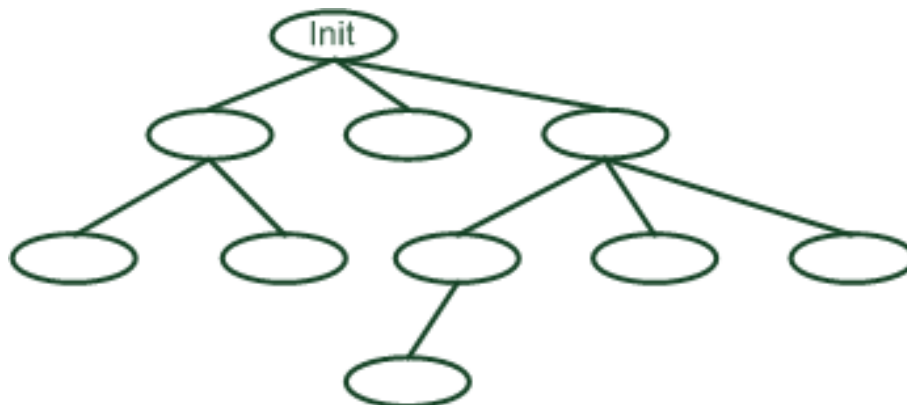


1. Procesy i współbieżność

Opracował: Sławomir Samolej
Politechnika Rzeszowska,
Katedra Informatyki i Automatyki,
Rzeszów, 2013.

1.1. Wprowadzenie

Proces to przestrzeń adresowa i pojedynczy wątek sterujący, który działa w tej przestrzeni, oraz potrzebne do tego zasoby systemowe. Minimalne zasoby do wykonywania się procesu to: procesor, pamięć i urządzenia wejścia-wyjścia. We współczesnych wielozadaniowych systemach operacyjnych w istocie każda instancja działającego programu jest procesem. Podczas uruchamiania programu w systemie operacyjnym następuje przeniesienie kodu programu z jakiegoś nośnika (najczęściej systemu plików) do pamięci operacyjnej. Kod programu jest „otaczany” pewnymi strukturami danych identyfikującymi go na czas wykonywania i zapewniającymi jego ochronę (tzw. blok kontrolny procesu). Następuje także powiązanie (planowanie przydziału) z programem zasobów systemu mikroprocesorowego, z których będzie korzystał. Zasobami przyszłego procesu są czas procesora, pamięć, system plików oraz urządzenia wejścia-wyjścia. Tak przygotowany do wykonywania program staje się procesem, który jest gotowy do wykonywania. Procesy w systemie operacyjnym tworzą drzewo (rys. 1). Każdy proces ma dokładnie jeden proces – rodzic. Procesy mogą posiadać wiele procesów-dzieci. W systemach uniksowych (w tym linux) wszystkie procesy są dziećmi pewnego pierwotnego procesu (Init). Procesy wykonywane są współbieżnie. Są albo przydzielane do różnych rdzeni procesora, albo wykonywane naprzemiennie na jednym procesorze w taki sposób, że użytkownik ma wrażenie, jakby wykonywały się ciągle.



Rys. 1. Drzewo procesów w systemie operacyjnym

Podstawowymi usługami systemu operacyjnego jest uruchamianie i nadzorowanie procesów. Istnieje również możliwość zatrzymywania (typowego lub awaryjnego) procesu, jak również monitorowania wykonywanych procesów. Kolejną usługą systemu jest dostarczenie mechanizmów komunikacji i synchronizacji procesów. Komunikacja pozwala na przesyłanie danych pomiędzy działającymi procesami, zaś synchronizacja to możliwość decydowania, kiedy pewne porcje obliczeń

mogą się odbywać w zależności o obliczeń wykonanych przez inne procesy. W dalszej części opracowania pokazane zostaną podstawowe programy monitorujące i zarządzające procesami o raz elementarne techniki komunikacji i synchronizacji pomiędzy procesami w systemie Linux.

1.2. Narzędzia do monitorowania systemu i procesów

Najbardziej uniwersalnymi narzędziami do monitorowania systemu Linux są pewne polecenia konsoli. W tabeli Tab. 1 wymienione zostaną najważniejsze z nich.

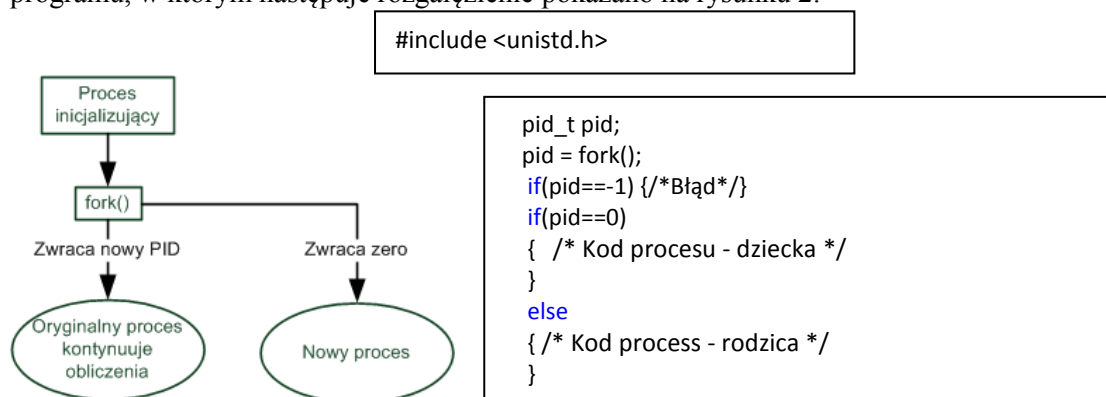
Tab. 1 Wybrane polecenia konsoli do monitorowania stanu systemu

Nazwa	Opis
ps	pobieranie informacji o aktywnych procesach
warianty wywołania:	
ps -ef	wszystkie procesy w systemie
ps -axjf	drzewo procesów
ps -eLf	informacja o wątkach
top	dynamiczna lista aktywnych procesów w systemie
pstree	drzewo procesów w systemie
nice	uruchomienie programu z zadaniem priorytetem
renice	zmiana priorytetu uruchomionego procesu
kill	przesłanie sygnału do procesu pracującego w systemie operacyjnym

Za pomocą wymienionych poleceń istnieje możliwość analizy systemu operacyjnego jako zbioru współwykonawców procesów. Pewne polecenia (np. kill) pozwalają na usuwanie procesów z systemu lub modyfikowanie ich priorytetów (np. nice, renice). W większości interfejsów graficznych systemu Linux dostępne jest również narzędzie o nazwie „Monitor systemu” (w wersji Linux LUbuntu: Menadżer zadań). Pozwala ono na monitorowanie procesów na poziomie interfejsu graficznego.

1.3. Tworzenie aplikacji wieloprocessowych

W systemach uniksowych istnieje możliwość rozgałęzienia programu na dwa (lub więcej) współbieżnie wykonywane procesy. Służy do tego specjalna funkcja o nazwie fork(). Schemat programu, w którym następuje rozgałęzienie pokazano na rysunku 2.



Rys. 2. Technika rozgałęziania procesów z zastosowaniem instrukcji `fork()`.

Rozgałęzienie procesu na proces macierzysty i na proces potomny następuje po wywołaniu funkcji `fork()`. Jeśli funkcja zwróci wartość `-1`, to rozgałęzienie zakończyło się fiaskiem. Jeśli rozgałęzienie zakończyło się sukcesem to wartość zwrócona przez funkcję `fork()` informuje nas, w którym procesie w danej chwili jesteśmy. Jeśli funkcja zwróciła `0`, to jesteśmy w procesie potomnym (dziecku), jeśli funkcja zwróciła inną wartość, to jesteśmy w procesie – rodzicu. Należy zwrócić uwagę, że tworzony jest **jeden** program, który opisuje zachowanie dwu (lub kilku) procesów. W szablonie na rysunku 1 pokazano, jak zidentyfikować instrukcje wykonywane przez rodzica, a jak te wykonywane przez dziecko z zastosowaniem instrukcji `if`.

Kod źródłowy 1 zawiera pełny program, w którym następuje rozdzielenie na dwa procesy. Każdy z procesów pięciokrotnie wypisuje pewien tekst na ekranie i zawiesza swoje działanie na 1 sekundę (`sleep(1)`); Warto zwrócić uwagę na fakt, że po rozdzieleniu programu z zastosowaniem instrukcji `fork()` proces potomny dziedziczy po procesie-rodzicu wszystkie zmienne i ewentualne uchwytty do kanałów komunikacyjnych i plików. Procesy jednak nie współdzielą tych zmiennych. Każdy z nich ma dostęp do swojej kopii zmiennych lokalnych i globalnych. Funkcje `exit()` służą do prawidłowego kończenia procesów.

Kod źródłowy 1. Program tworzący 2 procesy.

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
    pid_t pid;
    int i;
    printf("Start programu\n");
    pid = fork();
    if(pid == -1) {exit(EXIT_FAILURE);}
    if(pid == 0)
    {
        for(i=0; i<5; i++) {printf("Tu dziecko\n"); sleep(1);}
        exit(EXIT_SUCCESS);
    }
    else
    {
        for(i=0; i<5; i++) {printf("Tu rodzic\n"); sleep(1);}
        exit(EXIT_SUCCESS);
    }
}
```

1.4. Komunikacja

W systemach uniksowych możliwe jest zorganizowanie komunikacji pomiędzy procesami na wiele sposobów. Jednym z prostszych jest zastosowanie potoków. Potok powołuje się za pomocą specjalnej funkcji `pipe()`, do której przekazywana jest dwuelementowa tablica liczb typu całkowitego. Kiedy funkcja jest wywoływana wypełnia ona tę tablicę specjalnymi liczbami będącymi uchwytami do potoku (systemowego kanału komunikacyjnego). Dwa zwrócone uchwyty połączone są w specjalny sposób. Wszystkie dane zapisane do uchwytu 1, mogą być odczytane przez dostęp do uchwytu 0. Przesyłanie danych z i do takiego potoku musi się odbywać z zastosowaniem funkcji `read()`, `write()`.

Kod źródłowy 2 pokazuje pełny program, w którym jeden proces wysyła dane do drugiego z zastosowaniem potoku.

Kod źródłowy 2. Komunikacja między procesami z zastosowaniem potku.

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

int main()
{
    int potoki[2];
    char dane_wysylane[10] = "0";
    char dane_odbierane[10] = "0";
    int dane_przetworzone = 0;
    pid_t pid;
    int pipe_result;
    int i;

    pipe_result=pipe(potoki);
    if(pipe_result!=0) {exit(EXIT_FAILURE);}

    pid = fork();
    if(pid==-1) {exit(EXIT_FAILURE);}
    if(pid==0)
    {
        for(i=0;i<5;i++)
        {
            dane_przetworzone = read(potoki[0], dane_odbierane, 10);
            printf("Odczytano %s bajtów: %d\n", dane_odbierane, dane_przetworzone);
            sleep(1);
        }
        exit(EXIT_SUCCESS);
    }
    else
    {
        for(i=0;i<5;i++)
        {
            dane_przetworzone = write(potoki[1], dane_wysylane, 10);
            printf("Wyslano %d bajtów\n", dane_przetworzone);
            dane_wysylane[0]++;
        }
        exit(EXIT_SUCCESS);
    }
}
```

Jest to modyfikacja programu podanego w kodzie źródłowym 1. W programie jest zdefiniowana tablica „potoki” do przechowywania uchwytów do kanału komunikacyjnego. Tablice „dane_wysylane”, „dane_odbierane” oraz „dane_przetworzone” są stosowane jako bufory które odpowiednio zawierają: dane do wysłania, dane odebrane, informację ile danych przesłano. Zmienna „pid” służy do rozpoznawania, w którym procesie wykonywane jest przetwarzanie, zaś zmienna „pipe_result” pozwala na przechwycenie rezultatu działania funkcji pipe() i sprawdzenie, czy potok został prawidłowo zdefiniowany. W programie następuje najpierw utworzenie potoku, a następnie rozdzielenie na dwa procesy. Ponieważ proces potomny dziedziczy zmienne po procesie-rodzicu otrzymuje on dostęp do kanału komunikacyjnego zdefiniowanego przez rodzica (tablica „potoki” u dziecka zawiera już wcześniej utworzone uchwyty do potoku). Proces-dziecko próbuje 5-krotnie odczytać dane z potoku (wywołanie funkcji read()), przy czym po każdym odczycie zawiesza swoje

wykonywanie na 1 sekundę. Proces rodzica pięciokrotnie wysyła dane do potoku (wywołanie funkcji `write()`). Opisy nowych funkcji zastosowanych w kodzie źródłowym 2 zawarto w tabelicy 2.

Tab. 2 Specyfikacje wybranych funkcji zastosowanych w kodzie źródłowym 2.

int pipe(int potoki[2]);	
Funkcja zwraca:	0, jeśli utworzenie potoku zakończyło się sukcesem kod błędu w przeciwnym wypadku
potoki[]	tablica uchwytów do potoku wypełniania przez funkcję pipe
size_t write(int fildes, const void *buf, size_t nbytes);	
fildes	– uchwyt do pliku/potoku
buf	– bufor z danymi do zapisu
nbytes	– ilość danych do zapisu
Funkcja zwraca ilość zapisanych bajtów, 0 gdy koniec pliku, -1 gdy błąd	
size_t read(int fildes, void *buf, size_t nbytes);	
fildes	– uchwyt do pliku/potoku
buf	– bufor na dane
nbytes	– maksymalna ilość bajtów do odczytania
Funkcja zwraca ile bajtów odczytała, 0-nic nie przeczytano koniec pliku, -1 – błąd.	

1.5. Synchronizacja

Synchronizacja to wypełnienie ograniczeń dotyczących kolejności wykonywania pewnych akcji przez procesy (np. pewna akcja wykonywana przez jeden proces może nastąpić tylko wtedy, gdy pewna inna akcja została wykonana w innym procesie). Podstawowym mechanizmem umożliwiającym synchronizację procesów jest semafor.

Semafor S jest obiektem systemu operacyjnego z którym związany jest licznik L zasobu przyjmujący wartości nieujemne. Na semaforze zdefiniowane są atomowe (takie, których nie może przerwać żadne zdarzenie w systemie operacyjnym) operacje `sem_init`, `sem_wait` i `sem_post`. Definicje operacji na semaforze pokazano w tabeli 3.

Tabela 3. Definicje operacji na semaforze.

Operacja	Oznaczenie	Opis
Inicjacja semafora S	<code>sem_init(S,N)</code>	Ustawienie licznika semafora S na początkową wartość N.
Zajmowanie	<code>sem_wait(S)</code>	Gdy licznik L semafora S jest dodatni ($L > 0$) zmniejsz go o 1 ($L = L - 1$). Gdy licznik L semafora S jest równy zero ($L=0$) zablokuj proces bieżący.
Sygnalizacja	<code>sem_post(S)</code>	Gdy istnieje jakiś proces oczekujący na semaforze S to odblokuj jeden z czekających procesów. Gdy brak procesów oczekujących na semaforze S zwiększ jego licznik L o 1 ($L=L+1$).

Schemat posługiwania się semaforem do synchronizacji pracy pomiędzy dwoma procesami pokazano na rysunku 3. Celem synchronizacji jest zapewnienie, że instrukcja Y wykona się po zakończeniu instrukcji A, przy czym instrukcje A i Y należą do różnych współbieżnie wykonywanych procesów.

W celu zapewnienia takiej synchronizacji należy powołać do życia semafor i zainicjować go wartością początkową 0. Możliwe są dwa scenariusze. W pierwszym instrukcja X wykona się szybciej, niż instrukcja A i w konsekwencji instrukcja `sem_wait()` wykona się wcześniej niż instrukcja `sem_post()`. Spowoduje to zatrzymanie pracy procesu P1 do momentu zakończenia instrukcji A i wykonania instrukcji `sem_post()` w procesie P2. Taką sekwencję synchronizacji ilustruje przebieg czasowy znajdujący się po lewej stronie rysunku 3. W drugim scenariuszu założono, że instrukcja A wykona się przed zakończeniem instrukcji X. W konsekwencji operacja `sem_post()` wykona się przed wykonaniem operacji `sem_wait()`. W tym przypadku proces P1 nie zostanie zatrzymany, ponieważ przed wykonaniem instrukcji `sem_wait()` semafor zostaje zwiększony o 1. W każdym przypadku zapewnione jest osiągnięcie założonego celu (instrukcja Y rozpoczyna wykonywanie po zakończeniu instrukcji A).

```
sem_init(&cosyn, 0) /* wartość początkowa 0 */
```

```
P2() {
    /* process sygnalizujący */
    instrukcja A;
    sem_post(&cosyn)
    instrukcja B;
}
```

```
P1() {
    /* process czekający */
    instrukcja X;
    sem_wait(&cosyn)
    instrukcja Y;
}
```



Rys. 3. Synchronizacja procesów z zastosowaniem semafora.

Kod źródłowy 3 pokazuje sposób realizacji synchronizacji 2 procesów z zastosowaniem semaforów. Program stosuje semafor z biblioteki `pthread`. Musi być skompilowany z opcją kompilatora `-pthread`. Funkcja `sem_open()` tworzy semafor. Funkcja `sem_init()` ustala jego początkową wartość. W programie proces-dziecko oczekuje w pętli niekończącej na sygnalizację semafora, która odbywa się pięciokrotnie w procesie-rodzicu. Komunikat „Nastąpiła synchronizacja” jest wysyłany na konsolę po synchronizacji.

Tabela 4 zawiera specyfikację wybranych funkcji biblioteki `pthread` sterujących pracą semaforów.

Tabela 4. Wybrane funkcje sterujące semaforami w bibliotece `pthread`.

Utworzenie semafora:	
sem_t *sem_open(const char *name, int oflag);	
name	– nazwa semafora w systemie, powinna zacząć się od znaku „/”.
oflag	– jest ustawiana na <code>O_CREAT</code> gdy chcemy utworzyć semafor (jeśli dodamy flagę <code>O_EXCL</code> funkcja zwróci błąd, w przypadku, gdy taki semafor już istnieje).

mode_t – ustalenie kontroli dostępu do semafora

value – ustalenie wartości początkowej semafora.

Funkcja zwraca „uchwyt” do semafora.

Inicjalizacja semafora:

int sem_init(sem_t *sem, int pshared, unsigned int value);

pshared - 0-semafor jest dzielony pomiędzy wątki, >0 może być dzielony pomiędzy procesy

value - początkowa wartość semafora

Czekanie na semaforze:

int sem_wait(sem_t *sem);

Sygnalizacja:

int sem_post(sem_t *sem);

Kod źródłowy 3. Synchronizacja 2 procesów z zastosowaniem semafora.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/times.h>
#include <semaphore.h>

int main()
{
    int a=0,i;
    pid_t pid;
    sem_t *sem;

    sem=sem_open("/thesem", O_CREAT);
    sem_init(sem, 1, 0);
    pid = fork();

    if(pid==-1) {exit(EXIT_FAILURE);}
    if(pid==0)
    { while(1)
      {
          sem_wait(sem);
          printf("Nastapila synchronizacja\n");
      }
    }
    else
    { for(i=0;i<5;i++)
      {
          printf("Zaraz zezwole na odblokowanie drugiego procesu...\n");
          sem_post(sem);
          sleep(1);
      }
    }
    exit(EXIT_SUCCESS);
}
```

1.6. Proponowany przebieg ćwiczenia

1. Uruchomić konsolę w Linux i przetestować działanie poleceń ps, top, pstre na domyślnym zbiorze procesów w systemie Linux. Uruchomić polecenie: „man ps” oraz „man top” w celu zrozumienia, jakie parametry procesów są wyświetlane w poszczególnych kolumnach odpowiedzi polecenia ps i top.

2. Napisać program wykonujący pętlę nieskończoną, np.:

```
int main()
{ while(1);
  return 0;
}
```

Uruchomić program i zaobserwować jego zachowanie w systemie z zastosowaniem poleceń ps, top, pstree oraz menadżera zadań. Usunąć proces za pomocą polecenia kill (w poleceniu należy podać numer PID procesu, który chcemy „zabić”). Do usunięcia procesu można się posłużyć drugą konsolą.

3. Uruchomić przykładową aplikację „two_processes.c”. Zaobserwować, czy z chwilą jej uruchomienia rzeczywiście w systemie następuje utworzenie dwu nowych procesów. Aby wydłużyć działanie aplikacji można zwiększyć ilość wykonywanych pętli.

4. Uruchomić przykładową aplikację „two_processes_pipe.c”. Zwrócić uwagę, w jaki sposób następuje przesyłanie danych pomiędzy procesami. Zmodyfikować aplikację w taki sposób, aby dane były nadawane co 5 sekund. Jak to wpłynie na tempo odbierania danych i zachowanie procesu odbierającego.

5. Uruchomić przykładową aplikację „two_processes_pipe.c”. Zwrócić uwagę, w jaki sposób zrealizowana jest synchronizacja dwu procesów.

6. Zaproponować program, który tworzy 2 procesy. Pierwszy proces w sposób nieskończony generuje kolejne liczby parzyste i wysyła je na konsolę, drugi w sposób nieskończony – kolejne liczby nieparzyste i wysyła je na konsolę. Pierwszy proces po wygenerowaniu liczby oczekuje 2 sekundy, drugi – 3. Po pewnym czasie „zabić” jeden z procesów.

7. Zaproponować program, który stworzy 3 procesy. Pierwszy, co 1 sekundę wypisuje tekst „Pozdrowienia”, drugi – co 2 sekundy „Czesc”, a trzeci, co 5 sekund „Witam”.

1.7. Literatura

- [1] Linux : programowanie / Neil Matthew, Richard Stones, RM, 1999
- [2] Linux. Niezbędnik programisty/ John Fusco, Helion 2009
- [3] Programowanie w Linuksie / K. Kuźniar, K. Lal, T. Rak, Helion, 2012