Instrukcja do laboratorium Systemów Operacyjnych

(semestr drugi)

Ćwiczenie drugie

(jedne zajęcia)

Temat: Procesy i sygnały w Linuksie.

Opracowanie:

mgr inż. Arkadiusz Chrobot

Wprowadzenie

1. Budowa procesu w Uniksie.

W systemach uniksowych (w tym w Linuksie) przestrzeń procesu 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

Działający proces 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, ze 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 rodzicielskiego do procesu potomnego 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.

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". Oto fragment kodu, pozwalający oprogramować zachowanie potomka i rodzica:

  ```
  int porcpid = fork();
  if(procid == -1) exit(EXIT_FAILURE);
  ```

```
if(procpid == 0) {
   /*tu kod potomka*/
} else {
   /* tu kod rodzica*/
}
```

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 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.

Zadania:

1. Napisz program, który utworzy dwa procesy: macierzysty i 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 dwa procesy. Proces macierzysty powinien poczekać na wykonanie procesu potomnego i zbadać status jego wyjścia.
4. Napisz program, który w zależności od wartości argumentu podanego w linii 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 dwa procesy, a następnie program procesu potomnego zastąpi programem drugim.
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 pod 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 SIGALARM.
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.