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 — 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.c. 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 [2].

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);
    }
    printf("INFO: completed\n");
    return 0;
}
```

Рисунок 1.1 – Пример работы 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
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbabaaaabbbbaaabbbbbbbaabbbaabb
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaba
INFO: thread 0 joined with result: 32765
INFO: thread 1 joined with result: 32765
INFO: completed
```

Рисунок 1.2 – Пример несинхронизированного вывода

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 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.

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

*** Управление жизненным циклом потока + std::thread + std::mutex + std::lock_guard + std::condition_varibale + std::promise + std::future + std::async *** Примитивы синхронизации

ЗАКЛЮЧЕНИЕ

...

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Поток выполнения [Электронный ресурс]. — 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.