Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Пермский национальный исследовательский политехнический университет»

Факультет: Прикладной математики и механики
Кафедра: Вычислительной математики, механики и биомеханики

Направление: 09.04.02 Информационные технологии и системная инженерия
Профиль: «Информационные технологии и системная инженерия»

Лабораторные работы по дисциплине: «Параллельное программирование»

Выполнил
студент гр. ИТСИ-24-1м
Дурыманов Данил Игоревич

Принял Преподаватель кафедры ВММБ
Истомин Денис Андреевич

Пермь 2025

Лабораторная работа №1

Задача на лабораторную работу:

1. При помощи SSE инструкций написать программу (или функцию), которая перемножает массив из 4х чисел размером 32 бита.
2. Написать аналогичную программу (или функцию) которая решает ту же задачу последовательно.
3. Сравнить производительность
4. Проанализировать сгенерированный ассемблер: gcc -S sse.c

Выполнение:

В рамках лабораторной работы были реализованы две версии алгоритма умножения элементов массива. Первая версия задействует набор инструкций SSE, предоставляющий возможности параллельной обработки данных. Вторая версия представляет собой классическую последовательную реализацию, где элементы обрабатываются по одному.

В ходе эксперимента производилось замеривание времени выполнения обеих реализаций. Результаты показали, что использование SIMD-инструкций (в частности, SSE) позволяет достичь значительного прироста в скорости выполнения: SSE-реализация работает примерно в 7 раз быстрее по сравнению с обычной последовательной. Такой прирост объясняется тем, что SIMD позволяет выполнять операции над несколькими данными одновременно, тогда как традиционный подход выполняет их поочерёдно.

Кроме того, был изучен ассемблерный код, автоматически сгенерированный компилятором GCC. При анализе видно, что в код действительно были включены инструкции SSE, что подтверждает корректную компиляцию и использование SIMD на уровне низкоуровневого представления. Присутствие этих инструкций указывает на то, что компилятор эффективно преобразовал исходный код в оптимизированные машинные команды.

Выводы:

На практике было продемонстрировано, насколько эффективным может быть применение SIMD-технологий для обработки массивов и выполнения вычислительных операций. SSE-инструкции обеспечивают параллельную обработку данных, что в разы повышает производительность программы. Особенно заметно это становится при сравнении с последовательным методом, где каждый элемент обрабатывается отдельно. Анализ ассемблерного кода подтвердил, что компилятор корректно применил SSE-инструкции, а значит, задуманная оптимизация была успешно реализована. Эта лабораторная работа позволила наглядно убедиться в том,

как современные технологии процессорной архитектуры могут ускорить выполнение вычислений даже в относительно простых задачах.

Лабораторная работа №2

Задача на лабораторную работу:

1. При помощи Pthreads написать программу (или функцию), которая создает n потоков и каждый из потоков выполняет длительную операцию.
2. Написать аналогичную программу (или функцию) которая решает ту же задачу последовательно.
3. Сравнить производительность

Выполнение:

В рамках лабораторной работы были реализованы две версии программы: многопоточная и однопоточная. В многопоточной реализации использовалась POSIX-библиотека потоков (Pthreads), с помощью которой создавались несколько потоков, каждый из которых выполнял вычислительно затратную задачу. В однопоточной реализации та же задача выполнялась поочерёдно — без распараллеливания.

После запуска и измерения времени выполнения обеих версий была зафиксирована значительная разница в производительности. Программа, использующая Pthreads, показала примерно десятикратное преимущество по скорости выполнения по сравнению с последовательной реализацией. Это связано с тем, что при наличии нескольких процессорных ядер потоки способны работать одновременно, что позволяет значительно сократить общее время работы программы.

Многопоточность оказалась особенно эффективной при выполнении независимых задач, так как потоки могли выполняться параллельно без необходимости синхронизации данных. Это позволило задействовать аппаратные ресурсы более полно и добиться существенного ускорения.

Выводы:

Проведённый эксперимент подтвердил высокую эффективность многопоточного подхода при решении ресурсоёмких задач. Использование Pthreads позволило реализовать параллельную обработку, благодаря чему общее время выполнения программы заметно сократилось. Особенно это проявляется в условиях наличия нескольких ядер, когда каждый поток может быть назначен на отдельное ядро, обеспечивая настоящую параллельность.

В результате выполнения лабораторной работы удалось убедиться, что грамотное использование многопоточности с применением Pthreads способно значительно повысить производительность приложений, особенно в задачах,

где возможна независимая обработка данных. Таким образом, использование Pthreads позволило убедиться в реальных преимуществах многопоточности при решении вычислительно сложных задач.

Лабораторная работа №3

Задача на лабораторную работу:

1. При помощи OpenMP написать программу (или функцию), которая создает `n` потоков и каждый из потоков выполняет длительную операцию.
2. Сравнить с последовательной программой и программой с Pthreads из предыдущей лабораторной работы.

Выполнение:

Была разработана многопоточная реализация, основанная на OpenMP — технологии для параллельного программирования, предоставляющей высокоуровневый интерфейс для распараллеливания задач. В данной программе создаются n потоков, и каждый поток выполняет длительную операцию независимо от других.

Далее была проведена серия сравнений по времени выполнения между тремя версиями программы:

- последовательно выполняющей операции в одном потоке;
- многопоточной реализацией с использованием библиотеки Pthreads;
- реализацией на OpenMP.

Результаты сравнения показали, что OpenMP-версия продемонстрировала наивысшую производительность среди всех трёх вариантов. Она значительно обогнала последовательную реализацию, как и ожидалось, благодаря распараллеливанию. Что касается сравнения с Pthreads, то OpenMP показала лишь незначительное преимущество, но тем не менее была немного быстрее.

Предположительно, это связано с тем, что OpenMP предоставляет более удобный и эффективный способ управления потоками на уровне компилятора. Платформа самостоятельно оптимизирует распределение задач между доступными ядрами процессора, минимизируя накладные расходы, связанные с ручным управлением потоками, как это происходит в случае с Pthreads.

Выводы:

Эксперимент наглядно продемонстрировал преимущества использования OpenMP при реализации многопоточных вычислений. Помимо ускорения выполнения за счёт параллелизма, OpenMP позволяет

писать более компактный, понятный и удобочитаемый код, избавляя программиста от необходимости вручную создавать, запускать и синхронизировать потоки.

В сравнении с Pthreads реализация на OpenMP показала не только лучшее время выполнения, но и большую простоту в разработке. Последовательный же вариант предсказуемо оказался наименее эффективным из всех трёх. Таким образом, OpenMP можно рассматривать как мощный и удобный инструмент для создания высокопроизводительных программ, особенно в случае параллельных вычислений.

Лабораторная работа №4

Задача на лабораторную работу:

1. Написать программу, которая запускает несколько потоков
2. В каждом потоке считывает и записывает данные в HashMap, Hashtable, synchronized HashMap, ConcurrentHashMap
3. Модифицировать функцию чтения и записи элементов по индексу так, чтобы в многопоточном режиме использование непотокобесопасной коллекции приводило к ошибке
4. Сравнить производительность

Выполнение:

Была разработана программа, в которой каждый поток выполняет операции записи и чтения данных из четырёх типов коллекций: стандартной HashMap, устаревшей потокобезопасной Hashtable, обёртки Collections.synchronizedMap над HashMap, а также высокопроизводительной ConcurrentHashMap.

Для выявления проблем с потокобезопасностью, функции, взаимодействующие с коллекциями, были специально сконструированы таким образом, чтобы в случае отсутствия должной синхронизации возникали ошибки (например, ConcurrentModificationException или непредсказуемое поведение данных).

В процессе тестирования производительности и стабильности в многопоточном окружении были получены следующие результаты:

- **ConcurrentHashMap** показала наивысшую скорость и корректную работу при параллельном доступе.
- **SynchronizedMap** обеспечивала безопасность, но за счёт полной блокировки, что снижало производительность.
- **HashMap** в условиях многопоточности вела себя нестабильно и приводила к ошибкам, хотя в однопоточном режиме оставалась самой быстрой.

- **Hashtable**, несмотря на встроенную синхронизацию, оказалась самой медленной из-за чрезмерной блокировки при доступе к каждому элементу.

Выводы:

Результат показал, насколько важно правильно выбирать коллекции при работе в многопоточном контексте. ConcurrentHashMap обеспечивает оптимальное сочетание производительности и безопасности, позволяя нескольким потокам одновременно читать и изменять данные без жёсткой блокировки.

В то время как HashMap можно использовать только в однопоточной среде, её применение без внешней синхронизации в параллельных сценариях приводит к ошибкам и непредсказуемым последствиям. SynchronizedMap улучшает безопасность, но проигрывает в скорости. Hashtable, хоть и обеспечивает потокобезопасность, морально устарела и неэффективна в современных многопоточных приложениях.

Таким образом, при проектировании многопоточных решений важно опираться на современные и оптимизированные структуры данных, такие как ConcurrentHashMap, которые позволяют добиться высокой производительности без ущерба для безопасности.

Лабораторная работа №5

Задача на лабораторную работу:

1. Написать программу, которая демонстрирует работу считающего семафора
2. Написать собственную реализацию семафора (наследование от стандартного с переопределением функций) и использовать его

Выполнение:

В рамках лабораторной работы была разработана многопоточная программа, в которой управление доступом к ресурсу осуществлялось с помощью семафора. Для этой цели был создан отдельный класс, реализующий собственную версию семафора.

Реализация была построена на основе механизма блокировок ReentrantLock. В классе переопределялись основные методы управления семафором, включая acquire() — для получения разрешения, release() — для освобождения ресурса, и availablePermits() — для определения текущего количества доступных разрешений.

Созданная структура была использована в многопоточном коде для ограничения количества одновременно работающих потоков, взаимодействующих с общей областью программы.

Выводы:

Семафор оказался эффективным инструментом для ограничения параллельного доступа к ресурсу. Он позволяет задать точное количество потоков, которым разрешено выполнение критической секции кода, тем самым предотвращая состояния гонки и избыточную нагрузку на систему.

Реализация собственной версии семафора на основе стандартных средств синхронизации помогла глубже разобраться в принципах его работы. Это также продемонстрировало, что при необходимости семафор можно адаптировать под специфические требования, что особенно важно в сложных многопоточных системах, где стандартного поведения может быть недостаточно.

Таким образом, работа показала, что знание и возможность изменения механизмов синхронизации, таких как семафоры, являются полезным инструментом для эффективного управления потоками в современных приложениях.

Лабораторная работа №6

Задача на лабораторную работу:

Необходимо создать клиент-серверное приложение:

1. Несколько клиентов, каждый клиент - отдельный процесс
2. Серверное приложение - отдельный процесс
3. Клиенты и сервер общаются с использоваени Socket

Необходимо релизовать функционал:

1. Клиент подключается к серверу
2. Сервер запоминает каждого клиента в java.util.concurrent.CopyOnWriteArrayList
3. Сервер читает ввод из консоли и отправляет сообщение всем подключенным клиентам

Выполнение:

Были созданы два основных компонента: серверное приложение и клиентская часть. Оба модуля используют сокеты для установления и поддержания соединения. При запуске сервера и подключении нескольких клиентов (в данном случае — двух), каждый клиент устанавливал соединение с сервером и ожидал входящих сообщений.

Сервер, в свою очередь, сохранял подключённых клиентов в структуре CopyOnWriteArrayList, которая обеспечивает потокобезопасный доступ без необходимости явной синхронизации. Это особенно важно, так как подключение и отключение клиентов может происходить параллельно.

Сервер читал ввод с консоли (например, сообщение от пользователя или команды) и отправлял его всем активным клиентам. Клиенты, получив сообщение, корректно его отображали в своей консоли.

Выводы:

В процессе выполнения лабораторной работы была успешно реализована базовая модель клиент-серверного взаимодействия с использованием сокетов. Сервер корректно обрабатывает многократные подключения, а благодаря использованию CopyOnWriteArrayList список активных клиентов остаётся защищённым от ошибок, связанных с одновременным доступом из разных потоков.

Практическое тестирование показало, что архитектура устойчива и надёжна: сообщения от сервера доставляются всем подключённым клиентам, а сами клиенты стабильно поддерживают соединение и обрабатывают входящие данные.

Таким образом, данный подход может служить основой для создания более сложных распределённых приложений, требующих безопасной и эффективной работы с несколькими клиентами в реальном времени.

Лабораторная работа №7

Задача на лабораторную работу:

Изучить использование библиотеки MappedBus: запустить example).

Выполнение:

Был изучен и успешно запущен пример, использующий библиотеку MappedBus. Программа демонстрировала реализацию обмена информацией между потоками или процессами за счёт применения файла, отображённого в память (memory-mapped file). Этот подход позволяет создавать каналы связи с высокой пропускной способностью и минимальными задержками.

Примерная реализация включала в себя чтение и запись данных через общее пространство, отображённое в файл, что даёт возможность передавать данные между различными сущностями (например, потоками одного приложения или независимыми процессами).

Выводы:

Благодаря запуску и анализу работы MappedBus стало понятно, что эта библиотека предоставляет эффективное решение для организации IPC (межпроцессного взаимодействия) и передачи данных между потоками без использования более тяжёлых и сложных механизмов, таких как сокеты или брокеры сообщений.

Работа с отображением памяти в файл позволяет добиться высокой производительности и низкого уровня задержек при обмене информацией. Это делает библиотеку особенно ценной для систем, где критична скорость передачи данных, например, в финансовых или высоконагруженных приложениях.

В целом, MappedBus представляет собой надёжный и производительный инструмент, позволяющий реализовать быстрый и масштабируемый обмен данными, с минимальными накладными расходами и простой интеграцией в Java-приложения.