Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационных технологий и управления

Кафедра вычислительных методов и программирования

Реферат
по дисциплине «Основы информационных технологий»
на тему «Создание многопоточных приложений»

Выполнил магистрант группы 6М1911 Будный Р. И. _____

Проверил д.ф.-м.н. Колосов С. В. _____

Минск 2016

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные операционные системы предоставляют средства для многопоточного программирования. Каждая исполняемая программа в рамках своего процесса может иметь один или несколько потоков выполнения. Поток выполнения — это наименьшая единица обработки задачи, исполнение которой может быть назначено ядром операционной системы [**?**]. Операционная система выделяет каждому потоку некоторый короткий промежуток времени, в течение которого происходит его выполнение. По истечении этого промежутка активный поток блокируется, и операционная система переходит к следующему. Таким образом достигается видимость одновременного выполнения нескольких потоков в рамках одного процесса — многопоточность.

Основным отличием потоков выполнения от процессов является использование общего адресного пространства. Потоки в рамках одного процесса выполнения имеют общий доступ к данным и другим ресурсам, предоставляемых данному процессу операционной системой.

Существуют различные программные реализации многопоточности. В различных ОС потоки могут быть реализованы на уровне ядра, в пользовательском пространстве, или с использованием различных гибридных схем. Каждый из этих подходов имеет ряд достоинств и недостатков.

Программный интерфейс управления потоками также зависит от конкретной операционной системы. Тем не менее, существуют стандартизированные программные интерфейсы управления потоками. Наиболее известным из них является стандарт управления потоками POSIX. В данном реферате производится обзор данного стандарта, на его примере рассматриваются основные операции управления потоками, а также базовые примитивы синхронизации. Кроме этого, рассматриваются средства поддержки многопоточности, предоставляемые стандартной библиотекой языка C++.

1 СТАНДАРТ РЕАЛИЗАЦИИ ПОТОКОВ ВЫПОЛНЕНИЯ POSIX

В прошлом программные интерфейсы управления потоками, предоставляемые различными операционными системами, существенно различались. Этот факт значительно усложнял написание кроссплатформенных программ. В 1995 году был разработан и опубликован стандарт программного интерфейса потоков POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995). Он представляет собой набор связанных типов, функций и констант языка C, позволяющих управлять жизненным циклом потоков и выполнять их синхронизацию. На данный момент программные интерфейсы упрвления потоками, предоставляемые практически всеми операционными системами на базе UNIX, являются POSIX-совместимыми [**?**].

В соответствии со стандартом, программный интерфейс управления потоками POSIX (Pthreads) описан в заголовочном файле <*pthreads.h*>. Следует отметить, что данный заголовочный файл не является частью стандартной библиотеки языка C.

1.1 Управление жизненным циклом потока

Перечислим основные функции и типы данных стандарта Pthreads, предназначенные для управления потоками.

Функция *pthread_create* предназначена для создания и запуска нового потока. Она принимает следующие аргументы:

— *pthread_t* thread* — идентификатор потока;

— *const pthread_attr_t* attr* — атрибуты потока;

— *void *(*start_routine)(void*)* — указатель на функцию, предназначенную для выполнения в новом потоке;

— *void* arg* — аргумент, передаваемый в *start_routine*.

Данная функция возвращает нулевое значение в случае успеха или код ошибки в противном случае.

Функция *pthread_exit* предназначена для завершения вызывающего потока. Она принимает параметр *void* value_ptr*, предназначенный для передачи возвращаемого значения в поток, ожидающий завершения.

Функция *pthread_join* используется для ожидания вызывающим потоком завершения работы указанного потока. Она принимает следующие аргументы:

— *pthread_t thread* — идентификатор потока, завершение которого мы собираемся ожидать;

— *void** *value_ptr* — двойной указатель на значение переменной *value_ptr*, переданное завершившимся потоком в соответствующий вызов *pthread_exit*.

Функции *pthread_attr_init* и *pthread_attr_destroy* предназначены для инициализации и удаления структуры атрибутов потока соответственно. Структура атрибутов потока используется для задания свойств создаваемого потока с помощью функций, описанных ниже.

С помощью функций *pthread_attr_(get/set)detachstate* можно указать, будет ли поток создан в состоянии *joinable* или *detached*. Разница между этими двумя типами потоков заключается в том, что вызовы функций *pthread_join* и *pthread_detach* в отношении *detached* потока приводят к ошибке. Функция *pthread_detach* используется для перевода указанного *joinable* потока в состояние *detached*.

Функции *pthread_attr_*stack** предназначены для управления параметрами стека запускаемого потока. Дело в том, что значения параметров стека не являются стандартизованными, а поэтому могут различаться на различных ОС. Каждая из этих функций принимает на вход структуру атрибутов потока и связанный параметр:

— *pthread_attr_(get/set)stacksize* — получение/установка размера стека создаваемого потока;

— *pthread_attr_(get/set)stackaddr* — получение/установка стартового адреса стека создаваемого потока.

Поскольку все эти функции осуществляют доступ к структуре атрибутов потока, их вызов должен осуществляться перед созданием потока.

Функция *pthread_self* позволяет вызывающему потоку получить свой идентификатор, а *pthread_equal* позволяет сравнить пару идетификаторов потока. Функция *pthread_once* принимает пару аргументов:

— структуру синхронизации *once_control*;

— функцию *init_routine*, подлежащую запуску в отдельном потоке.

Она устроена таким образом, что её многократные вызовы с одной и той же структурой синхронизации приводят к тому, что её функция-аргумент вызывается в отдельном потоке лишь один (первый) раз. На рисунке 1.1 представлен простейший пример работы с POSIX-потоками.

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 2
#define NUM_ITERS   100
#define NUM_REPEATS 80

static pthread_t threads[NUM_THREADS];
static char thread_args[NUM_THREADS];

void* task(void* arg) {
    char c = *((char*)arg);
    for (size_t i = 0; i < NUM_ITERS; ++i) {
        for (size_t j = 0; j < NUM_REPEATS; ++j) {
            putchar(c);
        }
        putchar('\n');
    }
}

int main(int argc, char** argv) {
    // init thread attributes
    pthread_attr_t thread_attr;
    pthread_attr_init(&thread_attr);
    pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_JOINABLE);
    // start threads
    for (size_t i = 0; i < NUM_THREADS; ++i) {
        thread_args[i] = 'a' + i;
        int err = pthread_create(&threads[i], &thread_attr,
                                 task, (void*)&thread_args[i]);
        if (err) {
            printf("ERROR: pthread_create: %d\n", err);
            exit(-1);
        }
    }
    // join threads
    for (size_t i = 0; i < NUM_THREADS; ++i) {
        int result_value;
        void* result = &result_value;
        int err = pthread_join(threads[i], &result);
        if (err) {
            printf("ERROR: pthread_join: %d\n", err);
            exit(-1);
        }
        printf("INFO: thread %ld joined with result: %d\n", i, result_value);
    }
    pthread_attr_destroy(&thread_attr);
    printf("INFO: completed\n");
    return 0;
}
```

Рисунок 1.1 – Работа с потоками POSIX

Здесь главный поток, выполняющий функцию *main*, создаёт, запускает *NUM_THREADS* потоков, выполняющих функцию *task*, и ожидает их завершения. Функция *task* выполняет циклический вывод аргумента типа *char* на консоль. На рисунке 1.2 приведен участок вывода данной программы.

```
...
aabbaabbaabbaabbaabbaabbaabbbbbbbb
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaba
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbabaaaabbbbaaabbbbbbbaabbbaabb
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaba
...
INFO: thread 0 joined with result: 32765
INFO: thread 1 joined with result: 32765
INFO: completed
```

Рисунок 1.2 – Пример работы программы 1.1

Нетрудно заметить, что результат вывода представляет собой случайную последовательность символов *'a'* и *'b'*. Это происходит вследствие того, что ОС выполняет переключение между выполняемыми потоками в случайные моменты времени, вызывая тем самым прерывание последовательности одинаковых символов, печатаемых данным потоком. Для того, чтобы вывод символов осуществлялся в определенном неслучайном порядке, необходимо выполнять синхронизацию потоков, рассматриваемую в следующем подразделе.

1.2 Примитивы синхронизации

1.2.1 Мьютексы

На практике часто возникает необходимость синхронизации работы некоторого множества потоков. Более точно, необходимость синхронизации потоков возникает всякий раз, когда ими осуществляется доступ к некоторому общему ресурсу, при этом хотя бы один из них изменяет его состояние [**?**]. Подобная ситуация называется состоянием гонки (англ. *race condition*). Выполним краткий обзор средств синхронизации потоков, описанных в стандарте Pthreads.

Наиболее простым средством синхронизации является мьютекс (от англ. MUTual EXclusion), предназначенный для взаимного исключения выполняющихся потоков. Он предоставляет следующие гарантии:

— выполнение различными потоками кода, защищенного мьютексом, выполняется последовательно;

— любые изменения состояния системы (значения переменных, файлов, буферов ввода/вывода и т. д.), осуществляемые в ходе выполнения кода, защищенного мьютексом, становятся доступны всем остальным потокам сразу после его освобождения.

Рассмотрим работу мьютекса M на примере взаимодействия двух потоков, P_1 и P_2, выполняющих защищенный им участок кода C.

1 P_1 и P_2 готовы приступить к выполнению C;

2 ОС переключается на P_1;

3 P_1 захватывает мьютекс M и начинает выполнение C;

4 ОС переключается на P_2;

5 P_2 пытается захватить M и блокируется ОС, поскольку мьютекс в данный момент уже захвачен P_1;

6 ОС переключается на P_1;

7 P_1 завершает выполнение C и освобождает M, делая тем самым совершенные им изменения состояния видимыми для всех потоков;

8 ОС переключается на P_2 и разблокирует его, поскольку M в данный момент не захвачен;

9 P_2 захватывает M, выполняет C, а затем освобождает M.

Сформулируем список правил безопасного использования мьютексов:

— каждый мьютекс, захваченный потоком, должен им освобождаться;

— захват и освобождение множества мьютексов должны осуществляться симметрично во избежание взаимоблокировок (*deadlocks*).

К сожалению, выполнение данных правил, несмотря на простоту их формулировок, на практике является делом весьма затруднительным. Кроме этого, следует иметь в виду, что использование мьютексов, особенно блокирование ими крупных участков кода, приводит к существенному уменьшению скорости работы программы по следующим причинам:

— операции над мьютексами реализуются посредством системных вызовов, которые выполняются достаточно долго по определению;

— запрещаются локальные оптимизации ассемблерных инструкций относительно захвата и освобождения мьютекса, что существенно сказывается на эффективности их исполнения центральным процессором;

— освобождение мьютекса приводит к инвалидации локальных кэшей (*L1* и *L2*) всех ядер процессора.

Рассмотрим программный интерфейс Pthreads для работы с мьютексами. Мьютекс имеет тип *pthread_mutex_t*, а структура его атрибутов — *pthread_mutexattr_t*. Её инициализация может производиться статически (константа *PTHREAD_MUTEX_INITIALIZER* или динамически (функция *pthread_mutexattr_init*). Для освобождения данной структуры используется функция *pthread_mutexattr_destroy*. С помощью данной структуры можно задавать тип мьютекса (обычный или рекурсивный). Инициализация и освобождение мьютексов производится посредством функций *pthread_mutex_init* и *pthread_mutex_destroy* соответственно.

Для захвата мьютекса используются функции *pthread_mutex_lock* и *pthread_mutex_trylock*. Отличие между этими двумя функциями заключается в том, что вторая является неблокирующей — если мьютекс на момент её вызова уже заблокирован другим потоком, она возращает код ошибки в вызывающий поток немедленно, а не блокирует его. Функция *pthread_mutex_unlock* используется для освобождения захваченного мьютекса.

Приведем пример использования мьютекса для для синхронизации вывода данных в консоль. Пусть требуется, чтобы вывод различных потоков не перекрывался, то есть чтобы отдельные символы, генерируемые этими потоками, не чередовались. Для этого нам требуется модифицировать функцию *task* так, как показано на рисунке 1.3.

```
// ...

static pthread_mutex_t mutex_task = PTHREAD_MUTEX_INITIALIZER;

void* task(void* arg) {
    char c = *((char*)arg);
    for (size_t i = 0; i < NUM_ITERS; ++i) {
        pthread_mutex_lock(&mutex_task);
        for (size_t j = 0; j < NUM_REPEATS; ++j) {
            putchar(c);
        }
        putchar('\n');
        pthread_mutex_unlock(&mutex_task);
    }
}

int main(int argc, char** argv) {
    // init thread attributes ...
    // start threads ...
    // join threads ...
    pthread_mutex_destroy(&mutex_task);
    // ...
}
```

Рисунок 1.3 – Использование мьютекса Pthreads

На рисунке 1.4 представлен пример вывода модифицированной версии программы.

```
...
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

Рисунок 1.4 – Пример работы программы 1.3

Как видно из рисунка, теперь вывод программы является более упорядоченным — чередований различных символов больше не наблюдается. Отметим, однако, что результат работы программы продолжает иметь случайный характер, поскольку ОС не предоставляет гарантий того, что поток, освободивший мьютекс, не сможет его захватить повторно, не уступая его своему конкуренту. Для того, чтобы эффективно указать ОС желаемый порядок взаимодействия потоков, наряду с мьютексами требуется использовать еще один примитив синхронизации — условную переменную.

1.2.2 Условные переменные

Условная переменная — это примитив синхронизации, который предназначен для указания порядка взаимодействия между потоками. Условные переменные и мьютексы всегда используются совместно. Рассмотрим простейший пример выполнения кода C, защищенного общим мьютексом M и условной переменной V потоками P_1. Поток P_2 играет при этом управляющую роль. Будем считать, что условная переменная в начале работы имеет значение ЛОЖЬ.

1 P_1 готов приступить к выполнению C;

2 ОС переключается на P_1;

3 P_1 захватывает M, проверяет значение V, и, поскольку её значение равно ЛОЖЬ, освобождает мьютекс и блокируется операционной системой на V;

4 ОС переключается на P_2;

5 P_2 захватывает M, изменяет значение V на ИСТИНА;

6 ОС переключается на P_1;

7 P_1 остаётся в заблокированном состоянии, поскольку M не был освобожден P_2;

8 ОС переключается на P_2;

9 P_2 освобождает мьютекс;

10 ОС переключается на P_1;

11 P_1 захватывает M, проверяет значение V, и, поскольку её значение равно ИСТИНА, продолжает свою работу, а затем освобождает M.

Pthreads предоставляет следующие функции для работы с условными переменными:

— *pthread_cond_init* и *pthread_cond_destroy* для инициализации и уничтожения условной переменной соответственно;

— *pthread_condattr_init* и *pthread_condattr_destroy* для инициализации и уничтожения атрибутов условной переменной соответственно;

— *pthread_cond_wait* для проверки значения условной переменной;

— *pthread_cond_signal* для разблокирования лишь одного потока, ожидающего изменения значения связанной переменной;

— *pthread_cond_broadcast* используется для разблокирования всех потоков, ожидающих изменения значения связанной переменной.

Приведем пример использования условной переменной для для синхронизации вывода данных в консоль. Пусть требуется, чтобы вывод различных потоков происходил по очереди, при этом отдельные символы, генерируемые этими потоками, не чередовались. Для этого требуется модифицировать функцию *task* так, как показано на рисунке 1.5.

```
// ...

static pthread_mutex_t mutex_task = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond_task = PTHREAD_COND_INITIALIZER;

void* task(void* arg) {
    char c = *((char*)arg);
    for (size_t i = 0; i < NUM_ITERS; ++i) {
        pthread_mutex_lock(&mutex_task);
        pthread_cond_wait(&cond_task, &mutex_task);
        for (size_t j = 0; j < NUM_REPEATS; ++j) {
            putchar(c);
        }
        putchar('\n');
        pthread_cond_signal(&cond_task);
        pthread_mutex_unlock(&mutex_task);
    }
}

int main(int argc, char** argv) {
    // init thread attributes ...
    // start threads ...
    pthread_mutex_lock(&mutex_task);
    pthread_cond_signal(&cond_task);
    pthread_mutex_unlock(&mutex_task);
    // join threads ...
    pthread_mutex_destroy(&mutex_task);
    pthread_cond_destroy(&cond_task);
    // ...
}
```

Рисунок 1.5 – Использование условной переменной Pthreads

На рисунке 1.6 представлен пример вывода модифицированной версии программы.

```
...
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

Рисунок 1.6 – Пример работы программы 1.5

2 СЕМАФОРЫ POSIX

Наряду с мьютексами и условными переменными, существует еще один широко известный примитив синхронизации, называемый семафором. Он был предложен голландским ученым Э. Дейкстрой в 1962 году. Семафор предназначен для ограничения числа потоков, которые могут одновременно исполнять некоторый защищенный участок кода.

Над семафором можно производить три операции:

— инициализация семафора, в ходе которой задается начальное значение счетчика;

— захват семафора, в ходе которого если счетчик имел положительное значение, то оно уменьшается на единицу, иначе поток, осуществляющий захват, блокируется;

— освобождение семафора, в ходе которого значение семафора увеличивается на единицу.

Нетрудно заметить, что мьютекс представляет собой двоичный семафор. В некоторых реализациях семафоров используется очередь — потоки, ожидающие освобождения семафора, будут проходить через семафор именно в том порядке, в котором они вызывали пытались его захватить [**?**].

Семафоры не являются частью стандарта Pthreads, а описаны в другом стандарте POSIX.1b, Real-time extensions (IEEE Std 1003.1b-1993). Тем не менее, их можно использовать совместно с POSIX threads. Программный интерфейс работы с семафорами описан в заголовочном файле *<semaphore.h>*.

Семафор POSIX имеет тип *sem_t*. Для работы с ним предусмотрены следующие функции:

— *sem_init* — инициализация семафора;

— *sem_wait* — захват семафора;

— *sem_post* — освободение семафора;

— *sem_getvalue* — получение текущего значения счетчика семафора;

— *sem_destroy* — уничтожение семафора;

На рисунке 2.1 приведена модификация исходной версии программы, использующая семафор для синхронизации многопоточного вывода в консоль.

```c
#include <semaphore.h>

// ...

static sem_t sem_task;

void* task(void* arg) {
    char c = *((char*)arg);
    for (size_t i = 0; i < NUM_ITERS; ++i) {
        sem_wait(&sem_task);
        for (size_t j = 0; j < NUM_REPEATS; ++j) {
            putchar(c);
        }
        putchar('\n');
        sem_post(&sem_task);
    }
}

int main(int argc, char** argv) {
    // init thread attributes
    // ...
    sem_init(&sem_task, 0, 1);
    // start threads
    // ...
    // join threads
    // ...
    sem_destroy(&sem_task);
}
```

Рисунок 2.1 – Использование семафора POSIX

На рисунке 2.2 приведен пример вывода модифицированной версии программы.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

Рисунок 2.2 – Пример работы программы 2.1

Можно утверждать, что вывод программы 2.1, использующей семафор, эквивалентен выводу программы 1.3, использующей мьютекс, поскольку и в том и другом случае вывод потоков является синхронизированным, но его порядок, вообще говоря, не определен.

3 СРЕДСТВА СТАНДАРТНОЙ БИБЛИОТЕКИ ЯЗЫКА C++ ДЛЯ ОРГАНИЗАЦИИ МНОГОПОТОЧНЫХ ВЫЧИСЛЕНИЙ

3.1 Парадигма RAII

Как известно, язык C++ является надмножеством языка C, а стандартная библиотека языка C не содержит средств для организации многопоточных вычислений. Подобно *libc*, стандартная библиотека языка C++ долгое время также не содержала средств для написания многопоточного кода. По-видимому, предполагалось, что интерфейсов, предоставляемых различными ОС C++-разработчикам будет достаточно. Действительно, можно утверждать, что *Pthreads* определяет интерфейс, достаточный для разработки многопоточных программ любой сложности.

В подразделе 1.2.1 было отмечено, что при работе с примитивами синхронизации необходимо соблюдать два правила:

— каждый примитив синхронизации, захваченный потоком, должен им освобождаться;

— захват и освобождение множества примитивов синхронизации должны осуществляться симметрично во избежание взаимоблокировок.

К сожалению, соблюдение этих правил при использовании всякого процедурного интерфейса управления потоками (в том числе и *Pthreads*) в реальных приложениях является затруднительным. Даже малейшее их нарушение приводит к трудноуловимым ошибкам в работе ПО.

Ключевой парадигмой языка C++ является RAII (англ. Resource Aquistition Is Initialization). Данная парадигма утверждает, что для управления ресурсами: памятью, открытыми файлами, соединениями с БД, и т. д., должны использоваться специальные объекты, конструкторы которых должны осуществлять захват ресурсов, а деструкторы — освобождение. В этом случае ресурсы, захваченные объектами, созданными на стеке, будут гарантированно освобождены в случае выхода из функции или возникновении исключительной ситуации [**?**].

Стандарт языка C++, опубликованный в 2011 году (C++11) описывает программный интерфейс базовых средств организации многопоточных вычислений. Данный стандарт является шагом вперед по сравнению с *Pthreads* вперед по друм причинам:

— поскольку C++11 является стандартом языка, обязательным для реализации разработчиками компиляторов языка C++, поддержка описанного

в нем программного интерфейса теперь является частью самого языка, а не отдельной операционной системы;

— поскольку программный интерфейс управления потоками был разработан в соответствии с RAII, его использование позволяет значительно сократить число ошибок программирования.

Отметим, что интерфейс работы с потоками в C++ описан в заголовочных файлах <*thread*>, <*mutex*>, <*condition*> и <*future*>. В следующих подразделах выполнено краткий обзор их содержания.

3.2 Базовые средства управления потоками

К базовым средствам управления потоками стандартной библиотеки будем относить средства, которые были ранее стандартизированы комитетом POSIX: потоки, мьютексы, условные переменные [**?**]. Отметим, что семафоры в этот список не входят.

Рассмотрим стандартный программный интерфейс управления жизненным циклом потока. Поскольку поток выполнения рассматривается как ресурс, ему соответствует объект типа *std::thread*. Данный объект обладает следующими методами:

— *get_id* — возращает уникальный идентификатор потока;

— *joinable* — возвращает *true*, если поток является *joinable*;

— *join* — блокирует текущий поток а ожидании окончания работы данного объекта-потока;

— *detach* — выполняет перевод данного объекта-потока в состояние *detached*.

В дополнение к этому, стандарт определяет ряд функций, позволяющих управлять поведением текущего потока:

— *yield* — используется для того, чтобы уступить вычислительные ресурсы другим потокам;

— *get_id* — для получения идентификатора текущего потока;

— *sleep_for* и *sleep_until* — для блокирования текущего потока на некоторый промежуток времени или до наступления некоторого момента времени в будущем.

Работа с мьютексами осуществляется путем использования объектов типов **_mutex*: *mutex*, *timed_mutex*, *recursive_mutex*, *recursive_timed_mutex*. Каждый из этих объектов предоставляет методы для блокирующего (*lock*) и неблокирующего (*try_lock*) захватов мьютекса, а также для его освобождения (*unlock*). *Timed*-версии мьютексов позволяют указать максимальное время, в течение которого должны осуществляться попытки захвата в абсолютных или относительных единицах.

Захват и освобождение мьютексов могут также производиться в рамках парадигмы RAII. Для этого предназначены типы *lock_guard* и *unique_lock*. Объекты данных типов выполняют захват связанного мьютекса в своих конструкторах, а освобождение — в деструкторах.

Функции *lock* и *try_lock* предназначены для отложенного захвата нескольких мьютексов с их последующим симметричным освобождением.

Условные переменные представлены типами *condition_variable* и *condition_variable_any*. Они отличаются тем, что первая из них способна работать только с *unique_lock*. Функция *notify_all_at_thread_exit* предназначена для того, чтобы данный поток по окончании своей работы разблокировал все потоки, связанные с условной переменной, переданной в качестве аргумента.

На рисунке 3.1 представлена программа, написанная на C++, функционально аналогичная программе 1.5.

```
1  // includes and defines
2  static vector<thread> threads;
3  static mutex mutex_task;
4  static condition_variable cond_task;
5
6  void task(char c) {
7      for (size_t i = 0; i < NUM_ITERS; ++i) {
8          {
9              unique_lock<mutex>(mutex_task);
10             cond_task.wait(lock);
11             for (size_t j = 0; j < NUM_REPEATS; ++j) {
12                 putchar(c);
13             }
14             putchar('\n');
15         }
16         cond_task.notify_one();
17     }
18 }
19
20 int main(int argc, char** argv) {
21     // start threads
22     for (size_t i = 0; i < NUM_THREADS; ++i) {
23         threads.push_back(thread(task, 'a' + i));
24     }
25     // start work
26     this_thread::sleep_for(std::chrono::seconds(1));
27     cond_task.notify_one();
28     // join threads
29     for (size_t i = 0; i < NUM_THREADS; ++i) {
30         threads[i].join();
31     }
32     printf("INFO: completed\n");
33     return 0;
34 }
```

Рисунок 3.1 – Использование условной переменной в языке C++

Отметим, что код программны, написанной на C++, значительно легче читается и является более лаконичным.

3.3 Дополнительные средства управления потоками

В дополнение к уже рассмотренным средствам, стандартная библиотека C++ предоставляет высокоуровневое средство организации вычислительного процесса, называемое *futures*. Оно предназначено для организации удобной передачи возвращаемых значений, в вызывающий поток. Опишем типичный сценарий работы с ним:

1 Вызывающий поток создает объект p типа *promise*, получает доступ к связанному с ним объекту f типа *future* с помощью метода *p.get_future*, передает в вызываемый поток p и блокируется в ожидании получить результатирующее значение с помощью *p.get*.

2 Вызываемый поток выполняет вычисления и сохраняет их результат в *promise*, используя метод *p.set_value*.

3 Вызывающий поток разблокируется и получает значение вычислений в качестве результата вызова *p.get*.

Отметим, что подобным образом можно передавать в вызывающий поток не только значения, но и исключения, возникающие в вызывываемом потоке. Перечислим средства, предназначенные для упрощения программирования наиболее типичных сценариев работы с *futures*:

— объекты класса *packaged_task* — оборачивают указанный вызываемый объект (функцию или функтор), ставя в соответствие его возвращаемому значению *future*;

— функция *async* вызывает указанную функцию или функтор, возвращая результат ее выполнения в виде *future*.

ЗАКЛЮЧЕНИЕ

В данной работе были рассмотрены основные средства организации многопоточного вычислительного процесса. Были описаны как абстрактные операции работы с потоками и их синхронизации, так и их стандартизированные программные реализации. На простых примерах показано использование программного интерфейса управления потоками POSIX, а также потоками стандартной библиотеки C++.