Michał Bronikowski

Systemy Operacyjne 2017
problem złodzieja jabłek

wersja z procesami ciężkimi

15 listopada 2017

Spis treści

1. Opis zadania

1.1 Problem złodzieja jabłek

Problem został przeze mnie wymyślony na potrzeby zadania. Wzorowałem się na klasycznym problemie synchronizacji tj. problemie producenta i konsumenta. Powiedzmy, że jesteśmy złodziejem jabłek i w naszej miejscowości są dwa sady A i B, w których w dość szybkim tempie przybywa jabłek. Kradniemy jabłka z sadu A lub z sadu B. W międzyczasie w sadach, dzięki działaniu właścicieli i natury jabłek przybywa. Problem polega na takim zsynchronizowaniu procesów sadów A i B oraz procesu złodzieja, żeby złodziej nie ukradł wszystkich jabłek z sadów, ponieważ to zwróciłoby uwagę właściciela i nasz złodziej mógłby mieć problemy.

1.2 Rozwiązanie

W rozwiązaniu korzystam z dwóch semaforów:

- skradziono
- uroslo

Złodziej czeka aż w sadach przybędzie jabłek co będzie skutkowało opuszczeniem semafara uroslo. Po czym przystępują do kradzieży jabłek z losowo wybranego przez siebie sadu. Po kradzieży zwalnia się semafor skradziono, a zamyka uroslo. Po zwolnieniu semafora skradziono w sadach ponownie rusza, produkcja jabłek.

1.3 Implementacja

Program należy uruchamiać na komputerze pod kontrolą systemu operacyjnego Linux. Program składa się z jednego pliku źródłowego sem.c. Załączony został również Makefile pomagający skompilować program.

```
$ make
```

1: Skompilowanie programu poleceniem make

Program uruchamiamy poleceniem:

```
$ ./sem
```

2: Uruchomienie programu

W programie zdefiniowałem strukturę Zasob_Dzielony

```
struct Zasob_Dzielony
{
    sem_t skradziono ;
    sem_t uroslo ;
    int jablko_z_A ;
    int jablko_z_B ;
    int zA,zB;
    int gdzie;
} *z_dz ;
```

3: struktura Zasob_Dzielony

Struktura zawiera elementy, które będą współdzielone przez wszystkie procesy programu.
Funkcja zarządzająca naszym złodziejem została zdefiniowana w następujący sposób:

```
void zlodziej()
{
    while(1)
    {
      sem_wait(&z_dz->uroslo);

      sad = rand()%2+1;
      if(sad == 1)
      {
          if(z_dz->jablko_z_A < 1)
          {
              printf("Nie ukradne z sadu A - nie maja ju  jablek\n");

          }
          else
          {
            --z_dz->jablko_z_A;
            z_dz->zA++;
          printf("Kradne jablko z sadu A mam juz %d jab ek z sadu A i %d jablek z sadu B \n "
    ,z_dz->zA,z_dz->zB);
          }
          sem_post(&z_dz->skradziono);
          z_dz->gdzie=1;
      }
      else
      {   if(z_dz->jablko_z_B < 1)
          {
              printf("Nie ukradne z sadu B - nie maja juz jablek\n");
          }
          else
          {
          --z_dz->jablko_z_B;
          z_dz->zB++;
          printf("Kradne jablko z sadu B mam juz %d jab ek z sadu A i %d jablek z sadu B \n "
    ,z_dz->zA,z_dz->zB);
          }
          sem_post(&z_dz->skradziono);
          z_dz->gdzie=2;
      }

    }
}
```

4: Funkcja zarządzająca złodziejem

Po każdej dokonanej kradzieży złodziej zaznacza, z którego sadu ukradł jabłka, poprzez ustawienie wartości współdzielonej zmiennej gdzie odpowiednio na 1 lub 2. Dzięki temu wiemy, w którym sadzie produkcja powinna ruszyć szybciej, aby w sad był konkurencyjny i nie zbankrutował (unikamy sytuacji, w której w sadzie nie będzie żadnego jabłka).

Przejdźmy do zarządania sadami.

```
void rosnij()
{
    while (1)
    {
        sem_wait(&z_dz->skradziono);
        if((z_dz->gdzie) == 1)
        {
            printf("Jestem z sadu A i mam %d jablek\n", ++z_dz->jablko_z_A);
        }
        else
        {
            printf("Jestem z sadu  B i mam %d jablek\n", ++z_dz->jablko_z_B);
        }
        sem_post(&z_dz->uroslo);
    }
}
```

5: Funkcja zarządzająca produkcją jabłek w sadzie

Po zwolnieniu semafora skradziono sprawdzam, w którym sadzie ostatnio skradziono jabłka i tam przystępuję do produkcji kolejnych. W drugim sadzie jabłek nie powinno zostać, ponieważ reguluję wartość startową semaforów jako 5 zarówno dla złodzieja, jak i sadowników.

W funkcji głównej programu inicjalizuję semafory i inicjalizuję wartości odpowiadające za ilość jabłek w poszczególnych sadach na 5.

```
int main()
{
    z_dz = mmap( NULL , sizeof( struct Zasob_Dzielony ) , PROT_READ | PROT_WRITE , MAP_SHARED
    | MAP_ANONYMOUS , -1 , 0 );
    sem_init( & z_dz->uroslo , 1 , 5 );
    sem_init( & z_dz->skradziono , 1 , 5 );
    z_dz->jablko_z_A = 5;
    z_dz->jablko_z_B = 5;
    pid = fork();

    if (pid < 0)
    {
        perror ("Niestety cos sie zepsulo :( \n");
        exit(1);
    }
    else if (pid == 0)
    {
        rosnij();
        exit(0);
    }

    zlodziej();
    return 1;
}
```

6: Funkcja głowna

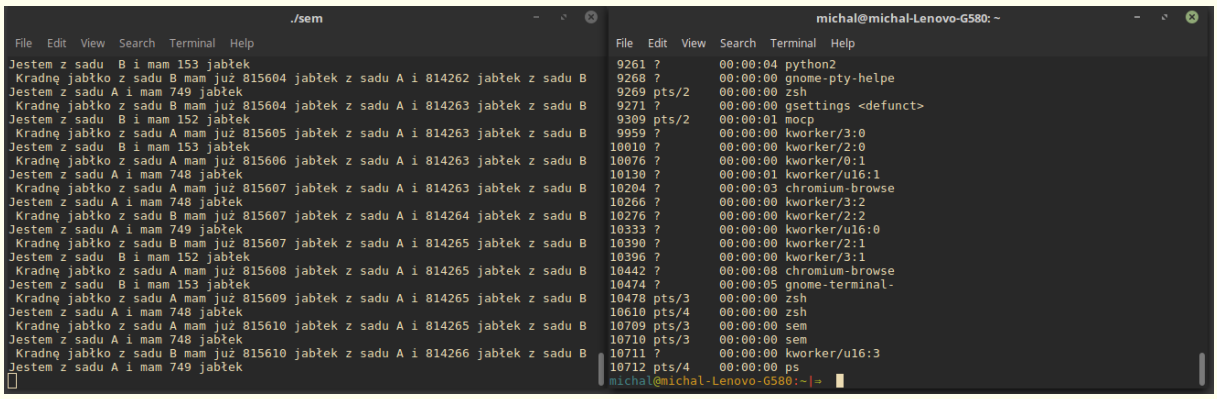
Wartość zmiennej pid jest identyfikatorem procesu pid = 0 odpowiada za proces sadowników i pid = 1 określa proces złodzieja. Sprawdzam również, czy nie nastąpił niespodziewany błąd związany z uruchomionymi procesami, czyli przypadek, gdy pid < 0.

2. Testowanie

Chcąc sprawdzić, czy program rzeczywiście korzysta z procesów. Uruchamiamy go i jednocześnie sprawdzamy aktualnie działające procesy poleceniem:

```
$ ps -A
```

7: Polecenie wyświetlające działające procesy



Procesy stworzone przez nasz program są widoczne jako: