Anthony Borza

Professor Tarquinio, Michael

Due Date: September 10, 2017

Homework 3

**Table of Contents**

[Objective: 3](#_Toc492979861)

[IOIntensive Class: 3](#_Toc492979862)

[Computationally Class: 3](#_Toc492979863)

[Controller Class: 3](#_Toc492979864)

[Time Class: 3](#_Toc492979865)

[Tables: 3](#_Toc492979866)

[Screenshots: 5](#_Toc492979867)

[IOIntensive: 5](#_Toc492979868)

[IOIntensive Times: 6](#_Toc492979869)

[Computationally: 7](#_Toc492979870)

[Computationally Times: 8](#_Toc492979871)

[Computationally before IOIntensive Times: 9](#_Toc492979872)

[IOIntensive Times before Computationally: 10](#_Toc492979873)

[Lessons Learned: 12](#_Toc492979874)

[Was there variance run to run? 12](#_Toc492979875)

[What is the differences between CPU and IO bound threads? 12](#_Toc492979876)

[What is the differences in average wait time between IO and CPU bound threads, etc.? 12](#_Toc492979877)

[References: 13](#_Toc492979878)

# **Objective:**

The purpose of this assignment is to demonstrate an understanding of the various issues around the OS scheduler. The following classes used were: IOIntensive, Computationally, Controller, and Time Class.

## **IOIntensive Class:**

The IOIntensive class is designed to iterate through five threads 100 times, starting at 0 and ending at 100. The threads are created using an array. Each thread must start, and then finish before the next thread can start. At the end of each thread, the total time taken in nanoseconds to complete that that thread will be displayed.

## **Computationally Class:**

The Computationally class is designed to iterate through five threads 100 times, starting at 0 and ending at 100. The threads are created using an array. Each thread must start, and then finish before the next thread can start. At the end of each thread, the sum of all numbers from 0 to 100 will be printed, followed by the total time taken in nanoseconds to complete that thread.

## **Controller Class:**

The Controller class is designed to run the program by calling the IOIntensive Class, and Computationally Class. This class is also uses to calculate the wait times for each thread, the average wait times for all threads, and the total run time for all the IOIntensive threads, and Computationally.

**Time Class:**

The Time class is an abstract class. It holds the variables: time, startTime, and methods getTime(), and getStartTime(), to calculate the times for the threads. These variables and methods can be accessed by all classes because they are in the same package.

# **Tables:**

Below is a representation of the wait times for each thread, average wait times, and total runtime for all threads. These times do change every time the program runs.

|  |  |  |
| --- | --- | --- |
| IOIntensive Thread Times  (First Run) | | |
| IOIntensive Threads | **Total Time** | **Total Wait Times** |
| Thread 1: | 2778634(Nano) | 148479(Nano) |
| Thread 2: | 2177189(Nano) | 105712(Nano) |
| Thread 3: | 1924202(Nano) | 93063(Nano) |
| Thread 4: | 1623630(Nano) | 111435(Nano) |
| Thread 5: | 1501956(Nano) | 110531(Nano) |
|  | **Overall Runtime**  10005611(Nano) | **Average Wait Time for All Threads**  113844(Nano) |

|  |  |  |
| --- | --- | --- |
| IOIntensive Thread Times  (Second Run) | | |
| IOIntensive Threads | **Total Time** | **Total Wait Times** |
| Thread 1: | 2853626(Nano) | 179199(Nano) |
| Thread 2: | 3273764(Nano) | 161429(Nano) |
| Thread 3: | 2832544(Nano) | 194859(Nano) |
| Thread 4: | 2134723(Nano) | 192450(Nano) |
| Thread 5: | 1886254(Nano) | 264130(Nano) |
|  | **Overall Runtime**  12980911(Nano) | **Average Wait Time for All Threads**  198413(Nano) |

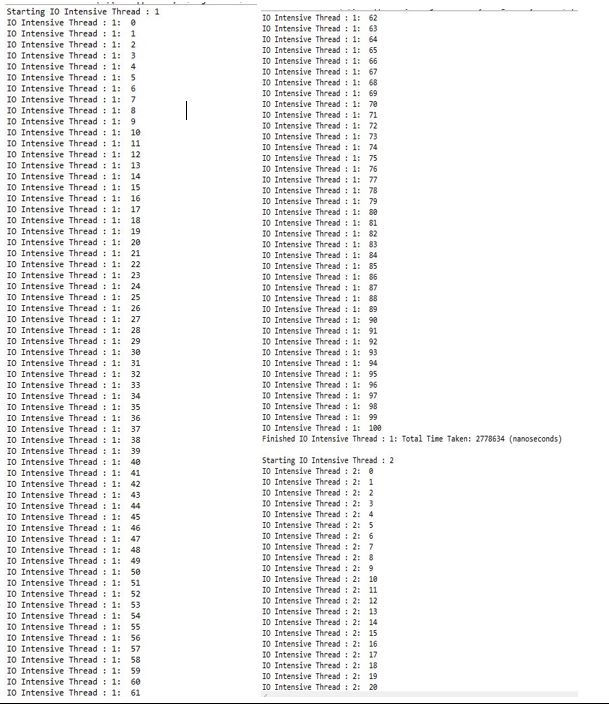
|  |  |  |
| --- | --- | --- |
| Computationally Thread Times  (Second Run) | | |
| Computationally Threads | **Total Time** | **Total Wait Times** |
| Thread 1: | 1965162(Nano) | 133722(Nano) |
| Thread 2: | 1596826(Nano) | 136130(Nano) |
| Thread 3: | 1322154(Nano) | 137336(Nano) |
| Thread 4: | 1420036(Nano) | 101797(Nano) |
| Thread 5: | 1174880(Nano) | 98183(Nano) |
|  | **Overall Runtime**  7479058(Nano) | **Average Wait Time for All Threads**  121433(Nano) |

|  |  |  |
| --- | --- | --- |
| Computationally Thread Times  (First Run) | | |
| Computationally Threads | **Total Time** | **Total Wait Times** |
| Thread 1: | 1168556(Nano) | 97882(Nano) |
| Thread 2: | 1709164(Nano) | 124686(Nano) |
| Thread 3: | 822507(Nano) | 102098(Nano) |
| Thread 4: | 836963(Nano) | 106315(Nano) |
| Thread 5: | 805942(Nano) | 106315(Nano) |
|  | **Overall Runtime**  5343132(Nano) | **Average Wait Time for All Threads**  5941867(Nano) |

# **Screenshots:**

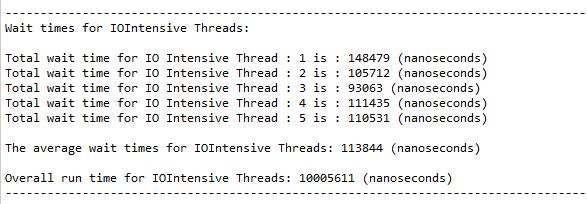
## **IOIntensive:**

The following image below shows IOIntensive thread 1 starting, and then when it is finished. It this shows the time taken to complete that thread. The image also shows IOIntensive thread 2 starting after IOIntensive thread 1 finished. This does the same for the next three IOIntensive threads.

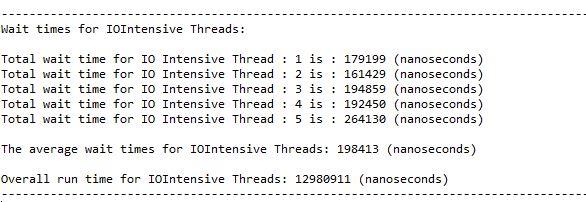


## **IOIntensive Times:**

The images below show the wait times for the IOIntensive threads 1 to 5 in nanoseconds followed by the average wait time for all threads. It also shows the total run time for all IOIntensive threads. This is just one example of the times. The times can change every time the program is run.

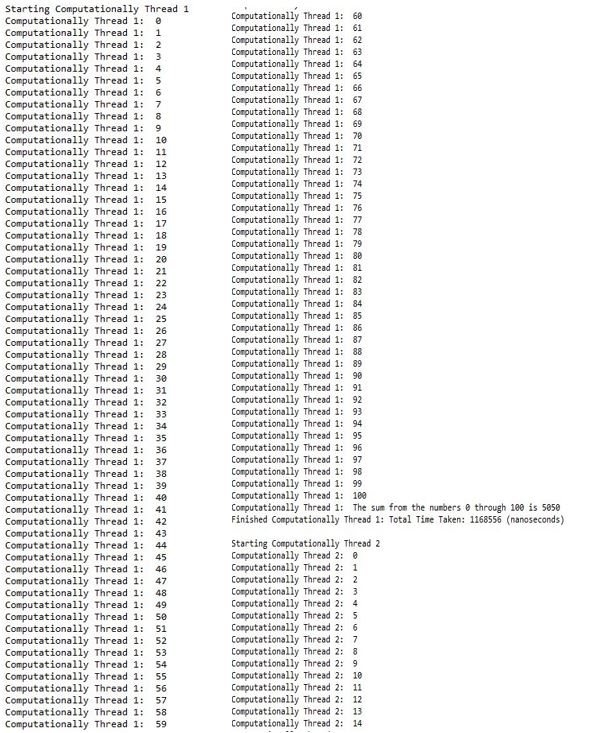


Notice how the times change after running the program a second time



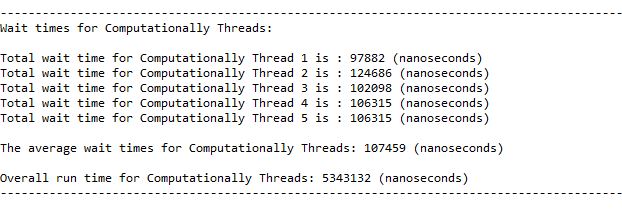
## **Computationally:**

The following image below shows Computationally thread 1 starting, and then when it is finished. Before the thread is finished it performs a mathematical computation of the sum of all numbers for 0 to 100. It this shows the time taken to complete that thread. The image also shows Computationally thread 2 starting after IOIntensive thread 1 finished. This does the same for the next three IOIntensive threads.

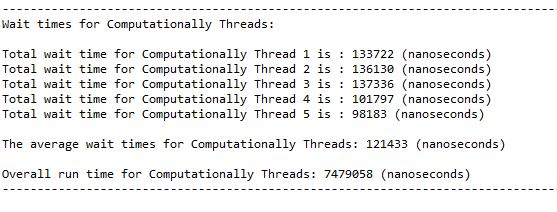


## **Computationally Times:**

The images below show the wait times for the Computationally threads 1 to 5 in nanoseconds, followed by the average wait time for all threads. It also shows the total run time for all Computationally threads. This is just one example of the times. The times can change every time the program is run.

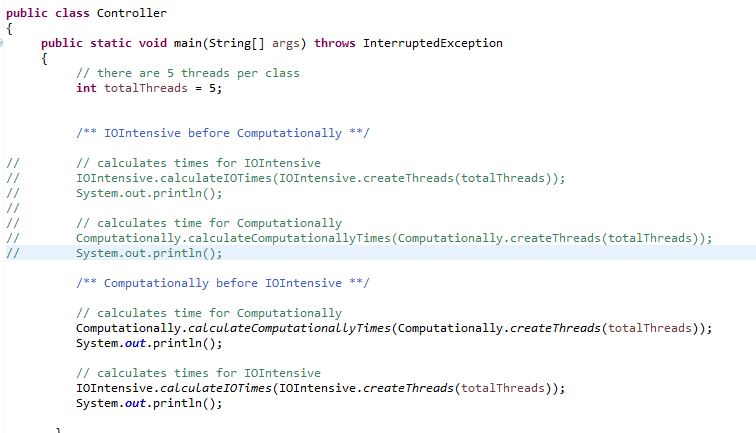


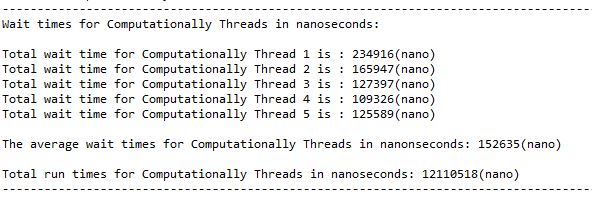
Notice how the times change after running the program a second time

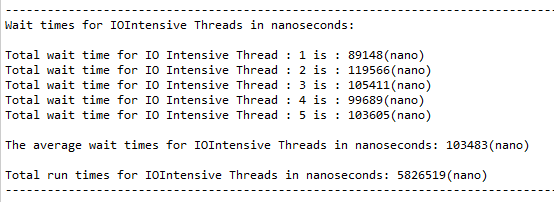


## **Computationally before IOIntensive Times:**

For this is work, you must comment out the following section in the Controller class:



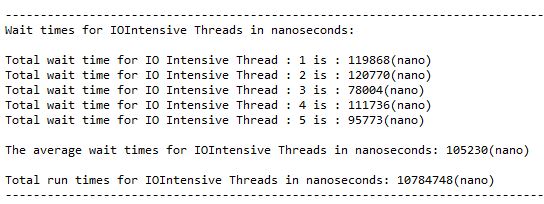


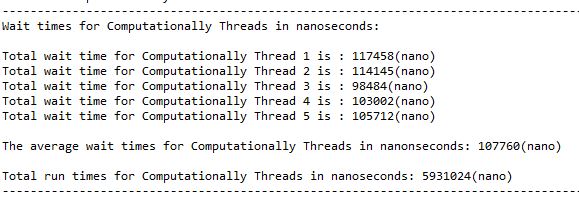


## **IOIntensive Times before Computationally:**

For this is work, you must comment out the following section in the Controller class:







# **Lessons Learned:**

## **Was there variance run to run?**

The CPU uses the FCFS algorithm to execute the processes (threads) in the order they arrive. The FCFS algorithm is a very efficient processing scheduling algorithm that saves valuable CPU resources. The variances did change every time the program was run. I think the reason they changed is because it is not an algorithm that is used to check for priority for its processes (threads). With that being said, since there is no priority, the processes (threads) are dispatched accordingly to their arrival time; and therefore, the times will change almost every time the program is run.

## **What is the differences between CPU and IO bound threads?**

The CPU bound is bottlenecked by the CPU, and is responsible for performing calculations; whereas, IO bound threads, are bottlenecked by the IO, and are responsible for completing task that process data from a disk.

## **What is the differences in average wait time between IO and CPU bound threads, etc.?**

The main differences are that when running the IO threads before the Computationally threads the wait time for the IO threads were longer. When running the Computationally threads before the IO threads, the average wait time is longer for the Computationally threads. I think that the average wait time varies between IO and CPU bound threads because it depends on whether there are available resources.

# **References:**

GeeksforGeeks. (2017). Program for FCFS Scheduling | Set 1. Retrieved from

<http://www.geeksforgeeks.org/program-fcfs-scheduling-set-1/>

Understanding Thread Priority. (2017). Retrieved from:

<http://leo.ugr.es/elvira/devel/Tutorial/Java/essential/threads/priority.html>