

## Лабораторная работа 3. Поток.

**Цели работы:** изучение методов параллельного программирования с использованием потоков.

### 1 Теоретическая часть

#### 1.1 Определения

**Активный участок кода** — участок кода, который либо выполняется в данный момент, либо ожидает разрешения на выполнение от планировщика ОС.

**Поток выполнения** — системная сущность, соответствующая некоторому активному участку кода программы. Потоки выполнения одного процесса существуют в одном адресном пространстве и потому не изолированы друг от друга.

**Общий ресурс** — ресурс (переменная, файл, системный объект), к которому имеют доступ несколько субъектов (потоков или обработчиков сигналов).

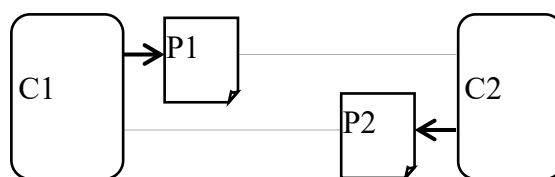
**Состояние гонки** — ситуация, при которой корректность работы программы зависит от того, когда и в каком порядке субъекты выполняют определенные действия.

**Гонка данных** — состояние гонки, вызванное доступом субъектов к общему ресурсу.

**Критическая секция** — участок кода, в котором происходит доступ к общему ресурсу.

**Синхронизация** — обеспечение упорядоченного доступа к общему ресурсу.

**Взаимоблокировка (deadlock)** — ситуация, возникающая, когда субъекты C1 и C2 блокируют ресурсы P1 и P2 соответственно, и при этом пытаются получить доступ к ресурсам P2 и P1 для продолжения выполнения. Поскольку ни один из субъектов не может получить необходимый ресурс, происходит зависание.



**Семафор** — примитив синхронизации, обеспечивающий не более чем  $N$  субъектам доступ к ресурсу в каждый момент времени.

**Мьютекс** — примитив синхронизации, вариант семафора, используемый для организации эксклюзивного доступа к ресурсу (не более чем одному потоку в каждый момент времени).

**Циклическая блокировка** — примитив синхронизации, аналог мьютекса, основанный на активном ожидании (цикле). В отличие от семафора и мьютекса, при неудачной попытке захвата циклическая блокировка не останавливает поток до получения сигнала от ОС о разблокировке, а постоянно проверяет статус блокировки. Выгоднее, чем мьютекс, в случае интенсивной работы с ресурсом и малом времени ожидания.

**Барьер** – примитив синхронизации, блокирующий выполнение потока до тех пор, пока  $N$  потоков не достигнут определенной точки выполнения.

**Условная переменная** (переменная состояния) – примитив синхронизации, предоставляющий возможность уведомления одного или нескольких потоков о достижении определенного условия.

## 1.2 Полезные системные вызовы и библиотечные функции

### 1.2.1 Потоки

#### 1.2.1.1 Создание потоков

Потоки создаются функцией `pthread_create`:

```
#include <pthread.h>

/*Создать новый поток*/
int pthread_create(
    pthread_t *thread           //результат
    , const pthread_attr_t *attr //атрибуты нового потока
    , void *(*start_routine) (void *) //стартовая функция потока
    , void *arg                 //аргумент для start_routine
);
```

Вызов возвращает -1 в случае ошибки.

Параметр `thread` является указателем на переменную в которую будет сохранен дескриптор созданного потока.

Параметр `attr` должен указывать на структуру, содержащую атрибуты создаваемого потока (см. *man pthread\_attr\_init*). Для создания потока с атрибутами по умолчанию в качестве значения этого параметра можно передать `NULL`.

Параметр `start_routine` должен указывать на стартовую функцию потока. Стартовая функция для потока является аналогом `main` для процесса, т. е. поток завершается, когда завершается выполнение стартовой функции. Функция, выступающая в роли стартовой, должна принимать в качестве аргумента указатель на аргумент функции и возвращать указатель на результат. Поскольку и результат, и аргумент могут быть любого типа, оба указателя имеют тип `void*` и должны явно приводиться к требуемому типу. Функция может не использовать свой аргумент, но сам аргумент должен быть указан в сигнатуре функции. Если функция не возвращает никакого значимого результата, она должна вернуть `NULL`.

Для передачи данных в функцию, указанную в `start_routine`, используется аргумент `arg`. Если известно, что `start_routine` не нуждается в аргументах, данный аргумент может иметь значение `NULL`.

#### 1.2.1.2 Завершение потоков

Поток завершается если завершается его функция `start_routine`. Досрочно завершить работу потока можно вызовом `pthread_exit`:

```
#include <pthread.h>

void pthread_exit(void *retval); //завершить поток
```

Параметр `retval` должен содержать указатель на результат работы потока (аналогично возвращаемому значению `start_routine`) либо `NULL`.

**Замечание:** вызов `exit()` или `_exit()` в любом из потоков процесса завершит весь процесс и все его потоки.

Помимо нормального завершения потока, поток может быть завершен другим потоком вызовом `pthread_cancel`:

```
#include <pthread.h>

//послать потоку запрос на завершение
int pthread_cancel(pthread_t thread);
```

Вызов принимает в качестве параметра дескриптор потока. Данный вызов посылает запрос на завершение. По умолчанию, по получении данного запроса, поток будет завершен. Данное поведение может быть изменено (см. *man pthread\_setcancelstate*).

В случае завершения потока из-за `pthread_cancel`, его результатом является константа `PTHREAD_CANCELED`.

### 1.2.1.3 Ожидание завершения потока

Дождаться завершения потока и получить результат его работы можно вызовом `pthread_join`:

```
#include <pthread.h>

//подождать завершения потока и получить результат работы
int pthread_join(pthread_t thread, void **retval);
```

Функция возвращает -1 в случае ошибки.

Первым аргументом является дескриптор потока.

Вторым аргументом `retval` является указатель на переменную, в которую будет помещен указатель на результат работы потока (помните, что функция потока возвращает `void*` - указатель непосредственно на результат либо `NULL`).

Если поток был завершен вызовом `pthread_cancel`, по адресу `retval` будет записана константа `PTHREAD_CANCELED`, которая гарантированно является невалидным указателем.

## 1.2.2 Синхронизация

### 1.2.2.1 Volatile

Ключевое слово `volatile` инструктирует компилятор отказаться от оптимизаций, связанных с доступом к переменной, помеченной данным ключевым словом. Если переменная, доступ к которой из нескольких потоков производится без использования блокировок, то она

должна быть помечена, как `volatile` (например `volatile bool flag`, [пример](#), см. метки L3 и L16).

### 1.2.2.2 Примитивы синхронизации и барьеры памяти

И компилятор, и процессор могут переупорядочивать выполнение операций чтения-записи, что может приводить к весьма печальным последствиям в многопоточных программах.

Если доступ к переменной защищен блокировкой, то компилятор сгенерирует код, гарантирующий доступ к переменной в корректном порядке. В этом случае говорят, что функции доступа устанавливают барьер памяти.

Как следствие, если общий ресурс является составным объектом (например, контейнером STL или просто структурой из нескольких полей) и/или над общим ресурсом выполняются составные операции – общий ресурс должен быть защищен примитивом синхронизации.

### 1.2.2.3 Мьютексы

Мьютекс используется для предоставления эксклюзивного доступа к общему ресурсу (т.е. только 1 поток имеет доступ к ресурсу).

Мьютекс имеет 2 возможных состояния: «захвачен» и «свободен». Попытка захвата «занятого» мьютекса приводит к блокировке потока до освобождения мьютекса. Как следствие, максимум 1 поток «владеет» мьютексом в каждый момент времени.

```
#include <pthread.h>

//инициализировать мьютекс
int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *mutexattr);

//захватить мьютекс
int pthread_mutex_lock(pthread_mutex_t *mutex);

//попытаться захватить мьютекс, если провал - результат==EBUSY
int pthread_mutex_trylock(pthread_mutex_t *mutex);

//освободить мьютекс
int pthread_mutex_unlock(pthread_mutex_t *mutex);

//уничтожить мьютекс
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Вызовы возвращают -1 при ошибке.

Функции `pthread_mutex_init` и `pthread_mutex_destroy` являются аналогами конструктора и деструктора. Дополнительно в `pthread_mutex_init` в параметре `mutexattr` может быть передан указатель на атрибуты создаваемого мьютекса. Если требуется создать мьютекс с атрибутами по умолчанию, данный параметр может быть равен `NULL`.

#### 1.2.2.4 Циклические блокировки

Если время работы критической секции мало, то вместо мьютекса выгоднее использовать циклическую блокировку. Название данная блокировка имеет из-за обычного внутреннего цикла, который проверяет статус блокировки в ожидании ее освобождения. Данный цикл неэффективно расходует процессорное время, т. к. процесс не приостанавливается в ожидании уведомления об освобождении блокировки, как в случае мьютекса, семафора и пр., а активно ждет. Отдельного примера работы с циклическими блокировками нет, т. к. они идентичны мьютексам.

```
#include <pthread.h>

//инициализировать блокировку
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);

//захватить блокировку
int pthread_spin_lock(pthread_spinlock_t * lock);

//попытаться захватить блокировку, если провал - результат==EBUSY
int pthread_spin_trylock(pthread_spinlock_t * lock);

//освободить блокировку
int pthread_spin_unlock(pthread_spinlock_t * lock);

//уничтожить блокировку
int pthread_spin_destroy(pthread_spinlock_t * lock);
```

Назначение функций и их применение аналогичны сходным функциям мьютекса. Параметр `pshared` используется, если циклическая блокировка используется для синхронизации потоков из разных процессов. В рамках лабораторной он должен быть равен 0.

#### 1.2.2.5 Барьеры

Барьеры представляют возможность дождаться достижения N потоками определенной точки выполнения.

```
#include <pthread.h>

//инициализировать барьер
int pthread_barrier_init(pthread_barrier_t *barrier,
    const pthread_barrierattr_t *attr, unsigned int count);

//заблокироваться на барьере и дождаться остальных потоков
int pthread_barrier_wait(pthread_barrier_t *barrier);

//уничтожить барьер
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

При инициализации барьера в параметре `count` указывается количество потоков, которые предполагается синхронизировать.

Вызов `pthread_barrier_wait()` блокирует вызывающий поток до тех пор, пока другие `count-1` потоков не вызовут эту функцию над той же переменной-барьером. Функция возвращает 0 в `count-1` потоках и константу `PTHREAD_BARRIER_SERIAL_THREAD` – в 1 из них (заранее неизвестно, в каком конкретно). Обычно (но не обязательно) данный поток выполняет финальное действие – например, объединяет результаты работы остальных потоков и вычисляет финальный результат.

После того, потоки разблокируются, барьер сбрасывается и снова становится пригоден для использования – нет необходимости заново его инициализировать.

#### 1.2.2.6 Условные переменные

Условные переменные предназначены для уведомления одного или нескольких потоков о выполнении некоторого условия.

```
#include <pthread.h>

//инициализировать условную переменную
int pthread_cond_init(pthread_cond_t *cond,
    pthread_condattr_t *cond_attr);

//разблокировать один из ожидающих потоков
int pthread_cond_signal(pthread_cond_t *cond);

//разблокировать все ожидающие потоки
int pthread_cond_broadcast(pthread_cond_t *cond);

//дождаться разблокировки
int pthread_cond_wait(pthread_cond_t *cond
    , pthread_mutex_t *mutex);
```

```
//подождать разблокировки до указанного времени,  
//если не разблокировались - errno==ETIMEDOUT  
int pthread_cond_timedwait(pthread_cond_t *cond  
                           , pthread_mutex_t *mutex  
                           , const struct timespec *abstime);  
  
//уничтожить условную переменную  
int pthread_cond_destroy(pthread_cond_t *cond);
```

Условные переменные обычно предназначены для работы с неким общим ресурсом, обработку которого требуется производить только при определенном условии (например, общим ресурсом является `std::vector`, обработку которого нужно производить, только если в нем что-то есть).

В простом случае есть 1 поток, который подготавливает данные для обработки, и N потоков, которые обрабатывают данные. Поскольку данные являются общим ресурсом, доступ к ним должен происходить из критической секции, которую защищает мьютекс.

Поток-обработчик:

- 1) захватывает мьютекс и входит в критическую секцию;
- 2) проверяет наличие данных для обработки (проверка условия), если есть - *goto 5*;
- 3) вызывает `pthread_cond_wait()`;
- 4) после разблокировки опять проверяет наличие данных (проверка условия), если нет – *goto 3*;
- 5) забирает свою часть данных и разблокирует мьютекс;
- 6) обрабатывает данные.

Поток-источник данных:

- 1) подготавливает данные для обработки;
- 2) захватывает мьютекс и входит в критическую секцию;
- 3) записывает данные в доступное для всех потоков хранилище (общий ресурс);
- 4) разблокирует один (`pthread_cond_signal`) или все (`pthread_cond_broadcast`) потоки-обработчики;
- 5) разблокирует мьютекс.

Вызов `pthread_cond_wait()` временно разблокирует указанный мьютекс до тех пор, однако в момент выхода из функции поток вновь получает блокировку, т.е. с точки зрения потока он никогда не терял блокировку.



Вызов `pthread_cond_broadcast()` разблокирует все ожидающие потоки-обработчики, однако мьютекс, связанный с критической секцией, потоки повторно захватывают по одному -> остаток критической секции потоки тоже выполняют по одному (порядок заранее не известен).

## 2 Задание на лабораторную работу.

Написать программы с использованием примитивов синхронизации согласно варианту.

Для успешной компиляции в CMakeLists.txt указать следующие строки.

```
find_package(Threads REQUIRED)
link_libraries(Threads::Threads)
```

Помните, что функцию `sleep()` и ее вариации нельзя использовать для синхронизации.

**Общее условие:** преподавателем выдается файл или несколько файлов с данными. Программа должна прочитать данные из файла(-ов), обработать данные *последовательно* и *параллельно* и вывести *результаты и время обработки*. Если результат для последовательного и параллельного случая не совпадает – обосновать причину расхождения.

Файлы содержат данные в бинарном формате – т.е. их необходимо читать/записывать вызовами `read()` и `write()` без всякого парсинга и обработки. Предполагается, что вы умеете определять размер файла и заранее подготавливать буфер соответствующего размера.

**Задание 1.** Написать программу, выполняющую умножение матриц. Преподавателем выдается 2 файла – 2 большие матрицы из чисел типа `double`. Матрицы гарантированно являются квадратными – размер матрицы можно (и нужно) вычислить из размера файла.

Демонстрацию корректности работы алгоритма проводить для матриц размера (10x10) с выводом результата на экран. Время работы замерять для выданных преподавателем матриц.

Данную задачу решить без использования отдельных примитивов синхронизации.

**Задание 2 (легкий уровень).** Написать программу, выполняющую поиск заданного значения в массиве элементов типа `int`. Результатом является индекс элемента в массиве либо -1. Если элементов с таким значением несколько – вернуть индекс **X** вхождения. В качестве примитива синхронизации использовать **Y**.

Вариант	X	Y
1	первого	мьютекс
2	первого	циклическую блокировку
3	первого	барьер
4	последнего	мьютекс
5	последнего	циклическую блокировку
6	последнего	барьер

**Задание 2 (средний уровень).** Написать программу, выполняющую поиск элементов с заданным значением в массиве элементов типа `int`. Результатом являются индексы всех элементов с заданным значением в порядке **X**. В качестве примитива синхронизации использовать **Y**. Помните про то, что в STL нет потокобезопасных коллекций, но с помощью блокировок с ними можно работать потокобезопасно.

Вариант	X	Y
1	возрастания	мьютекс
2	возрастания	циклическую блокировку
3	возрастания	барьер
4	убывания	мьютекс
5	убывания	циклическую блокировку
6	убывания	барьер

**Задание 3 (сложный уровень).** Напишите шаблон класса потокобезопасной очереди сообщений `threadsafe_queue<T>`. Примитивы синхронизации – мьютекс и условная переменная. Полезно вспомнить про блокировку с двойной проверкой.

*Примерный интерфейс:*

```
template <typename T>
class mt_queue{
public:
    mt_queue(size_t max_size);
    mt_queue(const mt_queue&) = delete; //no queue copy
    mt_queue(mt_queue&&) = delete;      //no queue move
    ~mt_queue();
    void enqueue(const T& v); //shall block if full
    T dequeue(); //shall block if empty
    bool full() const;
    bool empty() const;
    std::optional<T> try_dequeue();
    bool try_enqueue(const T& v);
};
```

Продемонстрировать работу с *M* потоками-писателями и *N* потоками-обработчиками. Способ демонстрации – на усмотрение студента. Самый простой способ – писатели пишут в очередь числа, читатели распечатывают числа из очереди (при этом порядок распечатанных элементов может отличаться от исходного, но сама очередь может работать корректно – если такое наблюдается, обосновать).

Помните, что необходимо уведомлять потоки-читатели о том, что новых данных больше не будет. Данная задача может решаться и вне класса `threadsafe_queue<T>` (отдельная

переменная, сигнал, сообщение со специальным значением и пр.). Итоговый способ реализации также остается на усмотрение студента.