Project 1: Problem 1

소프트웨어학부 20204898 박소은

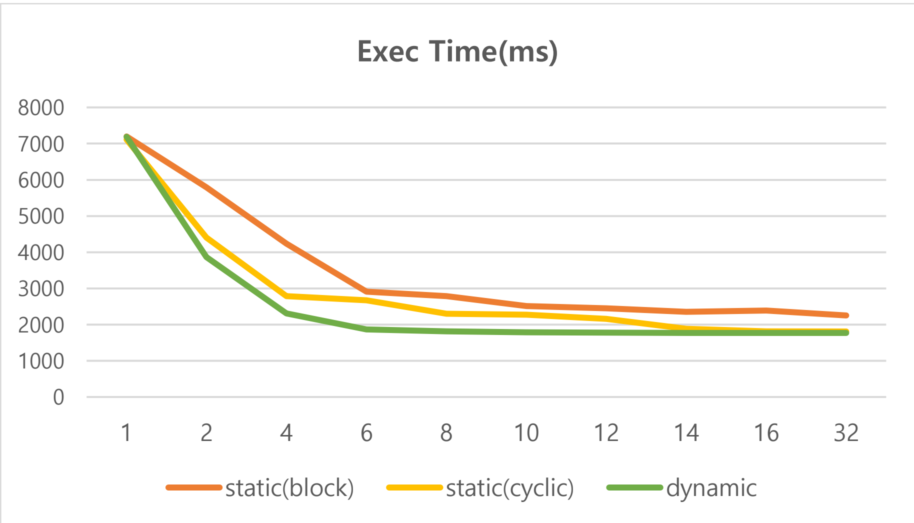
Environment

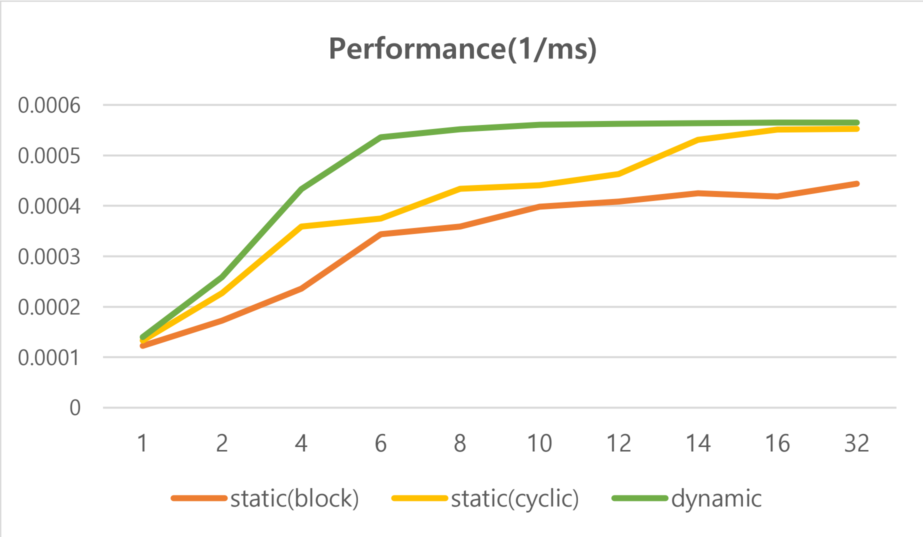
- **Processor**: Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
- **Number of cores**: 4
- **RAM**: 16.0GB(15.8GB available)
- **OS**: Windows 11 (64 bit)

Tables and graphs

Exec time	1	2	4	6	8	10	12	14	16	32
static(block)	7198	5791	4241	2911	2788	2511	2447	2352	2390	2253
static(cyclic)	7114	4404	2787	2669	2304	2270	2161	1885	1814	1810
dynamic	7188	3867	2308	1865	1813	1784	1778	1774	1770	1770

Performance	1	2	4	6	8	10	12	14	16	32
static(block)	0.000122579	0.00017	0.00024	0.00034	0.000359	0.000398	0.000409	0.000425	0.000418	0.000444
static(cyclic)	0.000133103	0.00023	0.00036	0.00037	0.000434	0.000441	0.000463	0.000531	0.000551	0.000552
dynamic	0.000139958	0.00026	0.00043	0.00054	0.000552	0.000561	0.000562	0.000564	0.000565	0.000565





Explanation / Analysis

As a result of running the program in a Quadcore environment, dynamic load balancing performed the best, followed by cyclic decomposition(static load balancing) and block decomposition(static load balancing).

In the case of static load balancing using block decomposition, threads are assigned consecutive numbers. For example, thread 1 computes {0 ~ 49999}, thread 2 computes {50000 ~ 99999}, thread 3 computes {100000 ~ 149999}, and the last thread computes {150000 ~ 199999}. Due to the nature of the program(computing the number of prime numbers), small numbers end quickly and large numbers take a long time, therefore threads which calculate large numbers end later than other threads. That means, load balancing is not good.

```
          < RESULT >
0 Thread: 811 ms
1 Thread: 2160 ms
2 Thread: 2988 ms
3 Thread: 4083 ms

Total Program Execution Time: 4086ms
1... 199999 prime# counter=17984
```

Thread 3 took 3272ms more than thread 0. Thread 0 finished in 811ms, but the program has to wait until thread 3 is over. Due to the load balancing problem, thread 0 remains idle.

In the case of static load balancing using cyclic decomposition, the task size is set to 10, and threads take turns calculating 10 consecutive numbers. Threads perform better than the block composition program because they calculate evenly from small to large numbers, which means load balance is better.

```
          < RESULT >
0 Thread: 2669 ms
1 Thread: 2799 ms
2 Thread: 2628 ms
3 Thread: 2417 ms

Total Program Execution Time: 2836ms
1... 199999 prime# counter=17986
```

According to the result, the execution time of the four threads is about 2400 to 2700ms, and the execution time is not much different. Total execution time is also better than the block composition.

In the best-performing dynamic load balancing, threads are assigned a number to calculate at runtime. The number to be calculated was managed by the IndexGenerator object. Threads get the number to be calculated by generateIndex() method, and the method is protected by 'synchronized' keyword, preventing the numbers from overlapping.

```
          < RESULT >
0 Thread: 2644 ms
1 Thread: 2643 ms
2 Thread: 2645 ms
3 Thread: 2644 ms

Total Program Execution Time: 2646ms
1... 199999 prime# counter=17982
```

Total execution time is slightly better than cyclic decomposition. The execution time of each thread was about 2,640ms, so the load balance was very good. It is because threads have few idle time, as they get numbers to calculate at runtime, working busy.

Java source code

Static (Block)

```
public class pc_static_block {
    private static int NUM_END = 200000; // default input
    private static int NUM_THREADS = 1; // default number of threads

    public static void main(String[] args) {
        if (args.length == 1) {
            NUM_THREADS = Integer.parseInt(args[0]);
        }

        int blockSize = (int) Math.ceil(NUM_END / NUM_THREADS);
        int totalCounter = 0;
        BlockThread[] threads = new BlockThread[NUM_THREADS];

        long startTime = System.currentTimeMillis();    // program execution time starts

        // start threads
        for(int i=0; i<NUM_THREADS; i++) {
            int end;
            if (i == NUM_THREADS-1) {
                end = NUM_END;  // if thread[i] is last thread: end number is NUM_END
            } else {
                end = i*blockSize + blockSize;  // if thread[i] is not last thread:
calculate by blockSize
            }

            threads[i] = new BlockThread(i*blockSize, end);
            threads[i].start();
        }

        // Thread join()
        for (int i=0; i<NUM_THREADS; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {}
        }

        // Get the total number of prime numbers
        for (int i=0; i<NUM_THREADS; i++) {
            totalCounter += threads[i].counter;
        }

        long endTime = System.currentTimeMillis();      // program execution time ends
        long timeDiff = endTime - startTime;

        // print the result
        System.out.println("\n          < RESULT > ");
        for (int i=0; i<NUM_THREADS; i++) {
            System.out.println(i +" Thread: " + threads[i].timeDiff + " ms");
        }
        System.out.println("\nTotal Program Execution Time: " + timeDiff + "ms");
        System.out.println("1... " + (NUM_END - 1) + " prime# counter=" + totalCounter +
"\n");
    }
```

```
}

class BlockThread extends Thread {
    int start, end, counter;
    long timeDiff;

    /*
       BlockThread tests whether the number is a prime number or not
       from 'start' number ~ to 'end' number
     */

    BlockThread(int start, int end) {
        this.start = start;
        this.end = end;
        this.counter = 0;
    }

    @Override
    public void run() {
        long startTime = System.currentTimeMillis();
        for (int num=start; num<end; num++) {
            if(isPrime(num))    counter++;
        }
        long endTime = System.currentTimeMillis();
        timeDiff = endTime - startTime;
    }

    private static boolean isPrime(int x) {
        if (x<=1)    return false;
        for(int i=2; i<x; i++) {
            if (x%i == 0)   return false;
        }
        return true;
    }
}
```

Static (Cyclic)

```
public class pc_static_cyclic {
    private static int NUM_END = 200000; // default input
    private static int NUM_THREADS = 1; // default number of threads
    private static int TASK_SIZE = 10;  // default task size

    public static void main(String[] args) {
        if (args.length == 1) {
            NUM_THREADS = Integer.parseInt(args[0]);
        }

        int totalCounter = 0;
        CyclicThread[] threads = new CyclicThread[NUM_THREADS];
        long startTime = System.currentTimeMillis();    // program execution time starts

        // Run threads
        for(int i=0; i<NUM_THREADS; i++) {
            threads[i] = new CyclicThread(i, NUM_END, NUM_THREADS);
            threads[i].start();
```

```java
        }

        // Thread join()
        for (int i=0; i<NUM_THREADS; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {}
        }

        // Get the total number of prime numbers
        for (int i=0; i<NUM_THREADS; i++) {
            totalCounter += threads[i].counter;
        }
        long endTime = System.currentTimeMillis();      // program execution time ends
        long timeDiff = endTime - startTime;

        // print the result
        System.out.println("\n         < RESULT > ");
        for (int i=0; i<NUM_THREADS; i++) {
            System.out.println(i +" Thread: " + threads[i].timeDiff + " ms");
        }
        System.out.println("\nTotal Program Execution Time: " + timeDiff + "ms");
        System.out.println("1... " + (NUM_END - 1) + " prime# counter=" + totalCounter +
"\n");
    }
}

class CyclicThread extends Thread {
    private static int TASK_SIZE = 10;  // default task size
    int threadIndex, NUM_END, NUM_THREADS, counter;
    long timeDiff;

    /*
        A CyclicThread get the number to calculate by its 'threadIndex'
        if 'threadIndex' == 0 and NUM_THREADS == 4:
            it starts calculating from 0 to 9,
            the next number to calculate is (0 + TASK_SIZE*NUM_THREADS),
            therefore calculates 40 ~ 49,
            and then 80 ~ 89 ... and so on.
     */

    CyclicThread(int threadIndex, int NUM_END, int NUM_THREADS) {
        this.threadIndex = threadIndex;
        this.NUM_END = NUM_END;
        this.NUM_THREADS = NUM_THREADS;
        this.counter = 0;
    }

    @Override
    public void run() {
        long startTime = System.currentTimeMillis();

        /*
            'num' is a starting number to calculate.
            If num is 0, start calculating from 0,
            If num is 3, start calculating from 30...
            (because the TASK_SIZE is 10)
         */
```

```
        int num = threadIndex * TASK_SIZE;
        while (num <= NUM_END) {
            for(int i=num; i<num+TASK_SIZE; i++) {
                if(isPrime(i))    counter++;
            }
            num += NUM_THREADS * TASK_SIZE;
        }

        long endTime = System.currentTimeMillis();
        timeDiff = endTime - startTime;
    }

    private static boolean isPrime(int x) {
        if (x<=1)    return false;
        for(int i=2; i<x; i++) {
            if (x%i == 0)   return false;
        }
        return true;
    }
}
```

Dynamic

```
public class pc_dynamic {
    private static int NUM_END = 200000; // default input
    private static int NUM_THREADS = 1; // default number of threads
    private static int TASK_SIZE = 10;  // default task size

    public static void main(String[] args) {
        if (args.length == 1) {
            NUM_THREADS = Integer.parseInt(args[0]);
        }

        int totalCounter = 0;
        DynamicThread[] threads = new DynamicThread[NUM_THREADS];
        IndexGenerator indexGenerator = new IndexGenerator();   // threads share the
indexGenerator

        long startTime = System.currentTimeMillis();    // program execution time starts

        // Start threads
        for(int i=0; i<NUM_THREADS; i++) {
            threads[i] = new DynamicThread(indexGenerator, NUM_END, NUM_THREADS);
            threads[i].start();
        }

        // Thread join()
        for (int i=0; i<NUM_THREADS; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {}
        }

        // Get the total number of prime numbers
        for (int i=0; i<NUM_THREADS; i++) {
```

```
            totalCounter += threads[i].counter;
        }

        long endTime = System.currentTimeMillis();      // program execution time ends
        long timeDiff = endTime - startTime;

        // print the result
        System.out.println("\n          < RESULT > ");
        for (int i=0; i<NUM_THREADS; i++) {
            System.out.println(i +" Thread: " + threads[i].timeDiff + " ms");
        }
        System.out.println("\nTotal Program Execution Time: " + timeDiff + "ms");
        System.out.println("1... " + (NUM_END - 1) + " prime# counter=" + totalCounter +
"\n");
    }
}
class DynamicThread extends Thread {
    private static int TASK_SIZE = 10;  // default task size
    int NUM_END, NUM_THREADS, counter, num;
    long timeDiff;
    IndexGenerator indexGenerator;

    DynamicThread(IndexGenerator indexGenerator, int NUM_END, int NUM_THREADS) {
        this.indexGenerator = indexGenerator;
        this.NUM_END = NUM_END;
        this.NUM_THREADS = NUM_THREADS;
        this.counter = 0;
    }

    @Override
    public void run() {
        long startTime = System.currentTimeMillis();
        num = indexGenerator.generateIndex();   // IndexGenerator gives the number to
calculate

        while (num <= NUM_END) {
            for(int i=num; i<num+10; i++) {
                if(isPrime(i))    counter++;
            }
            num = indexGenerator.generateIndex();
        }

        long endTime = System.currentTimeMillis();
        timeDiff = endTime - startTime;
    }

    private static boolean isPrime(int x) {
        if (x<=1)    return false;
        for(int i=2; i<x; i++) {
            if (x%i == 0)   return false;
        }
        return true;
    }
}

class IndexGenerator {
    public static int index = 0;
```

```java
    /*
        IndexGenerator stores the last number the threads have calculated.
        Threads can get the number by 'generateIndex()'
        'synchronized' keyword is used to make the function a critical section
        because 'index' variable should be protected when one thread is accessing 'index'
     */

    public synchronized int generateIndex() {
        index += 10;
        return index;
    }
}
```

Screen capture image of program execution and output

- pc_static_block

```
C:\Users\82104\OneDrive-CAU\4-1\멀티코어\Multicore_programming\Project1_Lab\src>javac pc_static_block.java

C:\Users\82104\OneDrive-CAU\4-1\멀티코어\Multicore_programming\Project1_Lab\src>java pc_static_block 4

          < RESULT >
0 Thread: 775 ms
1 Thread: 1855 ms
2 Thread: 2776 ms
3 Thread: 3563 ms

Total Program Execution Time: 3564ms
1... 199999 prime# counter=17984
```

- pc_static_cyclic

```
C:\Users\82104\OneDrive-CAU\4-1\멀티코어\Multicore_programming\Project1_Lab\src>javac pc_static_cyclic.java

C:\Users\82104\OneDrive-CAU\4-1\멀티코어\Multicore_programming\Project1_Lab\src>java pc_static_cyclic 8

          < RESULT >
0 Thread: 1794 ms
1 Thread: 1817 ms
2 Thread: 1866 ms
3 Thread: 1826 ms
4 Thread: 1858 ms
5 Thread: 1738 ms
6 Thread: 1834 ms
7 Thread: 1840 ms

Total Program Execution Time: 1867ms
1... 199999 prime# counter=17986
```

- pc_dynamic

```
C:\Users\82104\OneDrive-CAU\4-1\멀티코어\Multicore_programming\Project1_Lab\src>javac pc_dy
namic.java

C:\Users\82104\OneDrive-CAU\4-1\멀티코어\Multicore_programming\Project1_Lab\src>java pc_dyn
amic 16

          < RESULT >
0 Thread: 1813 ms
1 Thread: 1812 ms
2 Thread: 1807 ms
3 Thread: 1815 ms
4 Thread: 1810 ms
5 Thread: 1814 ms
6 Thread: 1779 ms
7 Thread: 1763 ms
8 Thread: 1812 ms
9 Thread: 1811 ms
10 Thread: 1742 ms
11 Thread: 1674 ms
12 Thread: 1813 ms
13 Thread: 1778 ms
14 Thread: 1606 ms
15 Thread: 1488 ms

Total Program Execution Time: 1818ms
1... 199999 prime# counter=17982
```

How to compile and execute the source code

- **Static (Block)**
  - Compilation: `$ javac pc_static_block.java`
  - Execution

    Default: `$ java pc_static_block`

    `N Threads: $ java pc_static_block N`
- **Static (Cyclic)**
  - Compilation: `$ javac pc_static_cyclic.java`
  - Execution

    Default: `$ java pc_static_cyclic`

    `N Threads: $ java pc_static_cyclic N`
- **Dynamic**

- Compilation: `$ javac pc_dynamic.java`
- Execution

  Default: `$ java pc_dynamic`

  `N Threads: $ java pc_dynamic N`