Uniwersytet Bielsko-Bialski

LABORATORIUM

Obliczeń Równoległych i Systemów Rozproszonych

Sprawozdanie nr 7

Użycie i zarządzanie wątkami

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

Igor Gawłowicz / 59096

Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z semaforami jako elementem InterProcess Communication (IPC) w systemie UNIX oraz z ich zastosowaniem do rozwiązywania problemów związanych z synchronizacją i koordynacją między procesami.

Przebieg ćwiczenia

Kolejnym elementem funkcjonalnym InterProcess Communication (IPC) UNIX System Va także POSIX* są semafory. Stanowią one realizację najwcześniejszych pomysłów i sposobów rozwiązania problemów współużytkowania zasobów i koordynacji, który zaproponował Edsger Wybe Dijkstra.

Ponownie możemy sprawdzić aktualnie otwarte semafory za pomocą polecenia `ipcs -s`

```
$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

Każda z nich identyfikowana jest unikalnym kluczem (semkey) oraz identyfikatorem (semid), każda posiada określonego właściciela (tutaj: kmirota) i prawa dostępu (perms). Ostatnia kolumna podaje ilość semaforów w tablicy (tutaj: 1). Usunięcie takiego obiektu z pamięci jądra odbywa się za pomocą ipcrm [-S semkey -s semid].

Jądro systemowe zarządza tablicą semaforów wykorzystując strukturę danych o następującej definicji zawartej w sys/sem.h (a de facto włączanej z bits/sem.h)

```
struct semid_ds
{
    struct ipc_perm sem_perm;
    __time_t sem_otime;
    __time_t sem_ctime;
    unsigned long int sem_nsems;
};
```

Generalnie obligatoryjnymi są 4 wymienione pola struktury semid_ds, a standard UNIX System Vi POSIX, dopuszcza rozszerzania tej definicji. Uprawnienia przechowywane w - podobnie jak i dla pozostałych obiektów IPC - w strukturze ipc_perm (bits/ipc.h).

```
struct ipc_perm
{
    __key_t_key;
    __uid_t uid;
    __uid_t cuid;
    __gid_t gid;
    __gid_t cgid;
    unsigned short int mode;
```

```
    unsigned short int __seq;
};
```

Zgodnie z POSIX.1-2001 wartości dla wszystkich nowotworzonych semaforów są nieokreślone, aczkolwiek w przypadku wielu implementacji systemowych - mimo wszystko wprowadza się inicjowanie (i tak przykładowo w LINUX są inicjowane zerami 0). Do dobrej praktyki należy zakładanie iż wartość jest ta jest nieokreślona.

Warto jeszcze zauważyć, że w odniesieniu do:

- maksymalna ilość semaforów związana z danym identyfikatorem jest ograniczona predefiniowanym w linux/sem.h parametrem SEMMSL;
- wartość ta określana jest w momencie tworzenia tablicy i nie może być później zmieniana
- jeżeli odwołujemy się do istniejącej już tablicy, to wartość ta nie ma znaczenia może być zero (0).

Podając wartość klucza można skorzystać z funkcji

**key_t ftok(const char pathname, int id);*

podając pewne arbitralnie obrane: pathname ścieżką (ale istniejącą) oraz id będący liczbą całkowitą niezerową.

Przygotujemy teraz program, który utworzy zbiór semaforów o zadanym w linii komend rozmiarze. Z założenia ta tablica będzie dostępna dla procesów użytkownika

```
#include <stdio.h> //...standardowe wejście/wyjście
#include <stdlib.h> //...komunikat błędu perror ()
#include <unistd.h> //... identyfikacja procesu geppid()
#include <sys/sem.h> //... stąd deklaracje dot. semaforów
#include <sys/stat.h> //...maski uprawnień, choć można inaczej
//skoro czytamy z linii komend, to nagłówek main()
int main(int argc, char** argv)
{
    int nsems;
    int semflag;
    key_t key;
    //...zmienna dla ilość semaforów w tworzonym zbiorze
    //...zmienna dla maski tworzenia semaforów
    //...zmienna dla klucza identyfikującego zbiór

    if( argc>1 ) //...sprawdźmy na początek, czy wywołanie było poprawne
    {
        sscanf(argv[1], "%d", &nsems); //...odczytujemy ilość semaforów
        if (nsems > 0) //...sprawdzmy tak na wszelki wypadek
        {
            key = ftok( "/tmp", 'a'+'t'+'h'+'r'+'i'+'r');
            //...taki sobie komunikat diagnostyczny
            printf( "[pid=%u] tworzy zbiór %d semaforów [key=%x]\n", (unsigned)
getpid(),nsems, (unsigned) key);
            //...przygotujemy maskę tworzenia semaforów
            semflag = IPC_CREAT | S_IRUSR | S_IWUSR;
            //...i to już wszystkie czynności przygotowawcze do semget()
            if(semget(key,nsems, semflag )==-1) //...jeżeli,to kłopot
```

```
            {
                perror("\tsemget()...");
                exit(3);
            }
            /*- a dalej to już tylko diagnostyka ewentualnych błędów -*/

        }
        else
        {
            printf("\tbłędna ilość semaforów (n=%d)\n",nsems );
            exit(2);
        }

    }
    else
    {
        printf("\t%s [%s]\n%s", argv[0], "n",
        "\tn -rozmiar tworzonej tablicy semaforów\n");
        exit(1);
    }
    return 0;
}
```

I teraz możemy uruchomić nasz program podając jakiś parametr

```
$ ./setsem 5
[pid=1369] tworzy zbiór 5 semaforów [key=8a5067ec]
```

Po czym możemy sprawdzić czy semafor faktycznie powstał

```
ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x8a5067ec 0          zciwolvo   600        5
```

Ponownie jak w poprzednich z funkcji IPCs kontrola nad semaforami odbywa się za pomocą polecenia **semctl()**

Gdybyśmy chcieli po działaniu programu usunąć nasz semafor musialibyśmy zmodyfikować kod w następujący sposób

```
#include <stdio.h> //...standardowe wejście/wyjście
#include <stdlib.h> //...komunikat błędu perror ()
#include <unistd.h> //... identyfikacja procesu geppid()
#include <sys/sem.h> //... stąd deklaracje dot. semaforów
#include <sys/stat.h> //...maski uprawnień, choć można inaczej
```

```c
//skoro czytamy z linii komend, to nagłówek main()
int main(int argc, char** argv)
{
    int nsems;
    int semflag;
    int semid;
    key_t key;

    if( argc>1 ) //...sprawdźmy na początek, czy wywołanie było poprawne
    {
        sscanf(argv[1], "%d", &nsems); //...odczytujemy ilość semaforów
        if (nsems > 0) //...sprawdzmy tak na wszelki wypadek
        {
            key = ftok( "/tmp", 'a'+'t'+'h'+'r'+'i'+'r');
            //...taki sobie komunikat diagnostyczny
            printf( "[pid=%u] tworzy zbiór %d semaforów [key=%x]\n", (unsigned)
getpid(),nsems, (unsigned) key);
            //...przygotujemy maskę tworzenia semaforów
            semflag = IPC_CREAT | S_IRUSR | S_IWUSR;
            semid = semget(key, nsems, semflag);
            //...i to już wszystkie czynności przygotowawcze do semget()
            if(semid == -1) //...jeżeli,to kłopot
            {
                perror("\tsemget()...");
                exit(3);
            }
            else
            {
                if (semctl(semid, 0x0, IPC_RMID) == -1)
                {
                    perror("\tsemctl()");
                    exit(1);
                }
            }
            /*- a dalej to już tylko diagnostyka ewentualnych błędów -*/

        }
        else
        {
            printf("\tbłędna ilość semaforów (n=%d)\n",nsems );
            exit(2);
        }

    }
    else
    {
        printf("\t%s [%s]\n%s", argv[0], "n",
        "\tn -rozmiar tworzonej tablicy semaforów\n");
        exit(1);
    }
    return 0;
}
```

```
$ ./setsem 5
[pid=1421] tworzy zbiór 5 semaforów [key=8a5067ec]
zciwolvo@cloudshell:~ (glassy-courage-399021)$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

Operacje na semaforach wykonujemy za pomocą funkcji **semop()**

Utworzymy teraz krótki program w którym zdefiniujemy pojedyńczy semafor, zainicjiujemy jego wartość, a następnie odwołamy się do niego z procesu potomnego wygenerowanego za pomocą **fork()**

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/sem.h>
#include <sys/wait.h>
int main(void)
{
    int status;
    key_t semkey; //... zmienne opisujące
    int nsems, semflag, semid; //...tworzoną tablicę semaforów
    struct sembuf sems; //...tworzoną tablicę semaforów struktura dla semop()
    nsems = 1; //... tworzymy tablicę semaforów, z jednym semaforem
    semkey = ftok( "/tmp", 'k'+'m' );
    semflag = IPC_CREAT | S_IRUSR | S_IWUSR;
    semid = semget(semkey, nsems, semflag);

    sems.sem_num = 0; //... indeks do semafora, pierwszy=0 !
    sems.sem_op = 7; //... może współdzielić 7 procesów
    sems.sem flg = 0x0; //... czyli z blokadą
    semop(semid, &sems,nsems ); //... inicjujemy semafor
    //... teraz pozostaje już tylko utworzyć proces potomny
    switch( fork() )
    {
        case -1: //... obsługa błędu fork()
            printf("!.!.!... błąd fork()...!.!.!\n" );
            exit( 1 );
            break;
        case 0: //... kod dla procesu potomnego
            printf("[u] semval=%d\n", (unsigned) getpid(), semctl(semid, 0,
GETVAL) );
            exit( 0 );
            break;
        default: //...kod dla procesu nadrzędnego
            wait(&status );
            printf( "[%u] semval=%d\n", (unsigned) getpid(), semctl(semid, 0,
GETVAL) );
            semctl(semid, 0x0, IPC_RMID); //... usuwamy semafory
```

```
    }
    return 0;
}
```

```
$ ./semval
[1395] semval=7
[1394] semval=7
```

Oznacza to że program za działał i nasuwa nam się pytanie, dlaczego zarówno program macierzysty i potomek widzi nowo utworzony semafor?

Jest to dość proste ponieważ obiekty IPC, są dziedziczone w programie.

Jeśli by tak nie było obiekty IPC nie miałyby zbyt dużego użytku i byłyby troszkę zbyteczne.

Następny kod, w którym semafor będzie użyty celem ochrony sekcji krytycznej następnych procesów.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/sem.h>
union semun //...struktura potrzebna dla semctl()
{
    int value;
    struct semid_ds *stat;
    unsigned short int *array;
    struct seminfo *info;
};

int main(int argc, char ** argv)
{
    key_t key;
    int semid, flag;
    struct sembuf P={0,-1,0}; //...żądanie dostępu {numer, operacja, flaga}
    struct sembuf V={ 0,+1,0}; //...zwolnienie zasobu {numer, operacja, flaga}
    union semun control;
    int i,p,k,n;
    double x;
    if( argc<2) //...sprawdźmy, czy aby wywołanie jest poprawne
    {
        printf("%s %s\n %s\n", argv[0], "[n]", "n-krotność wykonania procesu" );
        exit(1);
    }

    sscanf(argv[1], "%d",&p ); //...jeżeli tak, to czytamy ilość powtórzeń
    key = ftok("/tmp", 'k'+'m' );
    flag = IPC_CREAT | S_IRUSR | S_IWUSR;
```

```
    semid = semget(key,1,flag); //... tworzymy tablicę semaforów (pojedynczy)
    control.value = 1; //... inicjowanie semafora (binarnego) wartością 1
    semctl(semid, 0x0, SETVAL, control);
    switch( fork() ) //...utworzenie procesu potomnego
    {
        case 1: //...obsługa błędu fork
            printf("!.!.!...fork()...!.!.!\n");
            exit(1);
            break;
        case 0: //...kod dla procesu potomnego
            printf("...[%u]...proces potomny.......start\n", (unsigned) getpid());
            fflush(stdout);
            break;
        default: //...kod dla procesu macierzystego
            printf("... [%u]...proces macierzysty...start\n", (unsigned)
getpid());
            fflush(stdout);
            break;
    }
    for(i=0;i<p;i++ )
    {
        semop(semid, &P, 1 );
        //...początek sekcji krytycznej
        printf(" [%u]...rozpoczyna wykonywanie sekcji krytycznej\n", (unsigned)
getpid());
        n = rand();
        for(k=0,x=0;k<n; k++ )
        {
            x += (double) rand()/RAND_MAX;
        }
        printf("    n=%d x=%f\n",n,x/(double)n );
        fflush(stdout);
        printf("    [%u]...kończy wykonywanie sekcji krytycznej\n", (unsigned)
getpid());
        //...koniec sekcji krytycznej
        semop(semid, &V, 1 );
    }

    semctl(semid, 0x0, IPC_RMID, control ); //...usuwamy tablicę semaforów
    return 0;
}
```

```
$ ./forks-1 2
... [1452]...proces macierzysty...start
    [1452]...rozpoczyna wykonywanie sekcji krytycznej
... [1453]...proces potomny.......start
    n=1804289383 x=0.499997
    [1452]...kończy wykonywanie sekcji krytycznej
    [1453]...rozpoczyna wykonywanie sekcji krytycznej
    n=1804289383 x=0.499997
    [1453]...kończy wykonywanie sekcji krytycznej
```

```
[1452]...rozpoczyna wykonywanie sekcji krytycznej
n=298072528 x=0.500004
[1452]...kończy wykonywanie sekcji krytycznej
[1453]...rozpoczyna wykonywanie sekcji krytycznej
```

Niezależnie od czasu trwania wykonania sekcji krytycznej, jest ona w każdym przypadku skutecznie chroniona, choć znajduje się w obrębie części wspólnej kodu procesów. Zwróćmy także uwagę na kolejność wykonywania się procesów.

Często będzie nam zależało na uzyskaniu pożądanej kolejności wykonania procesów. Można to zrobić następująco

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/sem.h>
int main( void )
{
    union semun
    {
        int value;
        struct semid_ds *stat;
        unsigned short int *array;
        struct seminfo *info;
    } control;
    key_t key;
    int flag, semid;
    struct sembuf sems;

    key = ftok("/tmp", 'k'+'m' );
    flag = IPC_CREAT | S_IRUSR | S_IWUSR;
    semid = semget(key, 1, flag);
    control.value =+1;
    semctl(semid, 0x0, SETVAL, control );
    sems.sem_num = 0;
    sems.sem_flg = 0x0;
    switch( (int) fork() )
    {
        case 1:
            perror("...fork()...\t");
            exit(1);
            break;
        case 0:
            sems.sem_op -1;
            semop ( semid, &sems, 1);
            printf("...child...\t: %u\n", (unsigned) getpid() );
            sems.sem_op =+2;
            semop( semid, &sems, 1 );
            break;
        default:
```

```
            sems.sem_op =-2;
            semop( semid, &sems, 1);
            printf("...master...\t: %u\n", (unsigned) getpid() );
            sems.sem_op =+1;
            semop( semid, &sems, 1);
            break;
    }
    semctl(semid, 0x0, IPC_RMID );
    return 0;
}
```

```
$ ./forks-2 2
...child...     : 1496
...master...    : 1495
```

Ze względu na sposób w jaki zaincjowaliśmy i wykorzystaliśmy semafor proces potomny zawsze będzie wykonany jako pierwszy.

Posługując się semaforami można wymusić pewną krotność wykonywania określonego procesu podczas gdy wykonanie innego czasowo zawieszamy.

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

int main( void )
{
    union semun
    {
        //...tę strukturę musimy zdefiniować we własnym zakresie
        int count;
        struct semid_ds *stats;
        unsigned short int *array;
        struct seminfo *infos;
    } control;
    //...struktury wykorzystywane przez semop()
    struct sembuf PO={0,-1,0 }, VO={0,1,0 },Z0={0,0,0 };
    struct sembuf P1={ 1,-1,0 },V1={ 1,1,0 },Z1={ 1,0,0 }; //...i cała reszta
    key_t key;
    pid_t pid;
    int semid, flag,nsems, step;
    key=ftok("/tmp", 'k'+'m' );
    flag=IPC_CREAT | S_IRUSR | S_IWUSR;
    nsems = 2;
    semid = semget(key, nsems, flag);
    control.count = 1;
    semctl(semid, 0, SETVAL, control);
```

```
    control.count = 5;
    semctl(semid, 1, SETVAL, control); //czyli będzie 5 cykli

    pid = fork(); //...uaktywniamy proces potomny
    for(step=0; step<control.count; step++)
    {
        if(!pid) //...to wyłacznie dla potomka
        {
            printf("[%u]...procesu potomny... start\n", (unsigned) getpid() );
            semop(semid, &PO, 1 ); //... ustawiamy semafor '0'
            printf("[%u]...krytyczna...start\n", (unsigned) getpid() );
            sleep( 1 );
            printf("[%u]...krytyczna...stop\n", (unsigned)getpid() );
            semop(semid, &VO,1 ); //...i zwalniamy semafor '0'
            //...jeszcze zwiększymy o 1 wartość semafora 1,
            // blokującego proces nadrzędny
            semop(semid, &P1,1 ); //...zwiększamy wartość o '1' dla
            printf("[%u]...procesu potomny...stop\n\n", (unsigned) getpid() );
        }
    } //...no i załóżmy, że to wszystko, co miał zrobić potomek/potomkowie
    /* Jeżeli pozostawimy ten fragment w komentarzu, to...
    if(!pid)
    {
        printf("[%u]...proces potomny zwalnia pamięć\n", (unsigned) getpid());
        exit( 0 );
    }
    ...potomek wykona i całą resztę kodu (kiedy zakończy pętlę)*/
    semop(semid, &Z1, 1); //...na tym semaforze zatrzymał się parent
    //...przechodzi go w momencie kiedy, właściwą wartość ustawi potomek
    printf("[%u]...proces nadrzędny odzyskał sterowanie\n", (unsigned) getpid());
//...na koniec usuwany semafor z pomięci
    semctl(semid, 0x0, IPC_RMID );
    return 0;
}
```

```
$ ./forks-3
[1528]...procesu potomny... start
[1528]...krytyczna...start
[1528]...krytyczna...stop
[1528]...procesu potomny...stop

[1528]...procesu potomny... start
[1528]...krytyczna...start
[1528]...krytyczna...stop
[1528]...procesu potomny...stop

[1528]...procesu potomny... start
[1528]...krytyczna...start
[1528]...krytyczna...stop
[1528]...procesu potomny...stop
```

```
[1528]...procesu potomny... start
[1528]...krytyczna...start
[1528]...krytyczna...stop
[1528]...procesu potomny...stop

[1528]...procesu potomny... start
[1528]...krytyczna...start
[1528]...krytyczna...stop
[1528]...procesu potomny...stop

[1528]...proces nadrzędny odzyskał sterowanie
[1527]...proces nadrzędny odzyskał sterowanie
```

Wnioski

1. **InterProcess Communication (IPC):**
    - Semafor to mechanizm synchronizacji IPC służący do kontrolowania dostępu do współdzielonych zasobów między procesami.
    - Pozwala on na kontrolę dostępu do sekcji krytycznej, zapobiegając współbieżnym dostępom wielu procesów do jednego zasobu.
2. **Operacje na semaforach:**
    - Tworzenie semaforów odbywa się przy użyciu funkcji `semget()`.
    - Kontrola semaforów (np. inicjowanie, odczytywanie, usuwanie) odbywa się za pomocą funkcji `semctl()`.
    - Realizacja operacji P (wait) i V (signal) odbywa się przy użyciu funkcji `semop()`.
3. **Inicjacja semaforów:**
    - Semafor może być zainicjowany wartością początkową przez funkcję `semctl()` lub automatycznie inicjowany przez system.
4. **Dziedziczenie semaforów:**
    - Procesy potomne mogą dziedziczyć semafory utworzone przez procesy macierzyste, co pozwala na ich wspólne wykorzystanie.
5. **Kontrola dostępu i sekcje krytyczne:**
    - Semafor jest użyteczny do zapewnienia bezpieczeństwa w sekcjach krytycznych poprzez blokowanie lub zwalnianie dostępu.
    - Zastosowanie semaforów pozwala na kontrolę dostępu do współdzielonych zasobów, zapobiegając wyścigom (race conditions) między procesami.
6. **Synchronizacja procesów:**
    - Semafor może być wykorzystany do synchronizacji procesów, na przykład do wymuszenia kolejności ich wykonywania.

- Możliwe jest użycie semaforów do sterowania ilością powtórzeń lub zabezpieczenia sekcji krytycznych przed jednoczesnym dostępem wielu procesów.