САМОСТІЙНА РОБОТА № 4

Назва роботи: Синхронізація в ОС UNIX за допомогою семафорів.

Мета роботи: Засвоїти механізм синхронізації процесів через семафори.

1. Загальні відомості

1.1. System V Release 4 (SVR4) UNIX визначає такі InterProcess Communication (IPC):іменовані канали (NPIPES) черги повідомлень (MSGQUEUES) пам'ять, що розділяється (SHMEM)семафори (SEMAPHORES). Усі ці види комунікацій одночасно є засобами синхронізації процесів. Ця лабораторна робота присвячується синхронізації процесів через семафори.

1.2. Значення семафора - це ціле число в діапазоні від 0 до 32767. Оскільки в багатьох програмах потрібно більше одного семафора, ОС UNIX надає можливість створювати множини семафорів. Їхній максимальний розмір обмежений системним параметром `SEMMSL`.

1.3. Перед тим як використовувати семафори (виконувати операції чи керуючі дії), потрібно створити множину семафорів (`semget(2)`) з унікальним ідентифікатором і асоційованою структурою даних. Унікальний ідентифікатор називається ідентифікатором множини семафорів (`semid`); він використовується для звертань до множини і структури даних.

З погляду реалізації множина семафорів являє собою масив структур. Кожна структура відповідає семафору і визначається в такий спосіб:

```
#include <sys/sem.h>

struct sem
{
    //Значення семафора
    ushort semval;
    //Ідентифікатор процесу, що виконував останню операцію
    short  sempid;
    //Число процесів, що очікують збільшення значення семафора
    ushort semncnt;
    //Число процесів, що очікують обнулення значення семафора
    ushort semzcnt;
};
```

З кожним ідентифікатором множини семафорів асоційована структура даних, що містить наступну інформацію:

```
#include <sys/sem.h>

struct semid_ds
{
    // Структура прав на виконання операцій
    struct ipc_perm sem_perm;
    // Покажчик на перший семафор у множині
    struct sem *sem_base;
    // Кількість семафорів у множині
    ushort sem_nsems;
    // Час останньої операції
    time_t sem_otime;
    // Час останньої зміни
```

```
    time_t sem_ctime;
};
```

1.4. Процес, що виконав системний виклик semget(2), стає власником/творцем множини семафорів. Він визначає, скільки буде семафорів у множині; крім того, він специфікує початкові права на виконання операцій над множиною для всіх процесів, включаючи себе. Даний процес може уступити право власності або змінити права на операції за допомогою системного виклику semctl(2), призначеного для керування семафорами. Над кожним семафором, що належить множині, за допомогою системного виклику semop(2) можна виконати одну з трьох операцій:

- Збільшити значення.
- Зменшити значення.
- Дочекатися обнуління.

– 1.5. Операції можуть використовувати прапорці. Прапорець SEM_UNDO означає, що операція виконується в перевірочному режимі, тобто потрібно тільки довідатися, чи можна успішно виконати дану операцію.

– 1.6. Системний виклик semop(2) оперує не з окремим семафором, а з множиною семафорів, застосовуючи до нього "масив операцій". Операційна

– система, зрозуміло, виконує операції з масиву по черзі, причому порядок не обумовлюється. Якщо чергова операція не може бути виконана, то ефект попередніх операцій анулюється.

2. Синтаксис та призначення системних викликів

2.1. Системний виклик `semget`.

```
int semget (key, nsems, semflg)
            key_t key;
            int nsems;
            int semflg;
```

Системний виклик `semget` повертає ідентифікатор множини семафорів, який асоційований із ключем `key`.

Ідентифікатор і асоційовані з ним структура даних і множина з `nsems` семафорів створюються для ключа `key` у наступних випадках:

- Значення аргументу `key` дорівнює `IPC_PRIVATE`.
- Ключ `key` ще не має асоційованого з ним ідентифікатора множини семафорів і вираз (`semflg & IPC_CREAT`) істинний.

Використання параметру `semflg` таке саме, як і у системному виклику `msgget(2)` (лабораторна робота №4).

При успішному завершенні системного виклику повертається ідентифікатор множини семафорів. У випадку помилки повертається -1, а змінній `errno` присвоюється код помилки.

2.2. Системний виклик `semop`.

```
int semop (semid, sops, nsops);
           int semid;
           struct sembuf** sops;
           unsigned nsops;
```

Системний виклик `semop` використовується для виконання набору операцій над множиною семафорів, який асоційований з ідентифікатором `semid`. Аргумент `sops` (масив структур) визначає, над якими семафорами будуть виконуватися операції і які саме. Структура, що описує операцію над одним семафором, визначається в такий спосіб:

```
#include <sys/sem.h>

struct sembuf
{
    // Номер семафора
    short sem_num;
    // Операція над семафором
    short sem_op;
    // Прапорці операції
    short sem_flg;
};
```

Номер семафора задає конкретний семафор у множині, над яким повинна бути виконана операція.

Виконувана операція визначається в такий спосіб:

– Позитивне значення поля `sem_op` наказує збільшити значення семафора на величину `sem_op`.
– Негативне значення поля `sem_op` наказує зменшити значення семафора на абсолютну величину `sem_op`. Операція не може бути успішно виконана, якщо в результаті вийде негативне число.
– Нульове значення поля `sem_op` наказує порівняти значення семафора з нулем. Операція не може бути успішно виконана, якщо значення семафора відмінне від нуля.

Допустимі значення прапорців операцій (поле `sem_flg`):

– IPC_NOWAIT

Якщо яка-небудь операція, для якої заданий прапорець `IPC_NOWAIT`, не може бути успішно виконана, системний виклик завершується невдачею, причому значення жодного із семафорів не буде змінено.

– SEM_UNDO

Даний прапорець задає перевірочний режим виконання операції; він наказує анулювати її результат навіть у випадку успішного завершення системного виклику `semop(2)`. Іншими словами, блокування всіх операцій (у тому числі і тих, для яких заданий прапор `SEM_UNDO`) виконується звичайним чином, але коли нарешті всі операції можуть бути успішно виконані, операції з прапором `SEM_UNDO` ігноруються.

Аргумент `nsops` специфікує кількість структур у масиві. Максимально припустимий розмір масиву визначається системним параметром `SEMOPM`, тобто в кожному системному виклику `semop(2)` можна виконати не більш `SEMOPM` операцій.

2.3. Системний виклик `semctl`.

```
int semctl (semid, semnum, cmd, arg)
            int semid, cmd;
            int semnum;
            union semun
            {
                int val;
                struct semid_ds *buf;
                ushort *array;
            } arg;
```

Системний виклик `semctl` дозволяє виконувати операції керування семафорами. Семафори задаються аргументами `semid` і `semnum`. Операція визначається значенням аргументу `cmd`. Розглянемо деякі значення аргументу `cmd`:

GETVAL	Одержати значення семафора `semval`.
SETVAL	Встановити значення семафора `semval` рівним `arg.val`.
GETPI	Одержати значення `sempid`.

GETNCNT	Одержати значення semncnt.
GETZCNT	Одержати значення semzcnt.
GETALL	Прочитати значення семафорів у масив, на який вказує arg.array.
SETALL	Встановити значення семафорів рівними значенням елементів масиву, на який вказує arg.array.
IPC_RMID	Видалити із системи ідентифікатор semid, ліквідувати множину семафорів і асоційовану з ними структуру даних.

3. Приклад програми

3.1. Приклад батьківської програми (parent.c):

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#define SLEEP_TIME    5
#define ITERATION_NUM 7
void main(void)
{
   int i;
   int errFlag;
   int status;
   struct sembuf sem_command;
   // two rwxrwxrwx semaphore arrays, each of one semaphore
   int sem1Id = semget(IPC_PRIVATE, 1, 0xfff);
   if(sem1Id == -1)
   {
      fprintf(stderr, "Can't create first semaphore.\n");
      exit(1);
   }
   int sem2Id = semget(IPC_PRIVATE, 1, 0xfff);
   if(sem2Id == -1)
   {
      fprintf(stderr, "Can't create second semaphore.\n");
      exit(1);
   }
   sem_command.sem_num = 0;//selects first semaphore in array
   sem_command.sem_op = 2; //increments semval by 2
   sem_command.sem_flg =SEM_UNDO;//undoes when the process exits
   semop(sem1Id, &sem_command, 1);//activates one semaphore
   sem_command.sem_op = 1;//increments semval by 1
   semop(sem2Id, &sem_command, 1);//activates one semaphore
   char arg1[5];
   char arg2[5];
   int mPid;
   int pPid;
   int ecPid;
   int icPid = fork();
   switch(icPid)
```

```
{
   case -1:
      fprintf(stderr, "Can't fork for internal child.\n");
      exit(1);
   case 0:
      ecPid = fork();
      switch(ecPid)
      {
         case -1:
        fprintf(stderr, "Can't fork for external child.\n");
            exit(1);
         case 0:
            sprintf(arg1, "%d", sem1Id);
            sprintf(arg2, "%d", sem2Id);
            errFlag = execl("./echild","echild",arg1,arg2,0);
            if(errFlag == -1)
            {
      fprintf(stderr, "Can't execute the external child.\n");
               exit(1);
            }
         default:
            mPid = getpid();
            pPid = getppid();
            printf("IChild: My PID = %d, "
                           "My parent's PID = %d, "
                           "My child's PID = %d.\n",
                            mPid, pPid, ecPid);
            sem_command.sem_op = -1;//decrements semval by 1
            semop(sem1Id, &sem_command, 1);
            sem_command.sem_op = 0;//waits for semval == 0
            semop(sem2Id, &sem_command, 1);
            for(i = 0; i < ITERATION_NUM; i++)
            {
               printf("INTERNAL CHILD now is working.\n");
               sleep(SLEEP_TIME);
               //for(j=0;j<1000000;j++);
            }
            wait(&status);
            printf("\nIChild: EChild exit code is %d.\n",
                                 (status & 0xff00) >> 8);
            exit(1);
      }
   default:
      mPid = getpid();
      pPid = getppid();
      printf("Parent: My PID = %d, "
                     "My parent's PID = %d, "
                     "My child's PID = %d.\n",
                      mPid, pPid, icPid);
      sem_command.sem_op = 0;    // waits for semval==0
      semop(sem1Id, &sem_command, 1);
      printf("Parent:Press the <Enter>key to continue...\n");
      getchar();
```

```
         sem_command.sem_op = -1;  // decrements semval by 1
         semop(sem2Id, &sem_command, 1);
         for(i = 0; i < ITERATION_NUM; i++)
         {
            printf("PARENT now is working.\n");
            sleep(SLEEP_TIME);
            //for(j=0;j<1000000;j++);
         }
         wait(&status);
         printf("Parent: IChild exit code is %d.\n",(status &
0xff00) >> 8);
         exit(0);
   }
}
```

3.2. Приклад синівської програми (child.c):

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#define SLEEP_TIME    5
#define ITERATION_NUM 7
void main(int argc, char *argv[])
{
   int i;
   struct sembuf sem_command;
   if(argc != 3)
   {
      fprintf(stderr, "Missing arguments were detected.\n");
      exit(1);
   }
   sem_command.sem_num = 0;//selects first semaphore in array
   sem_command.sem_flg =SEM_UNDO;//undoes when the process exits
   int sem1Id = atoi(argv[1]);
   int sem2Id = atoi(argv[2]);
   int mPid = getpid();
   int pPid = getppid();
   printf("EChild: My PID = %d, My parent's PID = %d.\n", mPid,
pPid);
   sem_command.sem_op = -1;   // decrements semval by 1
   semop(sem1Id, &sem_command, 1);
   sem_command.sem_op = 0;   // waits for semval == 0
   semop(sem2Id, &sem_command, 1);
   for(i = 0; i < ITERATION_NUM; i++)
   {
      printf("EXTERNAL CHILD now is working.\n");
      sleep(SLEEP_TIME);
      //for(j=0;j<1000000;j++);
   }
   exit(0);
}
```

4. Варіанти завдань

n%2 + 1	**Опис завдання**
1	Синхронізація батьківського та синівського процесу.
2	Синхронізація двох синівських процесів.
де n – порядковий номер у журналі	

5. Зміст звіту

- Титульний лист;
- Завдання;
- Алгоритм рішення завдання;
- Код програми;
- Екранна форма з результатами роботи програми;
- Висновки.