



4. Синхронизация потоков в ОС GNU/Linux

Разделы:

- Проблема синхронизации
- Состояние гонки
- Мьютексы
- Семафоры
- Условные переменные

Состояние гонки

```
#include <malloc.h>
struct job
{
    struct job* next;
    /* Other fields describing work to be done... */
};

struct job* g_JobQueue;

extern void ProcessJob (struct job*);

void* ThreadFunction(void* arg)
{
    while (g_JobQueue != NULL)
    {
        struct job* nextJob = g_JobQueue;
        g_JobQueue = g_JobQueue->next;
        ProcessJob(nextJob);
        free(nextJob);
    }
    return NULL;
}
```



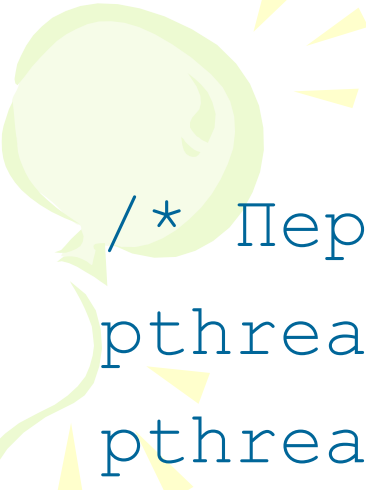
Состояние гонки

- Чтобы исключить возможность гонки, необходимо сделать операции **атомарными**
- Это означает неделимость и непрерываемость операции
 - Если она началась, то уже не может быть приостановлена или прервана до своего завершения
 - Выполнение других операций в это время невозможно

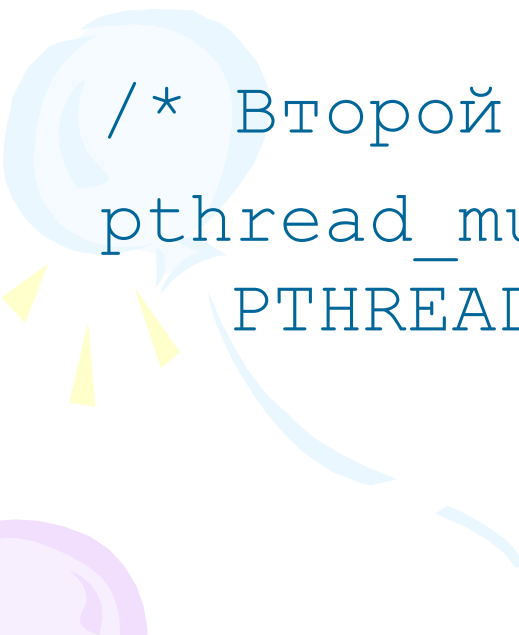
Мьютексы

- Один из вариантов избегания гонки - позволить только одному потоку в один момент времени обращаться к разделяемым данным
- Реализация такого решения требует поддержки от ОС
- В Linux имеется специальное средство, называемое **мьютексом**
- Если мьютекс захвачен каким-то потоком, то другие потоки, обращающиеся к мьютексу, оказываются заблокированным либо переведенным в состояние ожидания
- Как только мьютекс освобождается, поток может продолжить выполнение

Мьютексы



```
/* Первый вариант инициализации */  
pthread_mutex_t mtx;  
pthread_mutex_init(&mtx, NULL);
```



```
/* Второй вариант инициализации */  
pthread_mutex_t mtx =  
    PTHREAD_MUTEX_INITIALIZER;
```

Мьютексы

```
#include <malloc.h>
#include <pthread.h>
struct job
{
    struct job* next;
    /* Other fields describing work to be done... */
};
struct job* g_JobQueue;
extern void ProcessJob(struct job*);
pthread_mutex_t h_JobQueueMutex = PTHREAD_MUTEX_INITIALIZER;

void* ThreadFunction(void* arg)
{
    while (1)
    {
        struct job* nextJob;
        pthread_mutex_lock(&g_JobQueueMutex);
        if (g_JobQueue == NULL)
            nextJob = NULL;
        else
        {
            nextJob = g_JobQueue;
            g_JobQueue = g_JobQueue->next;
        }
        pthread_mutex_unlock(&g_JobQueueMutex);
        if (nextJob == NULL)
            break;
        ProcessJob(nextJob);
        free(nextJob);
    }
    return NULL;
}
```

Мьютексы

```
void EnqueueJob(struct job* newJob)
{
    pthread_mutex_lock(&g_JobQueueMutex);
    newJob->next = g_JobQueue;
    g_JobQueue = newJob;
    pthread_mutex_unlock(&g_JobQueueMutex);
}
```

Взаимоблокировки мьютексов

- Один поток пытается захватить один и тот же мьютекс дважды подряд
- Дальнейшие события зависят от типа мьютекса
- Типы мьютексов:
 - **Быстрый** – приводит к тупику
 - **Рекурсивный** – не приводит к тупику
 - **Контролируемый** – ОС обнаруживает повторный захват и выставляет сигнал

Взаимоблокировки мьютексов

- Для создания рекурсивных и контролируемых мьютексов необходимо создавать объект атрибутов, объявив переменную *pthread_mutexattr_t* и передав указатель на нее функции *pthread_mutexattr_init()*
- Затем следует задать тип мьютекса - *pthread_mutexattr_setkind_np()*
- Рекурсивный мьютекс - *PTHREAD_MUTEX_RECURSIVE_NP*
- контролируемый – *PTHREAD_MUTEX_ERRORCHECK_NP*
- Указатель на объект атрибутов необходимо передать *pthread_mutex_init()*
- После этого необходимо удалить объект атрибутов *pthread_mutexattr_destroy()*

Взаимоблокировки мьютексов

```
pthread_mutexattr_t attr;
```

```
pthread_mutex mtx;
```

```
pthread_mutexattr_init(&attr);
```

```
pthread_mutexattr_setkind_np(&attr,  
    PTHREAD_MUTEX_ERRORCHECK_NP);
```

```
pthread_mutex_init(&mtx, &attr);
```

```
pthread_mutexattr_destroy(&attr);
```

Неблокирующие проверки мьютексов

- Если функция *pthread_mutex_trylock()* обнаруживает, что мьютекс свободен, то работает аналогично функции *pthread_mutex_lock()*
- при этом она возвращает 0
- Если оказывается, что мьютекс уже захвачен каким-то потоком, то функция *pthread_mutex_trylock()* не блокирует программу, а немедленно завершается с кодом *EBUSY*

Потоковые семафоры

- **Семафор** - по сути счетчик (неотрицательное целое число), используемый для синхронизации
- ОС гарантирует, что проверка и модификация значения семафора могут выполняться безопасно и не приведут к гонке
- Операция *sem_wait* уменьшает значение счетчика на единицу
- Операция *sem_post* увеличивает значение счетчика на единицу
- Должен подключаться *semaphore.h*

Потоковые семафоры

- Семафор представляется переменной типа *sem_t*
- Его нужно инициализировать вызовом *sem_init()*
- Для установки семафора – *sem_post()*, для выполнения блокирующего ожидания – *sem_wait()*, для неблокирующего ожидания – *sem_trywait()*
- Если ожидание может привести к блокированию потока из-за того, что счетчик равен 0, то функция завершается с кодом *EAGAIN*
- Также есть функция *sem_getvalue()*, позволяющая узнать текущее значение счетчика семафора

Потоковые семафоры

```
#include <malloc.h>
struct job
{
    struct job* next;
    /* Other fields describing work to be done... */
};
struct job* g_JobQueue;
extern void ProcessJob(struct job*);
pthread_mutex_t g_JobQueueMutex = PTHREAD_MUTEX_INITIALIZER;
sem_t g_JobQueueCount;
void InitializeJobQueue ()
{
    g_JobQueue = NULL;
    sem_init(&g_JobQueueCount, 0, 0);
}
void* ThreadFunction(void* arg)
{
    while (g_JobQueue != NULL)
    {
        struct job* nextJob;
        sem_wait(&g_JobQueueCount);
        pthread_mutex_lock(&g_JobQueueMutex);
        nextJob = g_JobQueue;
        g_JobQueue = g_JobQueue->next;
        pthread_mutex_unlock(&g_JobQueueMutex);
        ProcessJob(nextJob);
        free(nextJob);
    }
    return NULL;
}
```

Потоковые семафоры

```
void EnqueueJob(/* Pass job-specific data here... */)
{
    struct job* newJob;

    newJob = (struct job*) malloc(sizeof (struct job));
    /* Set the other fields of the job struct here... */

    pthread_mutex_lock(&g_JobQueueMutex);
    newJob->next = g_JobQueue;
    g_JobQueue = newJob;

    sem_post(&g_JobQueueCount);

    pthread_mutex_unlock(&g_JobQueueMutex);
}
```

Условные переменные

```
#include <pthread.h>
extern void DoWork ();
int g_ThreadFlag;
pthread_mutex_t g_ThreadFlagMutex;
void InitializeFlag()
{
    pthread_mutex_init(&g_ThreadFlagMutex, NULL);
    g_ThreadFlag = 0;
}
void* ThreadFunction (void* threadArg)
{
    while (!0)
    {
        int flagIsSet;
        pthread_mutex_lock(&g_ThreadFlagMutex);
        flag_is_set = g_ThreadFlag;
        pthread_mutex_unlock(&g_ThreadFlagMutex);
        if (flagIsSet)
            DoWork();
        /* Else don't do anything. Just loop again. */
    }
    return NULL;
}
```




Условные переменные

```
void SetThreadFlag(int flagValue)
{
    pthread_mutex_lock(&g_ThreadFlagMutex);
    g_ThreadFlag = flagValue;
    pthread_mutex_unlock(&g_ThreadFlagMutex);
}
```

Условные переменные

- Условная переменная имеет тип *pthread_cond_t*
- Ей обязательно нужно сопоставить мьютекс
- Функции:
 - *pthread_cond_init()* инициализирует условную переменную
 - *pthread_cond_signal()* сигнализирует об изменении переменной. При этом разблокируется один из потоков, ожидающий сигнала. Если таких нет – сигнал игнорируется
 - *pthread_cond_broadcast()* похожа *pthread_cond_signal()*, только разблокирует **все** потоки, ожидающие сигнал
 - *pthread_cond_wait()* блокирует вызывающий ее поток до тех пор, пока не будет получен сигнал об изменении условной переменной

Условные переменные

- Должны выполняться всякий раз, когда программа так или иначе меняет результат проверки условия, контролируемого условной переменной, следующие шаги:
 1. Захватить мьютекс, дополняющий условную переменную.
 2. Выполнить действие, включающее изменение результата проверки условия
 3. Послать сигнал одному или нескольким потокам об изменении условия
 4. Освободить мьютекс

Условные переменные

```
#include <pthread.h>
extern void DoWork ();

int g_ThreadFlag;
pthread_cond_t g_ThreadFlagCondVar;
pthread_mutex_t g_ThreadFlagMutex;

void InitializeFlag()
{
    pthread_mutex_init(&g_ThreadFlagMutex,
        NULL);
    pthread_cond_init (&g_ThreadFlagCondVar,
        NULL);
    g_ThreadFlag = 0;
}
```

Условные переменные

```
void* ThreadFunction(void* threadArg)
{
    while (!0)
    {
        pthread_mutex_lock(&g_ThreadFlagMutex);
        while(0 == g_ThreadFlag)
            pthread_cond_wait(&g_ThreadFlagCondVar,
                              &g_ThreadFlagMutex);
        pthread_mutex_unlock(&g_ThreadFlagMutex);
        DoWork ();
    }
    return NULL;
}

void SetThreadFlag(int flagValue)
{
    pthread_mutex_lock(&g_ThreadFlagMutex);
    g_ThreadFlag = flagValue;
    pthread_cond_signal(&g_ThreadFlagCondVar);
    pthread_mutex_unlock (&g_ThreadFlagMutex);
}
```

See also

- Робачевский, А. Операционная система Unix, 2 изд./ А.Робачевский, С.Немнюгин, О.Стесик. – СПб.: БХВ-Петербург, 2010. – 656 с.
- POSIX thread (pthread) libraries - <http://www.yolinux.com/TUTORIALS/LinuxTutorialP>
- Инструменты Linux для Windows-программистов - <http://rus-linux.net/nlib.php?name=/MyLDP/BOOKS/Linux-tools/index.html>
- Лав, Р. Linux. Системное программирование/ Р.Лав. – СПб.: Питер, 2008. – 416 с.