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
Сыкулев Антон Александрович

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:

Была написана программа выполняющая перемножение массива из 4х чисел размером 32 бита при помощи SSE инструкций. Так же была написана программа, выполняющая эти действия последовательно. В результате сравнения производительности было выявлено, что программа, использующая SSE инструкции, выполняется в разы быстрее той (примерно в 7 раз), что делает перемножение последовательно. В сгенерированном коде ассемблера можно видеть что, операции, которые прописаны с помощью SSE были вставлены в код ассемблера.

Выводы:

После запуска обеих версий и замера времени выполнения стало ясно, что программа, использующая SSE, справляется с задачей заметно быстрее. В среднем она работает примерно в 7 раз быстрее, чем последовательная реализация. Это объясняется тем, что SSE-инструкции позволяют обрабатывать сразу несколько элементов массива за одну операцию, в то время как обычный подход обрабатывает каждый элемент отдельно.

Дополнительно был проанализирован сгенерированный ассемблерный код. В нём можно увидеть, что компилятор действительно вставил SIMD-инструкции, подтверждая, что операции выполняются с использованием SSE, как и задумывалось.

Таким образом, эта работа на практике показала, как использование SIMD-расширений, таких как SSE, может значительно ускорить выполнение даже относительно простых вычислительных задач.

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:

Была написана программа, которая используется pthreads, создавая многопоточное вычисление сложной задачи. Так же была написана программа, выполняющая вычисление сложной задачи последовательно. При сравнении производительности выяснилось что программа, использующая Pthreads, выполняется быстрее (примерно в 10 раз), чем та, что делает это последовательно.

Выводы:

После выполнения и тестирования обеих реализаций стало очевидно, что использование потоков значительно ускоряет выполнение программы. На практике многопоточная версия показала прирост производительности примерно в 10 раз по сравнению с последовательной. Это объясняется тем, что при наличии нескольких ядер процессора потоки действительно могут работать параллельно, эффективно распределяя нагрузку и сокращая общее время выполнения.

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:

При помощи OpenMP, была написана программа, которая создаёт n потоков и каждый из потоков выполняет длительную операцию. Сравнивая с последовательным выполнением и с программой, использующей pthreads,

заметно что программа, использующая OpenMP является быстрейшей из всех вышеперечисленной, сильно быстрее последовательных операций и немногим быстрее, чем программа использующая pthreads.

Выводы:

Результаты запусков показали, что программа с OpenMP демонстрирует наилучшую производительность среди всех трёх вариантов. Она выполнялась значительно быстрее, чем последовательная реализация, где все операции обрабатывались поочерёдно в одном потоке.

Сравнение с Pthreads также оказалось в пользу OpenMP. Хотя обе реализации используют многопоточность, OpenMP оказалась немного быстрее. Это связано, скорее всего, с тем, что OpenMP предлагает более высокоуровневый и оптимизированный механизм распределения задач между потоками. Она также упрощает написание кода и позволяет компилятору более эффективно управлять ресурсами на уровне процессора.

Можно сделать вывод, что OpenMP не только ускоряет выполнение задач за счёт распараллеливания, но и позволяет писать более лаконичный и понятный код по сравнению с ручным управлением потоками через Pthreads.

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 написана программа, которая в каждом потоке считывает и записывает данные в HashMap, HashTable, synchonized HashMap, ConcurrenctHashMap. Функция чтения, записи была написана так, что бы она вызывала ошибку при использовании непотокобезопасной коллекции. Производительность коллекций от самой быстрой, до самой медленной выглядит следующим образом - ConcurrentHashMap, SyncMap, HashMap и HashTable.

Выводы:

Наилучшие результаты показала ConcurrentHashMap — она обеспечила высокую скорость при корректной и безопасной работе с несколькими потоками. Далее по эффективности шла synchronizedMap, которая хоть и обеспечивала потокобезопасность, но делала это путём полной блокировки коллекции, что влияло на производительность. HashMap оказалась быстрее только в однопоточном режиме, а при многопоточном доступе приводила к ошибкам. Hashtable, хоть и потокобезопасна, показала наихудшие результаты по скорости из-за избыточной синхронизации.

В результате можно сделать важные выводы: при проектировании многопоточных программ важно выбирать подходящие структуры данных. ConcurrentHashMap обеспечивает оптимальный баланс между безопасностью и производительностью. Использование HashMap без дополнительной синхронизации недопустимо в многопоточной среде, а Hashtable и synchronizedMap постепенно уступают современным решениям, таким как ConcurrentHashMap, как по удобству, так и по эффективности.

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:

Была написана, которая выполняет задачу в несколько потоков, при этом используя написанный в отдельном классе семафор. Была написана реализация семафора, используя ReentrantLock. В этом классе переопределялись основные методы, такие как acquire(), release() и availablePermits().

Выводы:

Семафор позволяет точно ограничить количество потоков, которые могут одновременно выполнять критическую секцию кода или обращаться к ресурсу. Практика показала, что его использование помогает избежать конфликтов и перегрузки. Самостоятельная реализация семафора через наследование дала более глубокое понимание внутренней работы этого механизма и показала, что его можно модифицировать и настраивать под задачи конкретного приложения. Это особенно полезно в более сложных

многопоточных системах, где стандартные средства требуют дополнительной логики.

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 изучены и запущены классы отвечающие за клиент-серверное приложение, а именно класс сервиса и клиента, которые «общаются» между собой через сокет. При запуске сервера и запуске двух клиентов, возможно осуществление общения между клиентами.

Выводы:

Сервер прослушивает подключения от клиентов и сохраняет каждое активное подключение в потокобезопасную коллекцию CopyOnWriteArrayList. Это позволяет безопасно работать со списком клиентов даже в условиях многопоточности, когда одновременно могут подключаться и отключаться разные клиенты. После подключения, каждый клиент получает возможность принимать сообщения от сервера.

Во время тестирования приложение было запущено с двумя клиентами и одним сервером. Каждый клиент успешно подключался, получал сообщения с сервера и корректно отображал их. Это показало, что передача данных через сокеты работает стабильно, а использование CopyOnWriteArrayList обеспечивает безопасную работу с подключениями без необходимости ручной синхронизации.

Лабораторная работа №7

Задача на лабораторную работу:
Изучить использование библиотеки MappedBus: запустить example).

Выполнение:

Была изучен и запущен класс использующий библиотеку MappedBus. Класс реализовывает способ межпроцессного взаимодействия (IPC) или межпоточного обмена данными, используя memory-mapped файл.

Выводы:

В результате выполнения лабораторной работы была изучена и протестирована библиотека MappedBus, предоставляющая эффективный способ организации межпроцессного и межпоточного взаимодействия на основе памяти, отображённой в файл (memory-mapped file). Данный механизм позволяет реализовать быструю передачу объектов между потоками или даже отдельными процессами, минуя традиционные каналы связи, такие как сокеты или очереди.

Таким образом, библиотека MappedBus является удобным и мощным инструментом для высокопроизводительных Java-приложений, и её применение может значительно ускорить межпроцессное взаимодействие по сравнению с более традиционными средствами.