Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

ОТЧЁТ

к лабораторной работе №5

на тему

**УПРАВЛЕНИЕ ПОТОКАМИ, СРЕДСТВА СИНХРОНИЗАЦИИ**

Выполнил: студент гр.253504 Лавренова А.С.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2025

СОДЕРЖАНИЕ

Введение3

[1 [Ц](#_agswffz6g6ei)ель работы](#_Toc178067889) 4

[2 Теоретические сведения 5](#_Toc178067890)

[3 Описание и пример выполнения программы 7](#_Toc178067891)

Заключение [10](#_Toc178067899)

[Список использованных источников 11](#_Toc178067900)

[Приложение А (обязательное) 12](#_Toc178067901)

# ВВЕДЕНИЕ

Лабораторная работа направлена на углублённое изучение механизмов управления потоками и средств синхронизации, доступных в операционных системах семейства Unix/Linux. В программировании многопоточность играет ключевую роль при разработке высокопроизводительных приложений, способных эффективно использовать ресурсы многоядерных процессоров. Однако взаимодействие между потоками, особенно при доступе к общим ресурсам, требует использования специальных средств, обеспечивающих корректность и согласованность выполнения — таких как мьютексы, семафоры, барьеры и другие примитивы синхронизации.

В рамках данной работы проводилось практическое исследование эффективности различных средств синхронизации, реализованных в POSIX-совместимых системах. Особое внимание уделялось оценке производительности и времени выполнения в условиях конкурентного доступа, а также изучению поведения разных механизмов при изменении числа потоков и частоты обращения к критическим секциям. Это позволяет на практике выявить сильные и слабые стороны каждого подхода и сделать осознанный выбор при проектировании многопоточных приложений.

Кроме того, лабораторная работа позволила закрепить навыки работы с API потоков POSIX (pthread), освоить базовые принципы организации параллельных вычислений и разработать тестирующий стенд, позволяющий проводить эксперименты в различных конфигурациях. Полученные результаты служат основой для понимания того, как тонко и осознанно нужно подходить к синхронизации при создании надёжных и эффективных многопоточных систем.

1 ЦЕЛЬ РАБОТЫ

Цель данной лабораторной работы заключается в комплексном освоении и глубоких практических исследованиях механизмов многопоточности и синхронизации в операционных системах семейства Unix/Linux. Сейчас эффективность использования ресурсов и масштабируемость становятся критически важными факторами, умение разрабатывать и отлаживать многопоточные приложения является неотъемлемым навыком. В рамках данной работы особое внимание уделяется не только техническим аспектам создания потоков и их управления с помощью POSIX API, но и детальному изучению средств синхронизации, обеспечивающих корректную работу параллельных вычислений при совместном доступе к разделяемым данным.

В ходе выполнения лабораторной работы ставится задача реализовать универсальную тестовую программу, моделирующую конкурентный доступ множества потоков к критическим ресурсам с применением различных механизмов синхронизации: мьютексов нескольких типов (включая NORMAL, RECURSIVE и ERRORCHECK), POSIX и System V семафоров, спин-блокировок и барьеров. Программа должна поддерживать возможность конфигурирования числа потоков и частоты обращения к критическим секциям, чтобы оценить поведение и эффективность каждого из примитивов при разных условиях нагрузки. Это позволяет не только теоретически сравнить доступные средства, но и эмпирически подтвердить или опровергнуть предполагаемые особенности их функционирования.

Целью также является формирование навыков системного анализа производительности и надежности параллельных программ, развитие способности выявлять узкие места в многопоточных приложениях и делать обоснованный выбор наиболее эффективного механизма синхронизации с учётом особенностей конкретной задачи. Работа направлена на то, чтобы студент научился самостоятельно проектировать и исследовать многопоточные решения, оценивать их масштабируемость, выявлять накладные расходы, минимизировать влияние побочных факторов и грамотно интерпретировать результаты измерений.

В конечном итоге цель лабораторной работы — углублённо понять механизмы взаимодействия потоков в Unix-среде, научиться применять и сочетать различные средства синхронизации в зависимости от контекста задачи, а также выработать инженерное мышление при построении надёжных и производительных многопоточных приложений.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Linux — это свободная операционная система (точнее, семейство систем) с открытым исходным кодом. Само название Linux относится к ядру системы, которое является ее ключевым компонентом, управляющим центральным процессором (ЦП), памятью и периферийными устройствами компьютера. [1]

Поток выполнения - в вычислительной технике, наименьшая последовательность запрограммированных команд, которые могут независимо управляться планировщиком, являющимся частью операционной системы. Реализация потоков и процессов различается в операционных системах, но в большинстве случаев поток является составной частью процесса. Несколько потоков могут существовать в одном процессе, одновременно выполняться и совместно использовать ресурсы, такие как память, в то время как разные процессы не делят эти ресурсы. В частности, потоки процесса совместно используют свой исполняемый код и значения переменных в любой момент времени.

Множественные нити исполнения в одном процессе называют потоками и это базовая единица загрузки ЦПУ, состоящая из идентификатора потока, счетчика, регистров и стека. Потоки внутри одного процесса делят секции кода, данных, а также различные ресурсы: описатели открытых файлов, учетные данные процесса сигналы, значения umask, nice, таймеры и прочее.

У всех исполняемых процессов есть как минимум один поток исполнения. Некоторые процессы этим и ограничиваются в тех случаях, когда дополнительные нити исполнения не дают прироста производительности, но только усложняют программу. Однако таких программ с каждым днем становится относительно меньше.

С точки зрения пользователя в ОС UNIX существует два типа объектов: файлы и процессы. Все данные хранятся в виде файлов, доступ к периферийным устройствам осуществляется через чтение/запись в специальные файлы. При запуске программы ядро загружает соответствующий исполняемый файл, создает образ процесса и передает ему управление. Во время выполнения процесс может считывать или писать данные в файл. С другой стороны, вся функциональность ОС определяется выполнением соответствующих процессов. Таким образом, понятия файловой системы и процессов тесно взаимосвязаны. [2]

Мьютекс действует как хранитель доступа к разделу кода, содержащему общий ресурс, разрешая входить одному потоку и блокируя доступ ко всем остальным потокам. Это гарантирует, что управляемый код будет одновременно привлекаться только одним потоком. В Inux предусмотрено три типа мьютексов, которые имеют различное применение друг от друга.

Когда вы не указываете какой-либо тип при создании мьютекса, мьютекс создается с обычным типом (по умолчанию). Когда thread-1 получает блокировку мьютекса, thread-2 останавливает свое выполнение до тех пор, пока thread-1 не разблокирует его. Как только thread-1 разблокирует (освободит) блокировку мьютекса, thread-2 получит блокировку и начнет выполнять код thread-2.

Мьютекс Errorcheck блокируется потоком ровно один раз, как и обычный мьютекс. Если поток снова попытается заблокировать мьютекс, не разблокировав его, он получит сообщение об ошибке. Если поток, отличный от владельца, пытается разблокировать мьютекс проверки ошибок, возвращается ошибка. Таким образом, мьютексы проверки ошибок более информативны, чем обычные мьютексы.

Рекурсивный мьютекс может быть заблокирован одним потоком более одного раза, не вызывая взаимоблокировки. Когда поток впервые успешно блокирует рекурсивный мьютекс, он становится владельцем этого мьютекса, а счетчик блокировок устанавливается равным 1. Любой другой поток, пытающийся заблокировать мьютекс, блокируется до тех пор, пока мьютекс не будет разблокирован. Если владелец мьютекса снова попытается заблокировать мьютекс, счетчик блокировок увеличивается, и поток продолжает работу. Когда владелец разблокирует рекурсивный мьютекс, количество блокировок уменьшается.

Барьер – это механизм синхронизации, который позволяет приостановить выполнение потоков у некоторой точки программы до тех пор, пока все потоки не дойдут до этого места, и только затем продолжить выполнение дальше. [3]

В отличие от join-а, где мы ожидаем завершения выполнения потока, при использовании барьеров мы ожидаем встречи (рандеву, rendezvous) этих потоков в определённой точке. Мы определяем количество потоков, которые должны прибыть к этой точке, блокируем их там, а когда набирается нужное число потоков, разблокируем их все, позволяя работать дальше.

Наиболее часто используемый тип блокировки в ядре Linux — это спин-блокировки (spin lock). Спин-блокировка — это блокировка, которую может удерживать не более чем один поток выполнения. Если поток выполнения пытается захватить блокировку, которая находится в состоянии конфликта (contended), т.е. уже захвачена, поток начинает выполнять постоянную циклическую проверку (busy loop) — «вращаться» (spin), ожидая на освобождение блокировки. Если блокировка не находится в состоянии конфликта при захвате, то поток может сразу же захватить блокировку и продолжить выполнение. Циклическая проверка предотвращает ситуацию, в которой более одного потока одновременно может находиться в критическом участке. Следует заметить, что одна и та же блокировка может использоваться в нескольких разных местах кода, и при этом всегда будет гарантирована защита и синхронизация при доступе, например, к какой-нибудь структуре данных.

В данном разделе рассмотрены основные аспекты многопоточности в Linux, включая потоки, процессы и механизмы синхронизации, такие как мьютексы и барьеры. Эти инструменты обеспечивают безопасный доступ к общим ресурсам и эффективное управление выполнением потоков, что критично для разработки надежных многопоточных приложений.

**3 ОПИСАНИЕ И ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ**

Разработанная в рамках лабораторной работы программа предназначена для оценки эффективности различных средств синхронизации потоков в многопоточном приложении. Она реализована на языке программирования C++ с использованием POSIX API и запускается в среде операционной системы Linux. Основной задачей является сравнение времени выполнения при использовании следующих примитивов синхронизации: мьютексов разныхтипов(PTHREAD\_MUTEX\_NORMAL,PTHREAD\_MUTEX\_ERRORCHECK, PTHREAD\_MUTEX\_RECURSIVE), POSIX семафоров, семафоров System V, спин-блокировок (pthread\_spinlock\_t) и барьеров (pthread\_barrier\_t).

Программа реализует критическую секцию, к которой многократно обращаются потоки. Эти обращения синхронизируются с помощью указанных примитивов. Основное внимание уделяется точному измерению времени, затраченного на выполнение серии операций при заданном количестве потоков и интенсивности обращений (количестве итераций). Конфигурация задаётся параметрами командной строки: первым аргументом указывается количество создаваемых потоков, вторым — число итераций обращения к критической секции в каждом потоке.

Пример вызова программы:./CompIPC 10 1000. В данном примере создаётся 10 потоков, и каждый из них обращается к критической секции 1000 раз (рисунок 3.1).

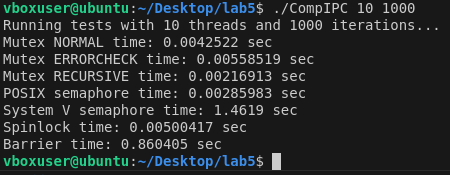


Рисунок 3.1 – Пример работы программного продукта

Полученные результаты экспериментального запуска программы с 10 потоками и 1000 итерациями позволяют сделать несколько важных наблюдений о производительности различных средств синхронизации. Наиболее быстрые и эффективные средства синхронизации в этом тесте показали крайне низкие значения времени выполнения — особенно это касается рекурсивных мьютексов и POSIX семафоров, что свидетельствует об их низких накладных расходах при работе в условиях умеренной конкуренции между потоками.

Мьютексы типов NORMAL и ERRORCHECK также продемонстрировали хорошую производительность, однако их время оказалось немного выше, что может быть связано с внутренними проверками корректности операций блокировки и разблокировки. Спинлоки, несмотря на предполагаемую высокую скорость, показали не самое минимальное время, что, возможно, обусловлено активным ожиданием, создающим лишнюю нагрузку на процессор в многопоточном режиме.

System V семафоры оказались самыми медленными из всех исследованных механизмов. Это объясняется их архитектурой: взаимодействие через ядро и работу с межпроцессной памятью, что значительно увеличивает время реакции даже при локальном запуске. Аналогичная ситуация наблюдается и с барьерами, которые хоть и полезны для синхронизации завершения этапов между потоками, имеют существенные накладные расходы при большом количестве итераций.

Таким образом, можно сделать вывод, что в условиях высокой частоты обращения к критической секции и при работе внутри одного процесса наилучшие результаты показывают легковесные механизмы синхронизации, такие как POSIX семафоры и рекурсивные мьютексы. Более тяжёлые средства, ориентированные на межпроцессное взаимодействие, хоть и надёжны, в подобных условиях оказываются менее производительными.

Для автоматизации тестирования и сбора статистики создан Bash-скрипт runtest.sh, в котором предусмотрены несколько конфигураций запуска. Скрипт автоматически запускает программу с различными параметрами и сохраняет все результаты в файл results.txt.

Типовой вывод программы в терминале показан на рисунке 3.2.

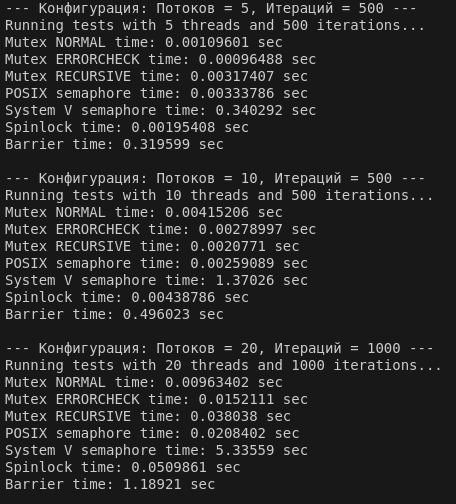


Рисунок 3.2 – Отработка теста

Проведённый сравнительный анализ различных средств синхронизации при нескольких конфигурациях количества потоков и числа итераций позволяет наглядно проследить, как масштабирование влияет на производительность. При небольшом числе потоков и низкой нагрузке все лёгкие примитивы — такие как обычные мьютексы, POSIX-семафоры и спинлоки — демонстрируют практически мгновенное выполнение, с разницей в доли миллисекунды. Это указывает на их эффективность в условиях слабой конкуренции.

По мере увеличения числа потоков и итераций начинает проявляться разница в накладных расходах: времена выполнения у мьютексов и POSIX семафоров возрастают умеренно, а у System V семафоров и барьеров — экспоненциально. Особенно сильно падает производительность у System V семафоров, что подтверждает их ограниченность в задачах высокой конкуренции в пределах одного процесса. Барьеры также демонстрируют значительный рост времени, особенно при увеличении количества потоков, что объясняется их природой — они требуют синхронного завершения всех потоков на каждом этапе.

В целом видно, что для задач, критичных к времени отклика, наиболее рациональным выбором остаются POSIX семафоры, обычные мьютексы и, в определённых случаях, спинлоки. Более тяжёлые механизмы, такие как System V семафоры и барьеры, лучше использовать в специфических сценариях, где необходима жёсткая координация между потоками или взаимодействие между процессами.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была проведена комплексная реализация и экспериментальное исследование поведения различных средств синхронизации потоков в операционной системе Linux. Основное внимание уделялось оценке эффективности таких примитивов, как мьютексы разных типов, POSIX-семафоры, System V семафоры, спин-блокировки и барьеры. Программа была построена таким образом, чтобы создавать измеримую нагрузку на критическую секцию, многократно вызываемую потоками, при этом все замеры производительности проводились в условиях, максимально приближённых к реальным: с ограничением внешнего ввода-вывода, с постоянным числом итераций, варьируемым количеством потоков и контролируемой частотой обращений.

Полученные результаты демонстрируют, как различия в архитектуре средств синхронизации напрямую влияют на производительность многопоточной программы. Простые и легковесные средства, такие как обычные мьютексы и POSIX-семафоры, показали наилучшее время выполнения при любом количестве потоков. Это подтверждает их пригодность для решения задач, требующих минимальных накладных расходов при синхронизации. Напротив, средства, предполагающие более сложное взаимодействие между потоками или даже процессами, например System V семафоры, продемонстрировали существенно более высокие задержки, что делает их менее предпочтительными для задач высокой интенсивности в пределах одного процесса.

Дополнительное внимание было уделено разработке автоматизированного скрипта тестирования, что позволило воспроизводимо и гибко варьировать параметры конфигурации, обеспечивая объективность замеров. Программа показала устойчивую и корректную работу на всех этапах, а результаты тестов позволили не только сравнить эффективность используемых механизмов, но и выявить их поведение при увеличении нагрузки и числа конкурирующих потоков.

В целом, проведённая работа способствовала глубокому пониманию устройства и взаимодействия потоков в Unix-подобных системах, а также позволила на практике освоить ключевые технологии синхронизации. Полученные знания и результаты экспериментов могут быть использованы в будущем при проектировании высоконагруженных многопоточных систем, где важно не только обеспечить корректное взаимодействие между потоками, но и достичь максимальной производительности с минимальными издержками.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Linux и UNIX: программирование в shell. Руководство разработчика. Дэвид Тейнсли – [Электронный ресурс]. – Режим доступа: https://coollib. com/b/275969/read?ysclid=m6ujdy2lag999562018.

[2] Внутреннее устройство Linux. Дмитрий Кетов – [Электронный ресурс]. – Режим доступа: https://djvu.online/file/wkYDVsl8LdABg?ysclid=mdi of2iooo685605108.

[3] UNIX. Профессиональное программирование. У. Ричард Стивенс, Стивен А. Раго – [Электронный ресурс]. – Режим доступа: http://amplab.ru/bo oks/7.pdf.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код

#include <iostream>

#include <pthread.h>

#include <semaphore.h>

#include <sys/time.h>

#include <sys/sem.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <unistd.h>

#include <cstring>

#include <vector>

using namespace std;

int iterations = 1000;

bool verbose = false;

// Получение времени в секундах

double get\_time() {

struct timeval tv;

gettimeofday(&tv, NULL);

return tv.tv\_sec + tv.tv\_usec / 1e6;

}

void critical\_function(int thread\_id) {

if (verbose)

cout << "Thread " << thread\_id << " accessed critical section." << endl;

}

//==================== MUTEX ====================

struct MutexCtx {

pthread\_mutex\_t\* mutex;

int thread\_id;

int iterations;

};

void\* mutex\_thread(void\* arg) {

MutexCtx\* ctx = (MutexCtx\*)arg;

for (int i = 0; i < ctx->iterations; ++i) {

pthread\_mutex\_lock(ctx->mutex);

critical\_function(ctx->thread\_id);

pthread\_mutex\_unlock(ctx->mutex);

}

return NULL;

}

pthread\_mutex\_t create\_mutex(int type) {

pthread\_mutex\_t mutex;

pthread\_mutexattr\_t attr;

pthread\_mutexattr\_init(&attr);

pthread\_mutexattr\_settype(&attr, type);

pthread\_mutex\_init(&mutex, &attr);

pthread\_mutexattr\_destroy(&attr);

return mutex;

}

void run\_mutex\_test(int type, const string& name, int num\_threads, int local\_iterations) {

pthread\_mutex\_t mutex = create\_mutex(type);

pthread\_t threads[num\_threads];

double t\_start = get\_time();

for (int i = 0; i < num\_threads; ++i) {

MutexCtx\* ctx = new MutexCtx{&mutex, i, local\_iterations};

pthread\_create(&threads[i], NULL, mutex\_thread, ctx);

}

for (int i = 0; i < num\_threads; ++i) pthread\_join(threads[i], NULL);

double t\_end = get\_time();

cout << name << " time: " << t\_end - t\_start << " sec" << endl;

pthread\_mutex\_destroy(&mutex);

}

//==================== SPINLOCK ====================

struct SpinCtx {

pthread\_spinlock\_t\* spin;

int thread\_id;

int iterations;

};

void\* spin\_thread(void\* arg) {

SpinCtx\* ctx = (SpinCtx\*)arg;

for (int i = 0; i < ctx->iterations; ++i) {

pthread\_spin\_lock(ctx->spin);

critical\_function(ctx->thread\_id);

pthread\_spin\_unlock(ctx->spin);

}

return NULL;

}

void run\_spinlock\_test(int num\_threads, int local\_iterations) {

pthread\_spinlock\_t spin;

pthread\_spin\_init(&spin, 0);

pthread\_t threads[num\_threads];

double t\_start = get\_time();

for (int i = 0; i < num\_threads; ++i) {

SpinCtx\* ctx = new SpinCtx{&spin, i, local\_iterations};

pthread\_create(&threads[i], NULL, spin\_thread, ctx);

}

for (int i = 0; i < num\_threads; ++i) pthread\_join(threads[i], NULL);

double t\_end = get\_time();

cout << "Spinlock time: " << t\_end - t\_start << " sec" << endl;

pthread\_spin\_destroy(&spin);

}

//==================== POSIX SEM ====================

struct SemPosixCtx {

sem\_t\* sem;

int thread\_id;

int iterations;

};

void\* sem\_posix\_thread(void\* arg) {

SemPosixCtx\* ctx = (SemPosixCtx\*)arg;

for (int i = 0; i < ctx->iterations; ++i) {

sem\_wait(ctx->sem);

critical\_function(ctx->thread\_id);

sem\_post(ctx->sem);

}

return NULL;

}

void run\_posix\_test(int num\_threads, int local\_iterations) {

sem\_t sem;

sem\_init(&sem, 0, 1);

pthread\_t threads[num\_threads];

double t\_start = get\_time();

for (int i = 0; i < num\_threads; ++i) {

SemPosixCtx\* ctx = new SemPosixCtx{&sem, i, local\_iterations};

pthread\_create(&threads[i], NULL, sem\_posix\_thread, ctx);

}

for (int i = 0; i < num\_threads; ++i) pthread\_join(threads[i], NULL);

double t\_end = get\_time();

cout << "POSIX semaphore time: " << t\_end - t\_start << " sec" << endl;

sem\_destroy(&sem);

}

//==================== SYSV SEM ====================

struct SemSysVCtx {

int sem\_id;

int thread\_id;

int iterations;

};

void\* sem\_sysv\_thread(void\* arg) {

SemSysVCtx\* ctx = (SemSysVCtx\*)arg;

struct sembuf op;

for (int i = 0; i < ctx->iterations; ++i) {

op = {0, -1, 0};

semop(ctx->sem\_id, &op, 1);

critical\_function(ctx->thread\_id);

op = {0, 1, 0};

semop(ctx->sem\_id, &op, 1);

}

return NULL;

}

void run\_sysv\_test(int num\_threads, int local\_iterations) {

key\_t key = ftok(".", 's');

int sem\_id = semget(key, 1, IPC\_CREAT | 0666);

semctl(sem\_id, 0, SETVAL, 1);

pthread\_t threads[num\_threads];

double t\_start = get\_time();

for (int i = 0; i < num\_threads; ++i) {

SemSysVCtx\* ctx = new SemSysVCtx{sem\_id, i, local\_iterations};

pthread\_create(&threads[i], NULL, sem\_sysv\_thread, ctx);

}

for (int i = 0; i < num\_threads; ++i) pthread\_join(threads[i], NULL);

double t\_end = get\_time();

cout << "System V semaphore time: " << t\_end - t\_start << " sec" << endl;

semctl(sem\_id, 0, IPC\_RMID);

}

//==================== BARRIER ====================

struct BarrierCtx {

pthread\_barrier\_t\* barrier;

int thread\_id;

int iterations;

};

void\* barrier\_thread(void\* arg) {

BarrierCtx\* ctx = (BarrierCtx\*)arg;

for (int i = 0; i < ctx->iterations; ++i) {

pthread\_barrier\_wait(ctx->barrier);

critical\_function(ctx->thread\_id);

}

return NULL;

}

void run\_barrier\_test(int num\_threads, int local\_iterations) {

pthread\_barrier\_t barrier;

pthread\_barrier\_init(&barrier, NULL, num\_threads);

pthread\_t threads[num\_threads];

double t\_start = get\_time();

for (int i = 0; i < num\_threads; ++i) {

BarrierCtx\* ctx = new BarrierCtx{&barrier, i, local\_iterations};

pthread\_create(&threads[i], NULL, barrier\_thread, ctx);

}

for (int i = 0; i < num\_threads; ++i) pthread\_join(threads[i], NULL);

double t\_end = get\_time();

cout << "Barrier time: " << t\_end - t\_start << " sec" << endl;

pthread\_barrier\_destroy(&barrier);

}

int main(int argc, char\* argv[]) {

int num\_threads = 20;

if (argc > 1) num\_threads = atoi(argv[1]);

if (argc > 2) iterations = atoi(argv[2]);

if (argc > 3 && strcmp(argv[3], "--verbose") == 0) verbose = true;

cout << "Running tests with " << num\_threads << " threads and " << iterations << " iterations...\n";

run\_mutex\_test(PTHREAD\_MUTEX\_NORMAL, "Mutex NORMAL", num\_threads, iterations);

run\_mutex\_test(PTHREAD\_MUTEX\_ERRORCHECK, "Mutex ERRORCHECK", num\_threads, iterations);

run\_mutex\_test(PTHREAD\_MUTEX\_RECURSIVE, "Mutex RECURSIVE", num\_threads, iterations);

run\_posix\_test(num\_threads, iterations);

run\_sysv\_test(num\_threads, iterations);

run\_spinlock\_test(num\_threads, iterations);

run\_barrier\_test(num\_threads, iterations);

return 0;

}