procesora
- czekający – czeka na zakończenie jakiegoś zdarzenia
- zakończony – proces zakończył działanie

Atrybuty procesu

- Numery identyfikacyjne procesu
 - Każdy proces jest identyfikowany przez unikalny numer ID procesu (*process ID*, *PID*). PID to (najczęściej) 16-bitowy numer przydzielany przez system podczas tworzenia procesu. Każdy proces posiada również swój "parent process" (poza jednym - *init*). Dlatego można wyobrazić sobie procesy w systemie jako drzewo, w którym korzeniem jest właśnie *init*. ID takiego procesu to *parent process ID*, *PPID*
 - Kiedy odnosimy się do procesu w językach C/C++, używamy typu `pid_t`, zdefiniowanego w `<sys/types.h>`. W naszym programie możemy uzyskać PID procesu używając funkcji systemowej `getpid()`. Możemy również uzyskać PPID dzięki funkcji `getppid()`. Funkcje te możemy wywoływać po wcześniejszym zainkluadowaniu `<unistd.h>`. Poniższy program pokazuje użycie tych funkcji:

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

```
int main () {
    printf ("PID: %d\n", (int) getpid ());
    printf ("PPID: %d\n", (int) getppid ());
    return 0;
}
```

- Warto zwrócić uwagę, że przy kolejnych uruchomieniach tego programu PID się zmienia, a PPID zostaje takie samo (jeśli uruchamiamy proces z tego samego shella, który jest "parent process" dla uruchamianego w nim procesu).

- Identyfikatory użytkownika i grupy
 - Każdy proces jest powiązany z rzeczywistym identyfikatorem użytkownika i

identyfikatorem grupy, które posiadał wywołujący proces użytkownik. Efektywne UID i GID to identyfikatory właściciela pliku programu. Istnieją funkcje systemowe do pobierania wartości UID i GID:

- `getuid()` - zwraca rzeczywisty identyfikator użytkownika jako wartość typu `uid_t`
- `geteuid()` - zwraca efektywny identyfikator użytkownika jako wartość typu `uid_t`
- `getgid()` - zwraca rzeczywisty identyfikator grupy jako wartość typu `gid_t`
- `getegid()` - zwraca efektywny identyfikator grupy jako wartość typu `gid_t`

■ Bieżący katalog roboczy i katalog główny

■ Proces związany jest z bieżącym katalogiem roboczym. Uruchamiany proces jest umieszczany w tym samym katalogu, co jego proces macierzysty. Również podczas uruchamiania, proces macierzysty przekazuje nowemu procesowi informacje o katalogu głównym. Do operacji powyższymi katalogami mamy następujące funkcje:

- `int chdir(const char* path)` - zmienia katalog roboczy procesu
- `int chroot(const char* path)` - zmienia katalog główny procesu

I. Tworzenie procesów

■ `system()` - w bibliotece standardowej C istnieje funkcja `int system(const char* command)`, dzięki której możliwe jest wykonanie w systemie polecenia podanego jako argument wywołania. Funkcja zwraca kod zwrócony przez polecenie, lub -1 w razie błędu.

■ `Fork()` - Funkcja `fork()` powoduje utworzenie przez jądro procesu będącego kopią procesu bieżącego. Jest to proces potomny (*child process*), który dostaje nowe PID. Funkcja `fork()` zwraca procesowi macierzystemu PID dziecka, a dziecku 0. W wypadku błędu zwraca -1. Poniższy program przedstawia kopiowanie procesów za pomocą `fork()`:

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

int main () {
    pid_t child_pid;
    printf ("PID glownego programu: %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid != 0) {
        printf ("to jest proces macierzysty, a jego PID to:
        %d\n", (int) getpid ());
        printf ("PID procesu potomnego: %d\n", (int)
        child_pid);
    }
    else
        printf ("to jest proces potomny, a jego PID to:
        %d\n", (int) getpid ());
    return 0;
}
```

■ `fork()` nie gwarantuje, że proces potomny będzie żył krócej niż macierzysty. Jeśli dojdzie do sytuacji, że proces macierzysty zakończy się przed potomnym, ten drugi zostanie przygarnięty przez `init`.

■ Rodzina `exec()`

■ Funkcje z rodziny `exec()` służą do zmieniającego w procesie programu na inny

program. Po wywołaniu `exec()` kończy się wykonywanie danego programu i zaczyna działanie od początku nowy program w tym samym procesie. Oto rodzina funkcji `exec()`.

- `int execl(const char *path, const char *arg, ...);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execlx(const char *path, const char *arg, ..., char * const envp[]);`
- `int execv(const char *path, char *const argv[]);`
- `int execvp(const char *file, char *const argv[]);`
- gdzie:
- `path` - ścieżka dostępu do pliku
- `file` - nazwa pliku (szukana w określonych ścieżkach)
- `arg` - tablica znakowa (zakończona NULem) z argumentami wywołania
- `argv[]` - tablica tablic znakowych (zakończonych NULem) z argumentami wywołania
- `argv[]` - tablica tablic znakowych (zakończonych NULem) ze zmiennymi środowiskowymi w formacie `ZMIENNA=wartosc`

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

int main () {
    printf("Jestes w katalogu:\n");
    execl ("/bin/pwd", "pwd", (char *)0);
    perror ("Blad uruchamiania pwd");
    return 0;
}
```

• Konczenie procesow

■ `void _exit(int status)` - kończy program tak, jakby program doszedł do końca funkcji `main` albo napotkał `return`. przekazywany argument to stan zakończenia procesu. Zwykle zwraca się 0, kiedy nie wystąpił błąd i inną wartość w przeciwnym wypadku.

■ `void exit(int status)` - *działa podobnie jak `_exit`, z tą różnicą, że dodatkowo wykonuje działania zależne od biblioteki od jakiej pochodzi, może np.:*

- przekazywać procesy potomne do `init`
- zwalniać pamięć
- zamykać otwarte pliki

I. Synchronizacja procesów

■ `wait()` - funkcja ta zawiesza działanie wywołującego ją procesu, aż do czasu, kiedy jego potomek zakończy działanie. Po tym, kiedy którykolwiek proces potomny się zakończy, program wznowia działanie. `wait()` zwraca PID zakończonego potomka.

■ `waitpid()` - deklaracja funkcji `waitpid()` wygląda następująco: `pid_t waitpid(pid_t pid, int *status, int options);`, gdzie:

- `pid` - PID procesu potomnego, na który parent ma czekać
- `status` - wskaźnik do zmiennej, w której będzie zawarty stan procesu potomnego po powrocie z `waitpid()`
- `options` - opcje (zdefiniowane w manualu - `man 2 waitpid`). Wartą

wspomnienia jest opcja WNOHANG. Jeśli umieścimy wywołanie `waitpid()` z tą opcją w pętli, będziemy mogli monitorować położenie, jednocześnie nie blokując działania procesu rodzicielskiego, jeśli potomek jeszcze działa.

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

int main () {
    int status, exit_status;
    pid_t pid = fork();
    if (pid < 0)
        printf("ERROR! Nie mozna utworzyc nowego procesu\n");
    if (pid == 0) { //tutaj wejdzie tylko jako potomek
        printf("Potomek (PID: %d) uspiony...\n", getpid());
        sleep(5);
        exit(0); //wyjscie z potomka
    }
    while (waitpid(pid, &status, WNOHANG) == 0) { //tutaj tylko
jako rodzic
        printf("Czekam na zakonczenie potomka...\n");
        sleep(1);
    }
    exit_status = WEXITSTATUS(status); //wyciagniecie wartosci ze
zwracanej przez waitpid
    printf("Potomek (PID: %d) zakonczyl dzialanie zwracajac %d\n",
pid, exit_status);
    return 0;
}
```

■ W przypadku, jeśli proces potomny zakończy się bez wywołania funkcji `wait()`, przechodzi w stan zawieszenia i staje się on procesem zombie

@@

Zadanie 1:

Utworzyć plik `zadanie1.c`

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

int main () {
    printf("PID: %d\n", (int) getpid ());
    printf("PPID: %d\n", (int) getppid ());
    return 0;
}
```

uruchomić `zadanie1`

Zadanie 2:

Utworzyć plik zadanie2.c

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main () {
    pid_t child_pid;
    printf ("PID glownego programu: %d\n", (int) getpid ());
    child_pid = fork ();
    if (child_pid != 0) {
        printf ("to jest proces macierzysty, a jego PID to: %d\n",
            (int) getpid ());
        printf ("PID procesu potomnego: %d\n", (int) child_pid);
    }
    else
        printf ("to jest proces potomny, a jego PID to: %d\n", (int)
            getpid ());
    return 0;
}
```

uruchomić zadanie2

Zadanie 3:

Utworzyć plik zadanie3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main () {
    printf("Jestes w katalogu:\n");
    execl ("/bin/pwd", "pwd", (char *)0);
    perror ("Blad uruchamiania pwd");
    return 0;
}
```

uruchomić zadanie3

Zadanie 4:

Utworzyć plik zadanie4.c

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
int main () {
    int status, exit_status;
    pid_t pid = fork();
    if (pid < 0)
        printf("ERROR! Nie mozna utworzyc nowego procesu\n");
    if (pid == 0) { //tutaj wejdzie tylko jako potomek
        printf("Potomek (PID: %d) uspiony...\n", getpid());
        sleep(5);
        exit(0); //wyjscie z potomka
    }
    while (waitpid(pid, &status, WNOHANG) == 0) { //tutaj tylko jako rodzic
        printf("Czekam na zakonczenie potomka...\n");
        sleep(1);
    }
    exit_status = WEXITSTATUS(status); //wyciagniecie wartosci ze zwracanej
    przez waitpid
    printf("Potomek %d zakonczyl dzialanie zwracajac %d\n", pid, exit_status);
    return 0;
}
```

uruchomić przykład 4

Zadanie 5:

Utworzyć plik zadanie5.c

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char* argv[])
{
    pid_t mojpil, x;

    mojpil = getpid();
    printf("[%u]: Uruchamiam ls -l -a\n", mojpil);
    x = fork();
    if (x == 0)
    {
        if (execl("/bin/ls", "ls", "-l", "-a", NULL) == -1)
        {
            printf("Uruchomienie ls nie powiodlo sie\n");
        }
    }
    else
    {
        waitpid(x, NULL, 0);
        printf("[%u]: ls -l -a zakonczony\n", mojpil);
    }
    return 0;
}
```

uruchomić zadanie5

– Proszę zwrócić uwagę na parametry w wywołaniu `execl`. Ten wariant funkcji `exec*` wymaga podania jako pierwszego parametru ścieżki do pliku wykonywalnego (inna odmiana - `execlp` przeprowadza poszukiwanie wg bieżącej wartości zmiennej środowiskowej `PATH`). Drugi parametr ("`ls`") to `argv[0]` uruchamianego programu (a jak wiemy, `argv[0]` powinien zawierać nazwę samego uruchamianego programu), dopiero trzeci to pierwszy faktyczny parametr przekazywany uruchamianemu programowi. Odpowiednikiem `execv` korzystającym z `PATH` jest `execvp`.