

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
Пеленев Денис Вячеславович

Принял Преподаватель кафедры ВММБ
Истомин Денис Андреевич

Пермь 2025

Оглавление

Лабораторная работа 1	4
Задание.....	4
Реализация	4
Результат.....	4
Лабораторная работа 2	5
Задание.....	5
Реализация	5
Результат.....	5
Лабораторная работа 3	7
Задание.....	7
Реализация	7
Результат.....	7
Лабораторная работа 4	9
Задание.....	9
Необходимо:	9
Реализация	9
Результат.....	10
Лабораторная работа 5	12
Задание.....	12
Необходимо:	12
Реализация	12
Результат.....	12
Лабораторная работа 6	13
Задание.....	13
Реализация	13
Результат.....	13
Лабораторная работа 7	15
Задание.....	15
Необходимо:	15

Реализация	15
Результат.....	15

Лабораторная работа 1

Задание

Необходимо:

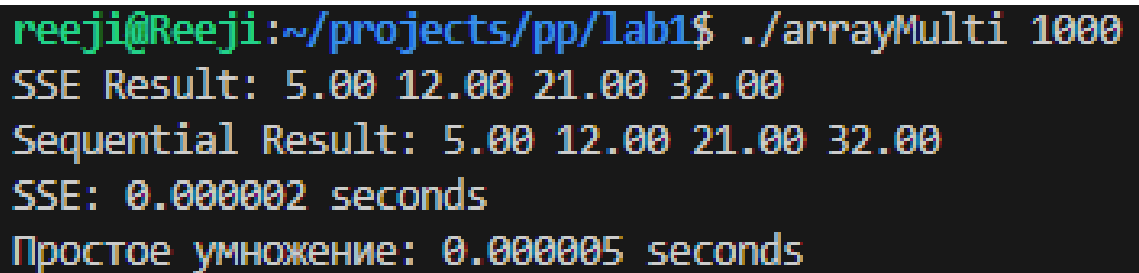
1. При помощи SSE инструкций написать программу (или функцию), которая перемножает массив из 4х чисел размером 32 бита;
2. Написать аналогичную программу (или функцию) которая решает ту же задачу последовательно;
3. Сравнить производительность;
4. Проанализировать сгенерированный ассемблер: `gcc -S sse.c`.

Реализация

Была написана программа, которая выполняет перемножение массива из 4 чисел при помощи последовательных действий, через цикл, а также при помощи SSE инструкции.

Программа, которая выполняет перемножение при помощи SSE инструкции, завершила работу в 4 раза быстрее. Это связано с тем, что SSE использует принцип SIMD, при помощи которой числа перемножаются не по одному, а 4 числа в массиве за раз.

Результат



```
reeji@Reeji:~/projects/pp/lab1$ ./arrayMulti 1000
SSE Result: 5.00 12.00 21.00 32.00
Sequential Result: 5.00 12.00 21.00 32.00
SSE: 0.000002 seconds
Простое умножение: 0.000005 seconds
```

Рис. 1 Результат работы программы в Лабораторной 1

Лабораторная работа 2

Задание

Необходимо:

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.

Реализация

Была написана программа, которая производит вычисление при помощи последовательных действий и используя pthreads для многопоточного вычисления в n потоках.

Программа, которая реализует многопоточное вычисление, выполнялась за 2.76 секунд, а последовательное за 9.87.

Результат

```
=== Последовательное выполнение ===
Sequential task #0 started
Sequential task #0 finished
Sequential task #1 started
Sequential task #1 finished
Sequential task #2 started
Sequential task #2 finished
Sequential task #3 started
Sequential task #3 finished
Sequential task #4 started
Sequential task #4 finished
Sequential task #5 started
Sequential task #5 finished
Sequential task #6 started
Sequential task #6 finished
Sequential task #7 started
Sequential task #7 finished
Sequential task #8 started
Sequential task #8 finished
Sequential task #9 started
Sequential task #9 finished
Время последовательного выполнения: 9.87 секунд
```

Рис. 2 Последовательное вычисление

```
=== Pthreads выполнение ===
MAIN: starting thread 0
MAIN: starting thread 1
MAIN: starting thread 2
MAIN: starting thread 3
MAIN: starting thread 4
MAIN: starting thread 5
MAIN: starting thread 6
MAIN: starting thread 7
MAIN: starting thread 8
MAIN: starting thread 9
      Thread #2 finished
      Thread #1 finished
      Thread #9 finished
      Thread #3 finished
      Thread #6 finished
      Thread #0 finished
      Thread #5 finished
      Thread #7 finished
      Thread #8 finished
      Thread #4 finished
Время выполнения Pthreads: 2.76 секунд
```

Рис. 3 Многопоточное вычисление

Лабораторная работа 3

Задание

Необходимо:

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.

Реализация

Была написана программа, которая производит вычисление при помощи последовательных действий и используя OpenMP для многопоточного вычисления в n потоках.

Программа, которая реализует многопоточное вычисление, выполнилась за 2.66 секунд, а последовательное за 9.87. По скорости вычислений OpenMP обгоняет последовательное вычисление и pthreads.

Результат

```
=== Последовательное выполнение ===
Sequential task #0 started
Sequential task #0 finished
Sequential task #1 started
Sequential task #1 finished
Sequential task #2 started
Sequential task #2 finished
Sequential task #3 started
Sequential task #3 finished
Sequential task #4 started
Sequential task #4 finished
Sequential task #5 started
Sequential task #5 finished
Sequential task #6 started
Sequential task #6 finished
Sequential task #7 started
Sequential task #7 finished
Sequential task #8 started
Sequential task #8 finished
Sequential task #9 started
Sequential task #9 finished
Время последовательного выполнения: 9.87 секунд
```

Рис. 4 Последовательное вычисление

```
=== OpenMP выполнение ===  
param: 10  
OpenMP threads: 1  
id: 7  
id: 3  
id: 8  
id: 0  
id: 1  
id: 2  
id: 6  
id: 9  
id: 4  
id: 5  
Время выполнения OpenMP: 2.66 секунд
```

Рис. 5 Многопоточное вычисление

Лабораторная работа 4

Задание

Необходимо:

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 непотокобезопасной коллекции приводило к ошибке;
4. Сравнить производительность.

Реализация

В данной лабораторной работе были рассмотрены 4 вида коллекции в многопоточном приложении из 50 потоков. В данных коллекциях были использованы функции записи и чтения из коллекции, что позволило проверить потокобезопасность.

1. `HashMap` – непотокобезопасная коллекция. При работе с ней возникали ошибки связанные с попыткой конкурентного доступа к общим данным. Что в свою очередь так же сказалось и на финальном результате.
2. `Hashtable` – потокобезопасная коллекция. В данной коллекции уже не возникало гонки данных, что обеспечило корректный результат работы.
3. `Collections.synchronizedMap` – потокобезопасная коллекция. В данной коллекции уже не возникало гонки данных, что обеспечило корректный результат работы.
4. `ConcurrentHashMap` – потокобезопасная коллекция. В данной коллекции уже не возникало гонки данных, что обеспечило корректный результат работы.

Как видно из результатов, `HashMap` не стоит использовать, когда необходима коллекция, к которой будет конкурентный доступ. Наиболее быстрой и при этом безопасной коллекцией является `ConcurrentHashMap`. Связано это с тем, что в отличие от `HashTable` и `synchronizedMap`, которые блокируются полностью при доступе к ним, в `ConcurrentHashMap` блокируются лишь отдельные бакеты, что позволяет другим потокам в это время вести работу с другими бакетами.

Результат

```
Thread pool-1-thread-5 encountered error: null  
Thread pool-1-thread-4 encountered error: null  
Thread pool-1-thread-28 encountered error: null  
Thread pool-1-thread-17 encountered error: null
```

Рис. 6 Ошибки при работе HashMap

```
HashMap integrity check:  
Expected size: 3, Actual size: 30  
Key: 0, Value: 145610  
Key: 1, Value: 104203  
Key: 2, Value: 145452  
  
hashTable integrity check:  
Expected size: 3, Actual size: 30  
Key: 0, Value: 166546  
Key: 1, Value: 167030  
Key: 2, Value: 166739  
  
syncMap integrity check:  
Expected size: 3, Actual size: 30  
Key: 0, Value: 165987  
Key: 1, Value: 167426  
Key: 2, Value: 166788  
  
cHashMap integrity check:  
Expected size: 3, Actual size: 30  
Key: 0, Value: 166265  
Key: 1, Value: 166356  
Key: 2, Value: 166625
```

Рис. 7 Результат работы коллекций

```
Execution times:  
    HashMap: 0,032 s,  
    Hashtable: 0,307 s,  
    SyncMap: 0,285 s,  
    ConcurrentHashMap: 0,055 s.
```

Рис. 8 Сравнение скорости работы коллекций

Лабораторная работа 5

Задание

Необходимо:

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

Реализация

В данной лабораторной работе был реализован семафор с применением ReentrantLock, а так же без него. ReentrantLock обеспечивает одному и тому же потоку возможность несколько раз захватывать разрешение на работу с общими данными.

В лабораторной работе использовалось 4 потока и 2 разрешения на работу с общими данными. Как видно из результатов, программа успешно справилась с работой.

Результат

```
-----  
Regular semaphore:  
-----  
Поток pool-1-thread-4 работает. Активных потоков: 2  
Поток pool-1-thread-2 работает. Активных потоков: 1  
Поток pool-1-thread-1 работает. Активных потоков: 2  
Поток pool-1-thread-3 работает. Активных потоков: 1  
Максимальное количество активных потоков: 2  
-----  
My semaphore:  
-----  
Поток pool-2-thread-1 работает. Активных потоков: 1  
Поток pool-2-thread-2 работает. Активных потоков: 2  
Поток pool-2-thread-3 работает. Активных потоков: 1  
Поток pool-2-thread-4 работает. Активных потоков: 2  
Максимальное количество активных потоков: 2  
-----  
My semaphore without lock:  
-----  
Поток pool-3-thread-1 работает. Активных потоков: 1  
Поток pool-3-thread-2 работает. Активных потоков: 2  
Поток pool-3-thread-4 работает. Активных потоков: 1  
Поток pool-3-thread-3 работает. Активных потоков: 2  
Максимальное количество активных потоков: 2
```

Рис. 9 Результат работы программы в Лабораторной 5

Лабораторная работа 6

Задание

Необходимо создать клиент-серверное приложение:

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

Необходимо реализовать 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.

Реализация

В рамках задачи выполнялось ознакомление и работа с Java IPC. Созданы клиент и сервер, которые обмениваются между собой сообщениями через socket. Сервер - смотрит кто подключился к определенному ip:port и добавляет новых пользователей, а затем отслеживает их действия. Клиент – отправляет сообщения на сервер.

Как видно из результатов, процессы-клиенты и сервер взаимодействуют корректно, выводя сообщения.

Результат

```
Server is running and waiting for connections...
New client connected: Socket[addr=/127.0.0.1,port=54297,localport=12347]
User user 1 connected.
[user 1]: hi!
New client connected: Socket[addr=/127.0.0.1,port=54332,localport=12347]
User user 2 connected.
[user 2]: hi!
[user 1]: How are you?
```

Рис. 10 Результат работы сервера

```
Connected to the chat server!  
Enter your username:  
user 1  
Welcome to the chat, user 1!  
Type Your Message  
hi!  
[user 2]: hi!  
How are you?  
|
```

Рис. 12 Результат работы первого пользователя

```
Connected to the chat server!  
Enter your username:  
user 2  
Welcome to the chat, user 2!  
Type Your Message  
hi!  
[user 1]: How are you?
```

Рис. 12 Результат работы второго пользователя

Лабораторная работа 7

Задание

Необходимо:

1. Изучить библиотеку mappedbus;
2. Запустить готовые тестовые примеры.

Реализация

В данной программе был рассмотрен пример для межпроцессорного взаимодействия. `ObjectWriter` пишет данные в memory-mapped file, а `ObjectReader` – читает данные из этого файла.

Как видно из результатов, у нас есть 2 `ObjectWriter` которые пишут в общий файл, а `ObjectReader` читает эти данные.

Результат

```
Read: PriceUpdate [source=1, price=16, quantity=32], hasRecovered=false
Read: PriceUpdate [source=0, price=0, quantity=0], hasRecovered=false
Read: PriceUpdate [source=1, price=18, quantity=36], hasRecovered=false
Read: PriceUpdate [source=0, price=2, quantity=4], hasRecovered=false
Read: PriceUpdate [source=1, price=20, quantity=40], hasRecovered=false
Read: PriceUpdate [source=0, price=4, quantity=8], hasRecovered=false
Read: PriceUpdate [source=1, price=22, quantity=44], hasRecovered=true
Read: PriceUpdate [source=0, price=6, quantity=12], hasRecovered=true
Read: PriceUpdate [source=1, price=24, quantity=48], hasRecovered=true
Read: PriceUpdate [source=0, price=8, quantity=16], hasRecovered=true
```

Рис. 13 Результат работы программы