Семафоры.

Зыкова А.Б

4 июня 2016 г.

1 Определение

Семафор - это объект, ограничивающий количество потоков, которые могут войти в заданный участок кода. Семафоры используются для синхронизации и защиты передачи данных через разделяемую память, а также для синхронизации работы процессов и потоков. Семафор представляет собой объект с целочисленным значением , которое мы можем манипулировать с двумя подпрограммами; в стандарте POSIX, эти процедуры являются sem_wait () и sem_post () Семафоры - это то, с помощью чего можно построить

- Легковесные мьютексы
- Легковесные условные переменные
- Легковесные read-write блокировки
- Примитив для решения проблемы обедающих философов
- Легковесный семафор

Если говорить простым языком, то потоки, которые желают завладеть ресурсом, можно представить как большую очередь в галерею, а семафор - охранник на входе. Он позволяет пройти внутрь только когда ему дают соответствующее указание.

Каждый поток сам решает когда встать в эту очередь. По сути, когда поток вызывает метод wait, он становится в очередь.

Вышибала на входе, т.е. семафор, должен уметь делать только одну операцию - V. При вызове этого метода семафор «отпускает» из очереди один из ожидающих потоков. (Совсем не обязательно это будет тот же поток, который вызвал wait раньше других.)

Но зачем нужны семафоры, громосткий, неуклюжий аналог мьютекса, если мы говорим о программной реализации. И какие задачи он должен решать.

Все дело, как и всегда, в истории создания. Эдсгер Вибе Дейкстра, нидерландский учёный, во второй половине 1950-х годов в поисках путей оптимизации разводки плат разработал алгоритм поиска кратчайшего пути на графе, ставший известным ныне как «алгоритм Дейкстры». Алгоритм на графах находящий кратчайшие пути от одной из вершин графа до всех остальных, стал только началом. Вслед за ним, одна из наиболее известных проблем параллельности была сформулирована и решена путем Дейкстра, эта задача известна как «Проблема обедающих философов».

2 Постановка задачи

Пять безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов.

Каждый философ может либо есть, либо размышлять. Приём пищи не ограничен количеством оставшихся спагетти — подразумевается бесконечный запас. Тем не менее, философ может есть только тогда, когда держит две вилки — взятую справа и слева (альтернативная формулировка проблемы подразумевает миски с рисом и палочки для еды вместо тарелок со спагетти и вилок).

Каждый философ может взять ближайшую вилку (если она доступна), или положить — если он уже держит её. Взятие каждой вилки и возвращение её на стол являются раздельными действиями, которые должны выполняться одно за другим.

Суть проблемы заключается в том, чтобы разработать модель поведения (параллельный алгоритм), при котором ни один из философов не будет голодать, то есть будет вечно чередовать приём пищи и размышления. Задача сформулирована таким образом, чтобы иллюстрировать проблему избежания взаимной блокировки (англ. deadlock) — состояния системы, при котором прогресс невозможен. И решена она была только после того, как Дейкстра ввел понятие "Семафор". Инициализация семафора (задать начальное значение счётчика):

```
init(n):
    счётчик := n
```

Захват семафора (ждать пока счётчик станет больше 0, после этого уменьшить счётчик на единицу):

```
enter():
    счётчик := счётчик - 1
```

Освобождение семафора (увеличить счётчик на единицу):

```
leave():
    счётчик := счётчик + 1
```

Предположим, что есть такой участок кода:

```
semaphore.init(5);
// .....
// .....
void DoSomething()
{
    semaphore.enter();
    // .......
    semaphore.leave();
}
```

Тогда не более пяти потоков могут одновременно выполнять функцию DoSomething(). На вернемся к задаче обедающий философов.

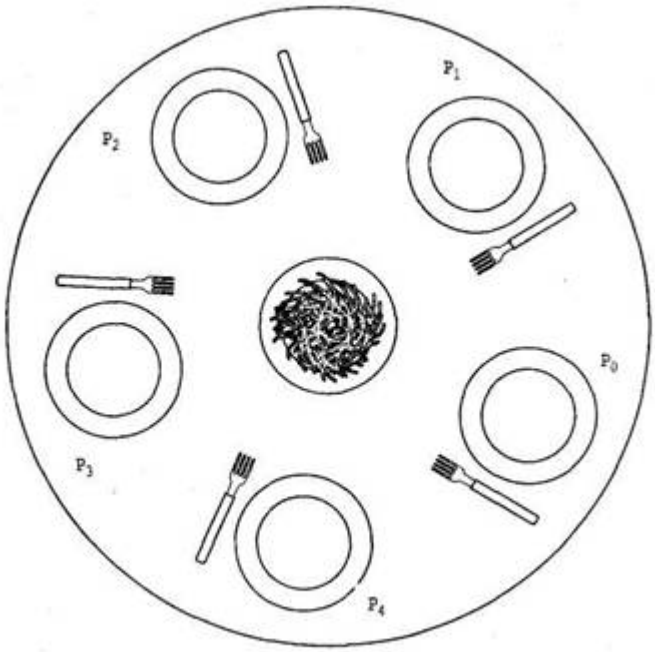

Рис. 6.9. Обеденный стол философов

Эта задача Дейкстры хорошо иллюстрирует проблемы взаимоблокировок и голодания. Кроме того, при решении данной задачи приходится сталкиваться со многими трудностями в организации параллельных вычислений. Задача об обедающих философах может рассматриваться как типичная задача, возникающая в многопоточных приложениях при работе с совместно используемыми ресурсами и, соответственно, может выступать в качестве тестовой при разработке новых подходов к проблеме синхронизации. Ее реализация на языке C++

```
#include "stdafx.h"
#ifndef FUNCT_THREAD_H
#define FUNCT_THREAD_H
#include <iostream>
#include <semaphore.h>
#include <cstdlib>
using namespace std;
sem_t *sem_fork;
sem_t servant_sem;
// класс описывающий философа
class Philosopher
{
public:
```

```
char * name_str; //имя
int left_fork, right_fork; // номера вилок
int eat_time, think_time;// время еды и размышления
void *life_philosopher(void *);
void philosopher_think();
void get_forks();
void put_forks();
void philosopher_eat();
Philosopher(const int&,const int&, char* );
static void *c_life_philosopher(void *arg)
    {
        Philosopher *f = static_cast<Philosopher *>(arg);
        f->life_philosopher(arg);
    };

};
// конструктор класса
Philosopher::Philosopher(const int& a,const int& b, char* s)
{
    left_fork=a;
    right_fork=b;
    name_str=s;

}
// захват вилок
void Philosopher::get_forks()
{
    sem_wait (&sem_fork [left_fork - 1]);
    sem_wait (&sem_fork [right_fork - 1]);
    cout<<name_str<<":Я захватил вилки"<<left_fork<<"и"<<right_fork<<endl;
}
// освобождение вилок
void Philosopher::put_forks()
{
    sem_post (&sem_fork [left_fork - 1]);
    sem_post (&sem_fork [right_fork - 1]);
    cout<<name_str<<":Я отдал вилки"<<left_fork<<"и"<<right_fork<<endl;
}
// философ ест
void Philosopher::philosopher_eat()
{
```

```
    srand ( time(NULL) );
    eat_time = rand () % 20 + 1;
    cout<<name_str<<":Я кушаю"<<endl;
    sleep(eat_time);
}
// философ размышляет
void Philosopher::philosopher_think()
{
    srand ( time(NULL) );
    think_time = rand () % 20 + 1;
    cout<<name_str<<":Я размышляю"<<endl;
    sleep(think_time);
}
// жизненый цмкл философа
void *Philosopher::life_philosopher(void *)
{
    sem_wait(&servant_sem);
    cout<<name_str<<":Я вошел в столовую"<<endl;
    philosopher_think();
    get_forks();
    philosopher_eat();
    put_forks();
    sem_post(&servant_sem);
    cout<<name_str<<":Я вышел из столовой"<<endl;



}

#endif // FUNCT_THREAD_H
```

Каждый философ, садясь за стол, сначала берет левую вилку, а затем правую. После того как философ пообедает, использованные им вилки заменяются. Увы, такое решение может привести к взаимоблокировке, если философы, одновременно проголодавшись, все вместе сядут за стол и одновременно возьмут лежащие слева вилки. В этой неприятной ситуации им придется голодать.

```
 // philos.cpp: главный файл проекта.
#include "stdafx.h"
#include "funct_thread.h"
```

```
#include <iostream>
#include <pthread.h>
#include <fstream>
#include <string>
#include <unistd.h>
#include <sys/time.h>
#include <semaphore.h>
using namespace std;
class Philosopher;
// массив семафоров под вилки
void init_semaf()
{
    sem_fork =new sem_t[5];
   for(int i=0; i<5; i++)
    {
        sem_init(&sem_fork[i],0,1);

    }
}
int main()
{

   init_semaf();
    sem_init(&servant_sem,0,4);// семафор очереди на 4 философа в столовой
    Philosopher Platon(1,2,(char*)"Платон");
    Philosopher Kant(2,3,(char *)"Кант");
    Philosopher Dekart(3,4,(char *)"Декарт");
    Philosopher Gegel(4,5,(char *)"Гегель");
    Philosopher Seneka(5,1,(char *)"Сенека");
    // потоки
    pthread_t thread_philosopher_1;
    pthread_t thread_philosopher_2;
    pthread_t thread_philosopher_3;
    pthread_t thread_philosopher_4;
    pthread_t thread_philosopher_5;



    //создание потоков записи

    int result=pthread_create(&thread_philosopher_1,NULL
```

```
,Philosopher::c_life_philosopher,&Platon);
if(result!=0)
{
    cout<<"Tread write 1 not create"<<endl;
}
result=pthread_create(&thread_philosopher_2,
NULL,Philosopher::c_life_philosopher,&Kant);
if(result!=0)
{
    cout<<"Tread write 2 not create"<<endl;
}
result=pthread_create(&thread_philosopher_3,
NULL,Philosopher::c_life_philosopher,&Dekart);
if(result!=0)
{
    cout<<"Tread write 3 not create"<<endl;
}
result=pthread_create(&thread_philosopher_4
,NULL,Philosopher::c_life_philosopher,&Gegel);
if(result!=0)
{
    cout<<"Tread write 4 not create"<<endl;
}
result=pthread_create(&thread_philosopher_5,
NULL,Philosopher::c_life_philosopher,&Seneka);
if(result!=0)
{
    cout<<"Tread write 5 not create"<<endl;
}
// ожидание выполнения потоков
result = pthread_join(thread_philosopher_1, NULL);
if (result != 0)
 {
    cout<<"Joining the 1 read thread"<<endl;
    return 0;
 }
 result = pthread_join(thread_philosopher_2,
  NULL);
if (result != 0)
 {
    cout<<"Joining the 2 read thread"<<endl;
```

```
        return 0;
     }
     result = pthread_join(thread_philosopher_3,
      NULL);
    if (result != 0)
     {
        cout<<"Joining the 3 read thread"<<endl;
        return 0;
     }
     result = pthread_join(thread_philosopher_4,
      NULL);
    if (result != 0)
     {
        cout<<"Joining the 4 read thread"<<endl;
        return 0;
     }
     result = pthread_join(thread_philosopher_5,
      NULL);
    if (result != 0)
     {
        cout<<"Joining the 5read thread"<<endl;
        return 0;
     }

    return 0;
}
```

3 Источники

- http://pages.cs.wisc.edu/ remzi/OSTEP/threads-sema.pdf
- https://habrahabr.ru/post/261273/
- http://os-pc.ru/proc/122-semafory.html
- http://shpargalum.ru/ shpora-gos-povtas/teoriya -vyichislitelnyix-proczessov/ semafornyie-primitivyi-dejkstryi.html
- http://www.scs.stanford.edu/histar/src/pkg/uclibc/ libpthread/ linuxthreads/semaphore.h