

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	3
1 Стандарт реализации потоков выполнения POSIX	4
1.1 Управление жизненным циклом потока	4
1.2 Примитивы синхронизации	7
1.2.1 Мьютексы	7
1.2.2 Условные переменные	10
2 Семафоры POSIX	12
3 Средства стандартной библиотеки языка C++ для организации многопоточных вычислений	14
Заключение	15
Список использованных источников	15

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 — наименьшая единица обработки задачи, исполнение которой может быть назначено ядром операционной системы [1]. Операционная система выделяет каждому потоку некоторый короткий промежуток времени, в течение которого происходит выполнение активного потока. 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. Наряду с этим, 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 IEEE POSIX 1003.с. Он представляет собой набор связанных типов, функций и констант языка С, позволяющих управлять жизненным циклом потоков и выполнять их синхронизацию. На данный момент программные интерфейсы управления потоками, предоставляемые практически всеми операционными системами на базе UNIX, являются POSIX-совместимыми [2].

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

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 идентификаторов 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.

```

1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define NUM_THREADS 2
6  #define NUM_ITERS 100
7  #define NUM_REPEATS 80
8
9  static pthread_t threads[NUM_THREADS];
10 static char thread_args[NUM_THREADS];
11
12 void* task(void* arg) {
13     char c = *((char*)arg);
14     for (size_t i = 0; i < NUM_ITERS; ++i) {
15         for (size_t j = 0; j < NUM_REPEATS; ++j) {
16             putchar(c);
17         }
18         putchar('\n');
19     }
20 }
21
22 int main(int argc, char** argv) {
23     // init thread attributes
24     pthread_attr_t thread_attr;
25     pthread_attr_init(&thread_attr);
26     pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_JOINABLE);
27     // start threads
28     for (size_t i = 0; i < NUM_THREADS; ++i) {
29         thread_args[i] = 'a' + i;
30         int err = pthread_create(&threads[i], &thread_attr,
31                                 task, (void*)&thread_args[i]);
32         if (err) {
33             printf("ERROR: pthread_create: %d\n", err);
34             exit(-1);
35         }
36     }
37     // join threads
38     for (size_t i = 0; i < NUM_THREADS; ++i) {
39         int result_value;
40         void* result = &result_value;
41         int err = pthread_join(threads[i], &result);
42         if (err) {
43             printf("ERROR: pthread_join: %d\n", err);
44             exit(-1);
45         }
46         printf("INFO: thread %ld joined with result: %d\n", i, result_value);
47     }
48     pthread_attr_destroy(&thread_attr);
49     printf("INFO: completed\n");
50     return 0;
51 }

```

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.

```

1  ...
2  aabbaabbaabbaabbaabbaabbbbbb
3  aaaaaaaaaaaaaaaaaaaaaaaaaaba
4  bbbbbbbbbbbbbbbbbbbbbbbbabaaaabbbbbaabbbbbbaabbaabb
5  aaaaaaaaaaaaaaaaaaaaaaaaaaba
6  ...
7  INFO: thread 0 joined with result: 32765
8  INFO: thread 1 joined with result: 32765
9  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 Прimitives синхронизации

1.2.1 Мьютексы

На практике часто возникает необходимость синхронизации работы набора потоков. Более формально, необходимость синхронизации потоков возникает всякий раз, когда ими осуществляется доступ к некоторому общему ресурсу, при этом хотя бы один из них изменяет его состояние [3]. 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.), выполненные в коде, защищенном 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 выполнения кода 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 возвращает код ошибки в вызывающий поток немедленно, а не блокирует его. Функция *pthread_mutex_unlock()* используется для освобождения захваченного мьютекса.

Приведем пример использования мьютекса 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.

```

1 // ...
2
3 static pthread_mutex_t mutex_task = PTHREAD_MUTEX_INITIALIZER;
4
5 void* task(void* arg) {
6     char c = *((char*)arg);
7     for (size_t i = 0; i < NUM_ITERS; ++i) {
8         pthread_mutex_lock(&mutex_task);
9         for (size_t j = 0; j < NUM_REPEATS; ++j) {
10             putchar(c);
11         }
12         putchar('\n');
13         pthread_mutex_unlock(&mutex_task);
14     }
15 }
16
17 int main(int argc, char** argv) {
18     // init thread attributes ...
19     // start threads ...
20     // join threads ...
21     pthread_mutex_destroy(&mutex_task);
22     // ...
23 }

```

Рисунок 1.3 – Использование мьютекса Pthreads

На рисунке 1.4 представлен пример вывода модифицированной версии программы.

```

1 ...
2 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
3 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
4 bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

```

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 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.

```

1 // ...
2
3 static pthread_mutex_t mutex_task = PTHREAD_MUTEX_INITIALIZER;
4 static pthread_cond_t cond_task = PTHREAD_COND_INITIALIZER;
5
6 void* task(void* arg) {
7     char c = *((char*)arg);
8     for (size_t i = 0; i < NUM_ITERS; ++i) {
9         pthread_mutex_lock(&mutex_task);
10        pthread_cond_wait(&cond_task, &mutex_task);
11        for (size_t j = 0; j < NUM_REPEATS; ++j) {
12            putchar(c);
13        }
14        putchar('\n');
15        pthread_cond_signal(&cond_task);
16        pthread_mutex_unlock(&mutex_task);
17    }
18 }
19
20 int main(int argc, char** argv) {
21     // init thread attributes ...
22     // start threads ...
23     pthread_mutex_lock(&mutex_task);
24     pthread_cond_signal(&cond_task);
25     pthread_mutex_unlock(&mutex_task);
26     // join threads ...
27     pthread_mutex_destroy(&mutex_task);
28     pthread_cond_destroy(&cond_task);
29     // ...
30 }
```

Рисунок 1.5 – Использование условной переменной Pthreads

На рисунке 1.6 представлен пример вывода модифицированной версии программы.

```

1 ...
2 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
3 bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
4 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
5 bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

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 попытки его захватить [4].

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()` — освобождение 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.

```

1  #include <semaphore.h>
2
3  // ...
4
5  static sem_t sem_task;
6
7  void* task(void* arg) {
8      char c = *((char*)arg);
9      for (size_t i = 0; i < NUM_ITERS; ++i) {
10         sem_wait(&sem_task);
11         for (size_t j = 0; j < NUM_REPEATS; ++j) {
12             putchar(c);
13         }
14         putchar('\n');
15         sem_post(&sem_task);
16     }
17 }
18
19 int main(int argc, char** argv) {
20     // init thread attributes
21     // ...
22     sem_init(&sem_task, 0, 1);
23     // start threads
24     // ...
25     // join threads
26     // ...
27     sem_destroy(&sem_task);
28 }

```

Рисунок 2.1 – Использование семафора POSIX

На рисунке 2.2 приведен пример вывода модифицированной версии программы.

```

1  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
2  aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
3  bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
4  bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

```

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

*** Управление жизненным циклом потока + `std::thread` + `std::mutex`
+ `std::lock_guard` + `std::condition_variable` + `std::promise` + `std::future` +
`std::async` *** Прimitives синхронизации

ЗАКЛЮЧЕНИЕ

...

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Поток выполнения [Электронный ресурс]. — [https://en.wikipedia.org/wiki/Thread_\(computing\)](https://en.wikipedia.org/wiki/Thread_(computing)) : [б. и.].
- [2] POSIX Threads Programming [Электронный ресурс]. — [Б. м. : б. и.]. — Режим доступа: <https://computing.llnl.gov/tutorials/pthreads/>.
- [3] Tanenbaum, Andrew S. Modern Operating Systems [Текст] / Andrew S. Tanenbaum. — 3rd изд. — Upper Saddle River, NJ, USA : Prentice Hall Press, 2007. — ISBN: 9780136006633.
- [4] Semaphore (programming) [Электронный ресурс]. — [Б. м. : б. и.]. — Режим доступа: [https://en.wikipedia.org/wiki/Semaphore_\(programming\)](https://en.wikipedia.org/wiki/Semaphore_(programming)).