Systemy Operacyjne 1
Laboratorium 2
„Procesy i sygnały w Linuksie”
(jeden tydzień)

dr inż. Arkadiusz Chrobot

7 października 2019

Wstęp

W tej instrukcji zawarte są informacje na temat tworzenia i obsługiwania procesów przez system Linux (Unix). Pierwsza cześć zawiera opis budowy pamięci logicznej procesu użytkownika, druga zawiera opis tworzenia nowego procesu z punktu z punktu widzenia programisty aplikacji. Trzecia część przedstawia pojęcie sygnału i objaśnia jego zastosowania. Część czwarta zawiera skrócone opisy najważniejszych funkcji, które związane są z opisywanymi zagadnieniami. Instrukcja kończy się listą zadań do wykonania na laboratorium.

1. Budowa procesu w Linuksie

W systemach uniksowych (w tym w Linuksie) przestrzeń procesu[1] użytkownika można podzielić na dwa konteksty: kontekst użytkownika i kontekst jądra. Pierwszy z nich może być podzielony na sześć obszarów: tekstu, stałych, zmiennych zainicjowanych, zmiennych niezainicjowanych, sterty i stosu. Drugi zawiera wyłącznie dane. Obszar tekstu zawiera rozkazy maszynowe, które są wykonywane przez sprzęt. Ten obszar jest tylko do odczytu, a więc może go współdzielić kilka procesów równocześnie. Obszar stałych jest również tylko do odczytu. We współczesnych systemach uniksowych jest łączony w jeden obszar z obszarem tekstu. Obszar zmiennych zainicjowanych zawiera zmienne, którym zostały przypisane wartości początkowe, ale proces może je dowolnie modyfikować. Obszar zmiennych niezainicjowanych (bss) zawiera zmienne, które mają wartość początkową zero, a więc nie trzeba ich wartości inicjujących przechowywać w pliku programu. Sterta (ang. *heap*) i stos (ang. *stack*) tworzą w zasadzie jeden obszar - sterta służy do dynamicznego przydzielania dodatkowego obszaru w pamięci, natomiast na stosie przechowywane są ramki stosu, czyli informacje związane z wywołaniem podprogramów. Sterta rozszerza się w stronę wyższych adresów, natomiast stos w stronę niższych adresów. Proces użytkownika nie ma bezpośredniego dostępu do kontekstu jądra, który zawiera informacje o stanie tego procesu. Ten obszar może być modyfikowany tylko przez jądro. Pewne wartości w tym kontekście mogą być modyfikowane z poziomu procesu użytkownika poprzez odpowiednie wywołania systemowe.

2. Tworzenie nowych procesów

Proces, czyli wykonujący się program, może stworzyć proces potomny używając funkcji `fork()` udostępnianej z poziomu biblioteki standardowej języka C. W systemie Linux funkcja ta jest „opakowaniem“ na wywołanie `clone()`, które nie jest wywołaniem standardowym, tzn. nie jest dostępne w innych systemach kompatybilnych z Uniksem i nie należy go bezpośrednio stosować w programach, które mają być przenośne. W Linuksie zastosowany jest wariant tworzenia procesów określany po angielsku *copy-on-write*. Oznacza to, że po stworzeniu nowego procesu współdzieli on zarówno obszar tekstu, jak i obszar danych (tj. stertę, stos, zmienne zainicjowane i niezainicjowane) z rodzicem. Dopiero, kiedy któryś z nich będzie próbował dokonać modyfikacji danych nastąpi rozdzielenie obszaru danych (proces potomny otrzyma kopię obszaru rodziciela). Aby wykonać nowy program należy w procesie potomnym użyć jednej z funkcji `exec()`. Sterowanie z procesu potomnego do procesu rodzicielskiego nigdy bezpośrednio nie wraca, ale proces rodzicielski może poznać status wykonania procesu potomnego wykonując jedną z funkcji `wait()`. Jeśli proces rodzicielski nie wykona tej funkcji, to zakończony proces proces potomny zostaje procesem *zombie*. W przypadku, gdy proces-rodziciel zakończy się wcześniej niż proces potomny, to ten ostatni jest „adoptowany” przez proces `init`, którego PID (identyfikator procesu) wynosi `1` lub inne procesy należące do jego grupy procesu rodzicielskiego.

Listing 1 przedstawia prosty program, w którym za pomocą wywołania funkcji `fork()` tworzony jest proces potomny. Jeśli ta funkcja zwróci wartość `-1`, to znaczy, że nie udało się jej utworzyć potomka. Opis wyjątku, który spowodował to niepowodzenie możemy uzyskać na ekranie dzięki funkcji `perror()`. Jak jej argument wystarczy przekazać ciąg znaków będący nazwą funkcji, ale w przypadku gdy ta funkcja jest używana kilkukrotnie w obrębie, a nawet kilku plików, jeśli program jest podzielony na osobne jednostki kompilacji, to można jeszcze dodać do tej nazwy numer wiersza i nazwę pliku, w którym ta funkcja jest wywoływana. Po zakończeniu swojej pracy proces potomny kończy się wywołując funkcję `exec()`. Proces

[1] Proces, to program, który został uruchomiony i się wykonuje.

rodzicielski czeka na jego zakończenie wywołując funkcję `wait()`. Ponieważ funkcja `exec()` nie zwraca kodu zakończenia, to nie jest on badany, ale ta czynność wykonywana jest w przypadku funkcji `wait()`.

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

void do_child_work(void)
{
        puts("Jestem potomkiem.");
        exit(0);
}

void do_parent_work(void)
{
        puts("Jestem rodzicem.");
        if(wait(0)<0)
                perror("wait");
}

int main(void)
{
        int pid = fork();
        if(pid<0)
                perror("fork");
        if(pid==0)
                do_child_work();
        else
                do_parent_work();
        return 0;
}
```

Listing 1: Tworzenie pojedynczego procesu potomnego

Rozważmy teraz bardziej skomplikowany przypadek. Spróbujmy utworzyć kilka procesów potomnych. Najprostsze rozwiązanie, jakie odruchowo przychodzi na myśli przedstawia listing 2. Czy jest ono poprawne? To zależy, co chcemy osiągnąć. Proszę zauważyć, że z każdą iteracją pętli wykonywane są dwie czynności: tworzony jest potomek i proces rodzicielski czeka na jego zakończenie. W ten sposób nie powstanie kolejny nowy potomek, zanim nie zakończy się jego poprzednik. Zatem procesy potomne wykonują się współbieżnie[2] z procesem rodzicielskim, a sekwencyjnie względem siebie.

[2]Współbieżne (ang. *concurrent*) wykonanie procesów na komputerze z jednym procesorem oznacza, że ten procesor jest dzielony, a więc przełączany między tymi procesami, a więc są one wykonywane „fragmentami" i nie zawsze da się przewidzieć w jakie kolejności. W przypadku komputera z wieloma procesami, wszystkie lub niektóre procesy, w zależności od ich liczby i liczby procesorów, wykonują się na osobnych procesorach, czyli *równolegle*. Angielski termin oznaczający współbieżność pochodzi stąd, że procesy *rywalizują* ze sobą o dostęp do procesora.

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

#define NUMBER_OF_CHILDREN 10

void do_child_work(void)
{
        puts("Jestem potomkiem.");
        exit(0);
}

void do_parent_work(void)
{
        puts("Jestem rodzicem.");
        if(wait(0)<0)
                perror("wait");
}

int main(void)
{
        int i;
        for(i=0;i<NUMBER_OF_CHILDREN;i++) {
                int pid = fork();
                if(pid<0)
                        perror("fork()");
                if(pid==0)
                        do_child_work();
                else
                        do_parent_work();
        }

        return 0;
}
```

Listing 2: Tworzenie sekwencyjnych procesów potomnych

Inną możliwość przedstawia listing 3. W tym programie funkcja `generate_child_processes()` tworzy w pętli określoną jej argumentem wywołania liczbę procesów potomnych. Proces potomny natychmiast po powstaniu wykonuje funkcję `do_child_work()`, a proces rodzicielski wykonuje następną iterację pętli, tworząc nowego potomka. W ten sposób wszyscy potomkowie mogą się wykonywać współbieżnie względem siebie i procesu rodzicielskiego. Funkcja `wait_for_children()` jest wykonywana wyłącznie przez proces rodzicielski. Procesy potomne wykonują jedynie `do_child_work()`, gdyż wywołuje ona na koniec funkcję `exit()`, która kończy działanie procesu. W funkcji `wait_for_children()` proces rodzicielski, w pętli wywołuje funkcję `do_parent_work()`, której zadaniem jest oczekiwania na zakończenie procesu potomnego. W tej sposób żaden z nich po zakończeniu procesu macierzystego nie zostanie procesem *zombie*.

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

#define NUMBER_OF_CHILDREN 10

void do_child_work(void)
{
        puts("Jestem potomkiem.");
        exit(0);
}

void do_parent_work(void)
{
        puts("Jestem rodzicem.");
        if(wait(0)<0)
                perror("wait");
}

void generate_child_processes(unsigned int number_of_children)
{
        int i;
        for(i=0;i<number_of_children;i++) {
                int pid = fork();
                if(pid<0)
                        perror("fork()");
                if(pid==0)
                        do_child_work();
        }
}

void wait_for_children(unsigned int number_of_children)
{
        int i;
        for(i=0;i<number_of_children;i++)
                do_parent_work();
}

int main(void)
{
        generate_child_processes(NUMBER_OF_CHILDREN);
        wait_for_children(NUMBER_OF_CHILDREN);

        return 0;
}
```

Listing 3: Tworzenie współbieżnych procesów potomnych

3. Sygnały

Sygnały można uznać za prostą formę komunikacji między procesami, ale przede wszystkim służą one do powiadomienia procesu, że zaszło jakieś zdarzenie, stąd też nazywa się je przerwaniami programowymi.

Sygnały są asynchroniczne względem wykonania procesu (nie można przewidzieć kiedy się pojawią). Mogą być wysłane z procesu do procesu lub z jądra do procesu. Programista ma do dyspozycji funkcję `kill()`, która umożliwia wysłanie sygnału do procesu o podanym PID. Z każdym procesem jest związana struktura, w której umieszczone są adresy procedur obsługi sygnałów. Jeśli programista nie napisze własnej funkcji obsługującej dany sygnał, to wykonywana jest procedura domyślna, która powoduje natychmiastowe zakończenie procesu lub inne, zależne od konfiguracji zachowanie. Część sygnałów można ignorować, lub zablokować je na określony czas. Niektórych sygnałów nie można samemu obsłużyć, ani zignorować, ani zablokować (np. SIGKILL).

4. Opis ważniejszych funkcji

`fork()` - stwórz proces potomny. Funkcja ta zwraca dwie wartości: dla procesu macierzystego - PID potomka, dla procesu potomnego `0`. Jeśli jej wywołanie się nie powiedzie, to zwraca wartość `-1`. Kodu programu z oprogramowanym zachowaniem potomka i rodzica został zaprezentowany na listingu 1.
Szczegóły: `man fork`
`clone()` - funkcja specyficzna dla Linuksa, służy do tworzenia nowego procesu.
Szczegóły: `man clone`
`getpid()` i `getppid()` - funkcje zwracają odpowiednio: PID procesu bieżącego i PID jego rodzica.
Szczegóły: `man getpid`
`sleep()` - służy do „uśpienia" procesu (zawieszenia jego działania) na określoną liczbę sekund.
Szczegóły: `man 3 sleep`
`wait` - nie jest to jedna funkcja, ale rodzina funkcji (`wait()`, `waitpid()`, `wait3()`, `wait4()`). Powodują one, że proces macierzysty czeka na zakończenie procesu potomnego. Status zakończenia procesu możemy poznać korzystając z odpowiednich makr.
Szczegóły: `man 2 wait`.
`exit()` - funkcja kończąca wykonanie procesu. Istnieje kilka innych podobnych funkcji.
Szczegóły: `man 3 exit`.
`exec` - rodzina funkcji (`execl()`, `execlp()`, `execle()`, `execv()`, `execv()`), które zastępują obraz w pamięci aktualnie wykonywanego procesu obrazem nowego procesu, odczytanym z pliku.
Szczegóły: `man 3 exec`.
`kill()` - funkcja powodująca wysłanie sygnału o określonym numerze do procesu o określonym PID.
Szczegóły: `man 2 kill`.
`signal()` - funkcja pozwala określić zachowanie procesu, po otrzymaniu odpowiedniego sygnału. Z tą funkcją powiązane są funkcje `sigblock()` i `sigsetmask()`. Współcześnie zalecane jest stosowanie `sigaction()` i `sigprocmask()` zamiast `signal()`.
Szczegóły: `man signal`, `man sigblock`, `man sigsetmask`, `man sigaction`, `man sigprocmask`.
`pause()` - funkcja powoduje, że proces czeka na otrzymanie sygnału.
Szczegóły: `man pause`.
`alarm()` - pozwala ustawić czas, po którym proces otrzyma sygnał SIGALRM.
Szczegóły: `man alarm`.
`perror()` - wypisuje na ekran komunikat związany z ostatnim napotkanym wyjątkiem pochodzącym od funkcji systemowej.
Szczegóły: `man 3 perror`.

5. Zadania

UWAGA! PROGRAMY NALEŻY NAPISAĆ Z PODZIAŁEM NA FUNKCJE Z PARAMETRAMI. KAŻDY PROGRAM MUSI SPRAWDZAĆ, CZY FUNKCJA, Z TYCH, KTÓRE SĄ OPISANE WYŻEJ NIE SYGNALIZUJE WYJĄTKU PODCZAS WYKONANIA, O ILE Z JEJ DOKUMENTACJI WYNIKA, ŻE MOŻE COŚ TAKIEGO ROBIĆ.

1. Napisz program, który utworzy proces potomny. Proces rodzicielski powinien wypisać swoje PID i PID potomka, natomiast proces potomny powinien wypisać swoje PID i PID rodzica.

2. Zademonstruj w jaki sposób mogą powstać w systemie procesy *zombie*.

3. Napisz program, który stworzy proces potomny. Proces macierzysty powinien poczekać na wykonanie procesu potomnego i zbadać status jego wyjścia.

4. Napisz program, który w zależności od wartości argumentu podanego w wierszu poleceń wygeneruje odpowiednią liczbę procesów potomnych, które będą się wykonywały współbieżnie. Każdy z procesów potomnych powinien wypisać 4 razy na ekranie swój `PID`, `PID` swojego rodzica oraz numer określający, którym jest potomkiem rodzica (1, 2, 3 …), a następnie usnąć na tyle sekund, ile wskazuje ten numer (pierwszy - 1 sekunda, 2 - dwie sekundy, trzeci - 3 sekundy, …). Proces macierzysty powinien poczekać na zakończenie wykonania wszystkich swoich potomków.

5. Napisz dwa programy. Program pierwszy stworzy proces potomny, a następnie zastąpi jego program drugim programem.

6. Napisz program, który wyśle do siebie sygnał `SIGALRM` i obsłuży go.

7. Napisz program, który stworzy dwa procesy. Proces rodzicielski wyśle do potomka sygnał `SIGINT` (można go wysłać „ręcznie“ naciskając na klawiaturze równocześnie `Ctrl + c`). Proces potomny powinien ten sygnał obsłużyć za pomocą napisanej przez Ciebie funkcji.

8. Napisz cztery osobne programy. Każdy z nich powinien obsługiwać wybrany przez Ciebie sygnał. Pierwszy z procesów będzie co sekundę wysyłał sygnał do drugiego procesu, drugi proces po odebraniu sygnału powinien wypisać na ekranie komunikat, a następnie przesłać sygnał do procesu trzeciego. Proces trzeci powinien zachowywać się podobnie jak drugi, a proces czwarty powinien jedynie wypisywać komunikat na ekranie. Odliczanie czasu w pierwszym procesie należy zrealizować za pomocą `SIGALRM`.

9. Napisz program, który udowodni, że obszar danych jest współdzielony między procesem potomnym i macierzystym do chwili wykonania modyfikacji danych przez jednego z nich.

10. Ze względów bezpieczeństwa zaleca się, aby w ramach funkcji obsługującej sygnał wykonywane były tylko proste czynności, jak np. ustawienie flagi informującej o otrzymaniu sygnału, a skomplikowane czynności żeby były wykonywane w osobnym kodzie. Przedstaw schemat takiego rozwiązania stosując proces macierzysty i potomny.

11. Pokaż w jaki sposób sygnały mogą być przez proces blokowane lub ignorowane.

12. Aby procesy potomne nie stawały się procesami zombie wystarczy, żeby proces macierzysty ignorował sygnał `SIGCHLD`. Napisz program, który sprawdzi, czy rzeczywiście tak się dzieje i co w takim przypadku zwraca `wait()` lub `waitpid()` po zakończeniu potomka.