Uniwersytet Bielsko-Bialski

LABORATORIUM

Obliczeń Równoległych i Systemów Rozproszonych

Sprawozdanie nr 7

Semafory posix

GRUPA: 2B / SEMESTR: 5 / ROK: 3

Igor Gawłowicz / 59096

Cel ćwiczenia

- Zrozumienie i praktyczne wykorzystanie semaforów nazwanych i nienazwanych w systemie operacyjnym.
- Porównanie działania semaforów zgodnych z standardem POSIX i UNIX System V.
- Praktyczne zastosowanie semaforów w zarządzaniu zasobami współdzielonymi między procesami lub wątkami.
- Nauka używania funkcji związanych z semaforami, takich jak `sem_open()`, `sem_close()`, `sem_unlink()`, `sem_init()`, `sem_destroy()`, `sem_wait()`, i `sem_post()`.

Przebieg ćwiczenia

Omawiane do tej pory rozwiązanie semaforów pochodzi z standardu UNIX System V i jest zasadniczo ukierunkowane na bardzo poważne zastosowania, kiedy dla realizacji określonego zadania wymagane jest rozstrzygnięcie bardzo złożonych problemów synchronizacji procesów.

Chociaż rozwiązanie UNIX SysV zawiera się on w standardzie POSIX, to jednak ten ostatni oferuje również i całkowicie odmienne podejście do problemu (dające docelowo identyczne możliwości zastosowań), równocześnie nie są elementem interface'u programowania IPC.

- POSIX zakłada w procesie jego utworzenia zawsze pojedynczy semafor, podczas gdy SysV zbiorowość semaforów;
- POSIX oferuje możliwość definiowania semaforów jako nazwanych lub nienazwanych pierwsze dają możliwość synchronizacji między całkowicie niepowiązanymi procesami, natomiast drugie umiejscawiane są w obrębie segmentu dzielonego pamięci;
- POSIX'owe przyjmują nieujemne wartości licznika i może być on zmieniany wyłącznie o ±1, w SysV nie istnieją tego rodzaju ograniczenia;
- POSIX nie daje możliwości ingerowania wobec uprawnień odnośnie użycia semafora
- semafory POSIX są znacznie bardziej efektywne, w sensie wydajności, niż SysV;
- semafory POSIX są jednak - z punktu widzenia użytkownika - znacznie bardziej elementarne i poręczne

Chęć użycia semafora nazwanego wiąże się z uprzednim jego otwarciem **sem_open().**

W momencie kiedy nie jest do niczego potrzebny należy dokonać zamknięcia **sem_close()**. Ostateczne usunięcia danego semafora dokonuje **sem_unlink()**.

Utwórzmy semafor nazwany POSIX, zainicjujemy i odczytamy stan jego licznika.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <semaphore.h>
int main( int argc,char** argv )
{
    sem_t *id;
    pid_t child;
```

```
    int status;
    id = sem_open( "szlaban",O_CREAT,S_IRUSR|S_IWUSR,0 );
    switch( (int)(child=fork()) )
    {
        case -1:
            perror("...fork()...");
            exit( 1 );
            break;
        case 0:// ...kod dla potomka
            printf("*** [%u] potomek czeka na semafor (%p)\n",
            (unsigned)getpid(),id);
            sem_wait( id );
            printf( "*** [%u] potomek zakończył\n",(unsigned)getpid() );
            exit( 0 );
        default: //...kod dla procesu nadrzędnego
            printf( "*** [%u] ustawia semafor (%p)\n",
            (unsigned)getpid(),id );
            sem_post( id );
            if( !wait( &status ) )
            {
                perror( "... wait()..." );
                exit( 2 );
            }
            else
            {
                printf( "*** [%u] wygląda, że to wszystko\n",
                (unsigned)getpid() );
                sem_close( id );
                sem_unlink( "szlaban" );
                }
    }
    return 0;
}
```

Po kompilacji i uruchomieniu otrzymamy taki wynik

```
$ ./named
*** [1304] ustawia semafor (0x7b7ffb74b000)
*** [1305] potomek czeka na semafor (0x7b7ffb74b000)
*** [1305] potomek zakończył
*** [1304] wygląda, że to wszystko
```

Wywołanie funkcji sem_unlink() - tradycyjnie, jak w przypadku wszystkich unlink() dokonuje usunięcia dowiązania

Semafory nienazwane POSIX nie stanowią dowiązań do systemu plików, a więc użycie nie musi poprzedzać użycie funkcji plikowych typu open(), close() czy – w końcu - unlink().

Użycie semaforów nienazwanych POSIX wymaga natomiast wywołań innych funkcji alokujących obszar pamięci dla niego sem_init() i zwalniającej sem_destroy().

Gdyby skorygować kod z przykładu wcześniejszego, dla wariantu semafora nienazwanego

```
#include <stdio.h> //...operacje wejścia/wyjścia
#include <sys/stat.h>// ...maski uprawnień (choć niekonieczne)
#include <fcntl.h> //...stała OCREAT
#include <semaphore.h> //...semafory POSIX, w ogólności (konieczne
int main( int argc,char **argv )
{
    sem_t *id; //...identyfikator tworzonego semafora
    int counter;// ...licznik semafora (pomocniczo)
    char name[] = "km";// ...nazwa semafora (jeżeli nazwany, jakaś musi być)
    counter = 7; //...inicjujemy zmienną dla licznika
    //...ostatecznie próbujemy utworzyć/otworzyć semafor (nazwany)
    id=sem_open(name,O_CREAT,S_IRUSR|S_IWUSR,(unsigned)counter);
    if(id!=SEM_FAILED)
    {
    //...jeżeli się udało, to ciąg dalszy
    printf( "*** otwarto semafor\t%s\n",name );
    if( !sem_getvalue( id,&counter ) )// ...odczyt licznika
    {
        printf( " licznik semafora\t%d\n",counter );
    }
    else
    {
        perror( "*** sem_getvalue()\t" );
    }
    if(!sem_close(id))
    {
        printf( "*** zamknięto semafor\t%s\n",name );
    }
    else
    {
        perror( "*** sem_close()\t" );
    }
    //...na wstępie pozostawmy te linie w komentarzu
    if(!sem_unlink(name))
    {
        printf( "*** usunięto semafor\t%s\n",name );
    }
    else
    {
        perror( "*** sem_unlink()\t" ); }
    }
    else
    {
        perror( "....sem_open()...." );
    } //...gdyby coś poszło nie tak
    return 0;
}
```

Co zwróci nam tym razem następujący wynik

```
$ ./unnamed
*** otwarto semafor     km
 licznik semafora       7
*** zamknięto semafor   km
*** usunięto semafor    km
```

Akcje P() i V(), czyli wait() i signal() realizowane są wywołaniami

Pierwsza z funkcji – sem_wait() - powoduje dekrementację licznika semafora, jeżeli jego wartość jest większa od zera to proces będzie kontynuował, w przeciwnym razie zostanie zablokowany (sem_trywait(), generuje zamiast tego błąd). Druga zaś powoduje inkrementację, jeżeli wartość licznika stanie się większa od zera, to proces zablokowany na nim będzie obudzony (o ile taki jest).

Właściwie użycie semaforów POSIX w relacji do SysV nie różni się istotnie (jeżeli pamiętać będziemy o różnicach podanych na wstępie). Poniżej prosty przykład ilustracyjny.

Gdyby skorygować kod z przykładu wcześniejszego dla wariantu semafora nienazwanego

```c
#include <stdio.h>
#include <semaphore.h>
int main( int argc,char **argv )
{
    sem_t id;
    int counter;
    counter = 7; //...wartość inicjująca dla licznika semafora
    if( !sem_init( &id,0,counter ) )
    {
        printf( "*** inicjacja semafor\n" );
        if( !sem_getvalue( &id,&counter ) )
        {
            printf( " licznik semafora\t%d\n",counter );
        }
        else
        {
            perror( "*** sem_getvalue()\t" );
        }
        if(!sem_destroy( &id ))
        {
            printf("*** usunięcie semafor\n");
        }
        else
        {
            perror( "*** sem_destroy()\t" );
        }
    }
    else
    {
        perror( "....sem_init()...." );
    }
```

```
    return 0;
}
```

Ponownie otrzymamy taki sam wynik

```
$ ./unnamed2
*** inicjacja semafor
 licznik semafora       7
*** usunięcie semafor
```

Właściwe użycie semaforów POSIX w relacji SysV nie różni się zbyt istotnie. Poniżej prosty przykład.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <semaphore.h>
int main( int argc,char** argv )
{
    sem_t *id;
    pid_t child;
    int status;
    id = sem_open( "szlaban",O_CREAT,S_IRUSR|S_IWUSR,0 );
    switch( (int)(child=fork()) )
    {
    case -1:
        perror("...fork()...");
        exit( 1 );
        break;
    case 0:
        printf("*** [%u] potomek czeka na semafor (%p)\n",
        (unsigned)getpid(),id);
        sem_wait( id );
        printf( "*** [%u] potomek zakończył\n",(unsigned)getpid() );
        exit( 0 );
    default:
        printf( "*** [%u] ustawia semafor (%p)\n",
        (unsigned)getpid(),id );
        sem_post( id );
        if( !wait( &status ) )
        {
            perror( "... wait()..." );
            exit( 2 );
        }
        else
        {
            printf( "*** [%u] wygląda, że to wszystko\n",
```

```
            (unsigned)getpid() );
            sem_close( id ); sem_unlink( "szlaban" );
        }
    }
    return 0;
}
```

```
$ ./forky
*** [1482] ustawia semafor (0x7b1c53743000)
*** [1483] potomek czeka na semafor (0x7b1c53743000)
*** [1483] potomek zakończył
*** [1482] wygląda, że to wszystko
```

O ile semafory kojarzone są raczej z procesami, to nic nie stoi – formalnie – na przeszkodzie aby użyć je wobec wątków. W kolejnym przykładzie użyjemy semafora nienazwanego (POSIX) celem zabezpieczenie wyłączności wykonania pewnego bloku kodu funkcji wątku. Załóżmy funkcję wątku postaci

Kod programu przedstawia się następująco - użyjemy dwóch wątków

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#define SHARE 1 //...współdzielenie semafora między procesami (true)
sem_t id;// ...identyfikator używanego semafora
int n;// ...zmienna globalna na której działać będą wątki
void* thread( void *ptr )
{
    int i = *((int *) ptr); //...z argumentu pobierzemy numer kolejny wątku
    printf( "wątek %d: start\n",i );
    sem_wait( &id ); //...wątek wykonuje P(), wyłączność
    printf( "wątek %d: krytyczna, start\n",i );
    printf( "wątek %d: n = %d\n",i,n );
    printf( "wątek %d: n++\n",i ); n++;
    printf( "wątek %d: n = %d\n",i,n );
    printf( "wątek %d: krytyczna, stop\n",i );
    sem_post( &id ); //...wątek wykonuje V(), zwolnienie
    printf( "wątek %d: stop\n",i );
    pthread_exit( 0 );
}

int main( int argc,char** argv )
{
    int i[]={1,2}; //...tablica numerów wątków
    pthread_t first,second;
    sem_init( &id,!SHARE,1 );// ...tworzymy semafor (POSIX, nienazwany)
    //...wątki startują
    pthread_create( &first, NULL,thread,(void *)(i+0) );
```

```
    pthread_create( &second, NULL,thread,(void *)(i+1) );
    /* tutaj pracują wątki */
    pthread_join( first,NULL ); pthread_join(second,NULL );
    //...wątki zakończyły działanie
    sem_destroy( &id );// ...semafor usunięty
    return 0;
}
```

Co da nam następujący output

```
$ ./threads
wątek 2: start
wątek 2: krytyczna, start
wątek 2: n = 0
wątek 2: n++
wątek 2: n = 1
wątek 2: krytyczna, stop
wątek 2: stop
wątek 1: start
wątek 1: krytyczna, start
wątek 1: n = 1
wątek 1: n++
wątek 1: n = 2
wątek 1: krytyczna, stop
wątek 1: stop
```

Wnioski

Oto cel ćwiczenia oraz wnioski dla sprawozdania na temat używania semaforów nazwanych i nienazwanych w systemie operacyjnym w kontekście różnic pomiędzy rozwiązaniami zgodnie z POSIX a UNIX System V:

Cel ćwiczenia:

1. **Różnice między POSIX a UNIX System V w kontekście semaforów:**
    - POSIX oferuje bardziej elastyczne podejście do zarządzania semaforami, umożliwiając zarówno semafory nazwane, jak i nienazwane, z różnymi możliwościami synchronizacji.
    - UNIX System V opiera się na zbiorczości semaforów, co sprawia, że jest bardziej skomplikowany w użyciu dla bardziej złożonych problemów synchronizacji.
2. **Funkcje i sposoby zarządzania semaforami:**
    - `sem_open()`, `sem_close()`, i `sem_unlink()` są używane do tworzenia, zamykania i usuwania semaforów nazwanych POSIX.
    - Semaforów nienazwanych POSIX nie są związane z systemem plików, dlatego nie wymagają operacji plikowych typu `open()`, `close()` czy `unlink()`.
    - `sem_init()` i `sem_destroy()` są stosowane do obsługi semaforów nienazwanych POSIX.
3. **Użycie semaforów w relacji do procesów i wątków:**

- Semaforów można używać zarówno w odniesieniu do procesów, jak i wątków. Funkcje obsługi semaforów w kontekście wątków są analogiczne do tych dla procesów.
- Semafor może być użyty do synchronizacji dostępu do zasobów współdzielonych, zapewniając bezpieczeństwo wykonywania kodu krytycznego.