acm.mipt.ru

олимпиады по программированию на Физтехе

Семафоры

Для разделения доступа к ресурсам могут использоваться СЕМАФОРЫ.

Семафор в UNIX – системное устройство. Все операции, связанные с изменением сстояния семафора – АТОМАРНЫЕ. То есть, выполнение программы не может быть прервано и отложено пока операция с семафором не будет полностью завершена.

Каждый семафор имеет некоторое значение, которое при создании семафора сразу утанавливается в 0.

Над семафором можно выполнять следующие операции:

1. Увеличить значение семафора
2. Дождаться когда значение семафора станет равным 0
3. Дождаться когда значение семафора станет равным числу N

При этом понятно, что первая операция безусловная, а две последние требуют проверки значения семафора.

Разделяемая память

Разделяемая память – системный ресурс, который служит для обмена информацией между процессами. Разделяемая память – это часть оперативной памяти, выделяемой в качестве ресурса, одновременно доступного нескольким процессам.

В процессе ее использования возможны конфликты доступа: попытка считать частично записанную информацию, попытка перезаписать непосредственно сейчас считываемую информацию.

Для избежания таких конфликтов, при работе с разделяемой памятью, используются семафоры.

Пример программ на языке С для работы с разделяемой памятью и семафорами:

заголовочный файл **sem.h**

```
#define PERM 0666
// структура для записи в разделяемую память
typedef struct mem_msg{
  int segmet;
  char buf[100];
} Message;

// операции над семафором 1
// для ожидания партнера
// запирающее значение этого семафора 0
// разрешающее - 1

// дождаться когда семафор станет 1 и
// установить его в 0
static struct sembuf proc_wait[1] = { 1, -1, 0 };
// установить семафор в 1
// операция без условия
static struct sembuf proc_start[1] = { 1, 1, 0 };

// операции над семафором 0
// для блокировки ресурса
// запирающее значение этого семафора 1
// разрешающее - 0

// описание двух операций
// выполняются обе
static struct sembuf lock[2] = {
  0, 0, 0,  // дождаться когда семафор станет 0
  0, 1, 0  // увеличить значение на 1
};

// описание операции (выполняется одна)
// дождаться когда семафор станет 1 и
// установить его в 0
 static struct sembuf unlock[1] = { 0, -1, 0 };
```

Сервер

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include <string.h>

#include "sem.h"

int main(){

// указатель на тип Message
// в дальнейшем под сообщения будет
// выделена разделяемая память
  Message *msgptr;
// ключ для создания массива семафоров и
// разделяемой памяти
  key_t key;
// дескриптор разделяемой памяти
```

Клиент

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include <string.h>

#include "sem.h"

int main(){

// указатель на тип Message
// в дальнейшем под сообщения будет
// выделена разделяемая память
  Message *msgptr;
// ключ для создания массива семафоров и
// разделяемой памяти
  key_t key;
// дескриптор разделяемой памяти
```

```
  int shmid;
// дескриптор массива семафоров
  int semid;

  int lng, n;
  char stop='A';
// создаем ключ
  if ( ( key = ftok("serv",'A' ) ) < 0){
   printf("Can't get key\n");
   exit(1);
  }

// сервер создает разделяемую память
// функция shmget
  if( ( shmid = shmget ( key, sizeof(Message), PERM|IPC_CREAT ) ) < 0 ){
    printf("Can't create shared mem\n");
    exit(1);
  }
// получаем указатель на область разделяемой памяти
// (функция shmat)
// и сразу преобразуем указатель к типу Message
  if ( ( msgptr = ( Message* ) shmat( shmid, 0, 0 ) ) < 0 ){
    perror(":(");
  }

// сервер создает массив семафоров (2 семафора)
// функция semget
// значение семафора при создании - 0
  if ( ( semid = semget( key, 2 , PERM|IPC_CREAT ) ) < 0){
    perror(":(");
    exit(1);
  }

// ждет когда появится клиент и
// поставит семафор в 1
// функция semop
  if ( semop( semid, &proc_wait[0], 1 ) < 0 ){
    printf(":(\n");
    exit(1);
  }
// если исполняется эта часть программы, то
// клиент уже работает.
// пытаемся заблокировать ресурсэ
// если значение семафора 0, то ставим его в 1
// ( выполняется две операции семафора)
  if ( semop( semid, &lock[0], 2 ) < 0 ){
    printf(":(\n");
    exit(1);
  }
// "Захваченный" ресурс - это разделяемая память
// никто не мешает ей пользоваться
// печатаем содержимое поля buf
// из разделяемой памяти
  printf("%s\n", msgptr->buf );

// освобождаем ресурс
// дожидаемся когда значение семафора будет 1
// (а в 1 мы его сами поставили прошлой операцией)
// и устанавливаем его в 0
  if ( semop( semid, &unlock[0], 1 ) < 0 ){
    printf(":(\n");
    exit(1);
 }

// "отстыкуем" сегмент разделяемой памяти
  if ( shmdt( msgptr ) < 0 ){
    printf(":(");
  }


 return 0;

}
```

```
  int shmid;
// дескриптор массива семафоров
  int semid;
  char buf[100];
  int lng, n;
  char stop='A';

// создаем ключ
  if ( ( key = ftok("serv",'A' ) ) < 0){
   printf("Can't get key\n");
   exit(1);
  }

// сервер уже создал разделяемую память
// получаем к ней доступ
  if( ( shmid =shmget( key, sizeof(Message), 0 ) )
    printf("Can't create shared mem\n");
    exit(1);
  }

// получаем указатель на область разделяемой памяти
// (функция shmat)
// и сразу преобразуем указатель к типу Message
  if ( ( msgptr = ( Message* ) shmat( shmid, 0, 0 )
    perror(":(");
  }

// сервер уже созда массив семафоров
// поучаем его дескриптор
  if ( ( semid = semget( key, 2, PERM ) ) < 0 ){
    perror(":(");
    exit(1);
  }
// "Захватываем" ресурс:
// дожидаемся пока семафор станет 0 и выставляемего
  if ( semop( semid, &lock[0], 2) < 0 ){
    printf(":(\n");
    exit(1);
  }
// Сообщаем сереверу, что клиент работает
// семафор номер 1 устанавливаем в 1.
  if ( semop( semid, &proc_start[0], 1 ) < 0 ){
    printf(":(\n");
    exit(1);
  }

// можем использовать разделяемую память
// получим строку и запишем ее в разделяемуюю памят
  scanf( "%s", buf );

  sprintf( msgptr->buf, "%s", buf );

// освобождаем ресурс
// семафор ставим в 1

  if ( semop ( semid, &unlock[0], 1 ) < 0 ){
    printf(":(\n");
    exit(1);
  }

// заболкируем ресурс, чтобы спокойно их удалить
  if ( semop ( semid, &lock[0], 2 ) < 0 ){
    printf(":(\n");
    exit(1);
  }
// отсоединяем разделяемую память
  if ( shmdt( msgptr ) < 0){
    printf(":(");
  }
// удаляем разделяемую память
  if ( shmctl( shmid, IPC_RMID, 0 ) < 0 ){
    printf(":(\n");
  }
// удаляем массив семафоров
  if ( semctl( semid, 2, IPC_RMID ) < 0 ){
   printf(":(\n");
 }

  return 0;
}
```

Рассмотрим простой пример игры в **крестики-нолики.**

Имеем два игрока, которые используют одно и тоже поле. Необходимо обеспечить очередность ходов игроков.

Мы не знаем какой из игроков первый предложит игру.

Самое первое действие, которое необходимо сделатть при организации игры – это обеспечить чтобы первый игрок дождался второго и не начинал игру один.

Для этого нужно использовать семафор номер 1

Далее доступ к полю (будем блокировать поле целиком) будем обеспечивать семафором номер 0.

Заголовочный файл для поля с семафорами *tictacSM.h*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include <iostream>
```

```
#include <cstdlib>
#include <fstream>

#define PERM 0666

using namespace std;


class TicTac{
// ключ для создания семафоров и разделяемой памяти
  key_t key;
// описание операций с семафором (ожидание партнера)
// процесс - в ожидание

  struct sembuf waitYou[1];
// первое поле - номер семафора (1)
// второе - операция с семафором. Здесь:
// дождаться когда занчение семафора будет равно 1
// ={ 1,-1,0};

// описание операций с семафором (партнер пришел)
// партнер может действовать

  struct sembuf ImHere[1];
// первое поле - номер семафора (1)
// второе - операция с семафором. Здесь:
// увеличить значение семафора на 1
//={ 1,1,0};

// описание операций с семафором (записает поле)
// используется 2 операции
  struct sembuf myTurn[2];
//  Значение семафора 0 - ресурс свободен, 1 - занят
// Первая операция - дождаться когда семафор будет 0
// вторая операция - ставим семафор в 1, то есть запираем ресурс
// = {
//    0,0,0,
//    0,1,0 };

// описание операций с семафором (записает поле)
// используется 1 операция
struct sembuf canWork[1];
//  Дождаться когда семафор будет 1 и обнулить его
// = { 0,-1,0 };

  int lng;
// очередность определяется по тому кто создал очередь семафоров
  int range;
  char stop;
  int set;
// дескрипторы для семафоров и разделяемой памяти
  int semid,shmid;
// указатель на разделяемую память
  int *msgptr;
  string s;
  int len; // размер поля

public:
// Для ключа нужны имя файла и символ
   TicTac(string, char, int);
// необходимо удалять семафоры после работы
   ~TicTac();
   int* operator[](int);
// запирание ресурса
   void myWork();
// освобождение ресурса
   void youWork();
// получить очередность
   int getRange();
// void myTurn();
//  печать поля
   void print();
};
```

Рассмотрим реализацию некоторых функций класса:

```
#include "tictacSM.h"

TicTac::TicTac(string s, char c, int n){

   len = n; // размер поля
// получим ключ
   if ((key=ftok(s.c_str(),c))<0){
      printf("Can't get key\n");
      exit(1);
   }
// Установим операции для семафоров:

// Захватить ресурс
// Эти операции будут выполняться сразу обе
   myTurn[0].sem_num = 0; // для семафора номер 0
   myTurn[0].sem_op = 0; //  дождаться 0
   myTurn[0].sem_flg = 0;

   myTurn[1].sem_num = 0; // для семафора номер 0
   myTurn[1].sem_op = 1; // увеличить значение на 1
   myTurn[1].sem_flg = 0;

// Освободить ресурс
   canWork[0].sem_num = 0; // для семафора номер 0
   canWork[0].sem_op = -1; //  дождаться 1 и поставить значение семафора в 0
   canWork[0].sem_flg = 0;
```

```
// Ожидание партнера
   waitYou[0].sem_num = 1; // для семафора номер 1
   waitYou[0].sem_op = -1; // дождаться 1 и поставить значение семафора в 0
   waitYou[0].sem_flg = 0;

// Уведомление "Я пришел"
   ImHere[0].sem_num = 1; // для семафора номер 1
   ImHere[0].sem_op = 1; // увеличить значение на 1
   ImHere[0].sem_flg = 0;

// Создать разделяемую память
   shmid = shmget(key, sizeof(int)*len*len, PERM|IPC_CREAT);
   switch (errno){
      case EEXIST:{
                    if((shmid = shmget(key, sizeof(int)*len*len, 0))<0){
                   printf("Can't create shared mem\n");
                   exit(1);
                }

                   break;
                 }
      case 0:    {

                   break;
                 }
      default:   {
                   printf("Can't create shared mem\n");
                   exit(1);
                 }

   }
//  получить указатель на разделяемую паамять
   if ((msgptr = (int*)shmat(shmid, 0, 0))<0){
      perror(":(");
      exit(1);
   }
//  создать семафоры (2 семафора)
   semid = semget(key,2,PERM|IPC_CREAT|IPC_EXCL);
   switch (errno){
      case EEXIST:{
                    if ((semid = semget(key, 2, PERM))<0){
                        perror(":(");
                         exit(1);
                    }
                   cout<<"Семафор уже есть\n";
                    range = 1;
                    break;
                 }
      case 0:    {
                   range = 0;
                   break;
                 }
      default:   {
                   printf("Can't create sema\n");
                   exit(1);
                 }

   }
   set = 0;
// получить свою очередь
   if(range == 0){
    cout<<"Ваш знак: 0\n";
    cout<<"Ставлю семафор на встречу в -1\n";

// semop - функция операции с семафорами
// тот, кто создал очередь семафоров ждет партнера
     if (semop(semid, &waitYou[0], 1)<0){
         printf(":(\n");
         exit(1);
     }
   }

   if(range == 1 ){

     cout<<"Ваш знак: X\n";
     cout<<"Чищу память\n";
     bzero(msgptr, sizeof(int)*len*len);
     cout<<"Я пришел\n";
// Уведомление "Я пришел"
     if (semop(semid, &ImHere[0], 1)<0){
      printf(":(\n");
      exit(1);
     }

   }

};
// Получить свою очередь
int TicTac::getRange(){
  return range;
};

// Удаление семафорров и разделяемой памяти
TicTac::~TicTac(){
   if (shmctl(shmid,IPC_RMID,0)<0){
      printf(":(\n");
   }

   if (semctl(semid,2,IPC_RMID)<0){
     printf(":(\n");
   }
};

void TicTac::print(){
```

```
  for(int j = 0;j<3;j++)
     {
      for(int i = 0;i<3;i++)
//  разделяемая память может быть использована как обычный массив
         cout << msgptr[j*3 + i]<<" ";

      cout<<endl;
     }
};

int* TicTac::operator[](int n){

    return msgptr + (n)*len;
};
// Запираем ресурс
void TicTac::myWork(){
   cout<<"Запираем ресурс\n";
// указываем, что выполняться будут сразу 2 операции (атомарно)
    if (semop( semid, &myTurn[0], 2)<0){
       printf(":(\n");
       exit(1);
    }
};

// освобождаем ресурс
void TicTac::youWork(){

    cout<<"Освобождаем ресурс\n";
    if (semop(semid, &canWork[0], 1)<0){
        printf(":(\n");
        exit(1);
   }

};
```

Пример использования класса **Tictac.**

```
#include "tictacSM.h"

int main(){
  TicTac pole("tic",'a',3);
  string s;
  int range = pole.getRange();

  int sgn = (range==1) ? 1 : -1;
  int x, y;
// обеспечиваем очередность 3 ходов с каждой стороны
  for(int k = 0; k < 3; k++){
      pole.myWork();
      pole.print();
      cout << "Ваш ход:";
      cin >> y >> x;
      pole[ y - 1 ][ x - 1 ] = sgn;
      cout << "ok\n";
      pole.print();
      pole.youWork();
  }
  return 0;
 }
```

Задачи

Задача 1.

Для класса **Tictac** все функции кроме конструктора, деструктора, функции **print()** и функции **myTurn()** перенести в закрытую область класса. Реализовать функцию **myTurn().**

Задача 2.

В саду созрели яблоки. Для их сбора приглашены N рабочих. Каждый рабочий собирает N_i количество яблок за один раз и складывает их в корзину. Все яблоки складываются в ОБЩУЮ корзину. Первый пришедший рабочий приносит эту корзину.

Кроме рабочих в саду летают K ворон. Каждая ворона может за один съесть K_i. Если яблок в корзине меньше, она съедает все. Если яблок в корзине нет, или рабочие не приступили к уборке, вороно ничего не ест.

Нужно написать классы **Worker**, **Crow** для моделирования количества яблок в корзине с учетом покражи, а также программу **model**, которая порождает N работников и K ворон как дочерние процессы.

Каждая программа, обращаясь к корзине записывает информацию в свой log-файл: сколько положил/съел и время.

model каждую реальную секунду выводит состояние корзины на экран. По истечении времени **model** убивает все процессы (см. Сигналы самостоятельно) и удаляет память и семафоры.

Пример интерфейсов классов для работников и ворон:

```
// Ворона
class Crow{
   int apple;// украденные яблоки за 1 час
   int *basket;// указатель на корзину с яблоками
public:
// конструктор. Вычисляется случайное количество яблок
    Crow();
// Деструктор. Указатель на корзину должен стать 0
    ~Crow();
// получит указатель на  корзину
    void getBasket(int *pbasket);
// возвращает количество укараденых яблок
    int steal();

};
//Все функции класса Crow нужно написать и отладить.

// Работник
```

```
class Worker{
  int apple; // количество яблок, которые он собирает в час
  int *basket;// указатель на корзину с яблоками
public:
// Конструктор по умолчанию. Мы не можем указывать
// сколько яблок он  соберет. Будет случайно
     Worker();

// Деструктор. Указатель на корзину должен стать 0
   ~Worker();
// возвращает количество яблок, которые собрал работник
     int job();
    // получит указатель на  корзину
   void getBasket(int *pbasket);
};
```

-- TatyanaOvsyannikova2011 - 01 Nov 2017