DOORS OS Project

Dennis Lin,

Phonepaseuth Viengxay,

Ricky Viengxay

Patrick Bobbie

CS 3502

April 23, 2018

**Table of Contents**

Introduction

Design Approach

Implementation Classes

Driver Class

Driver(Run)

Loader Class

Loader(Load)

Process Control Block

PCB Class

Memory System

MMU Class

Central Processing Unit

CPU

Schedulers

LongTermScheduler Class

LongTermScheduler()

Short Term Scheduler Class

ShortTermScheduler()

Dispatcher Class

Conclusion

FIFO Statistics

FIFO Table Statsitics

Priority Statistics

Priority Table Statistics

SJF Statistics

SJF Table Statistics

I/O’s per Job

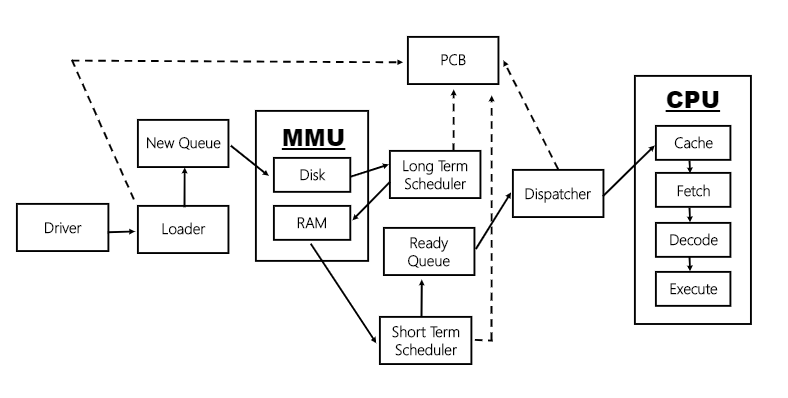
Pages Used per Job

Introduction

The purpose of this document is to inform you on our ideas and thought process on how we tackled this Operating Systems project. As a group, we collectively decided the most optimal programming language to use for this project was C++, because of its strong pointer functionalities and flexibility. To work around conflicting schedules, we decided to use GitHub, as our source to share and modify files. Our choice of IDE was CLion. When possible, we as a team would find a day out of the week to come together and discuss our progress, problems, and future work that needed to be done. Some of the difficulties faced with this project include communication throughout, as many of us had different time schedules, interpreting some of the instructions which were provided, and finding the most optimal design to get the OS working properly.

# Design Approach

Our team first reviewed the project specification block diagram that was provided to us to understand how our system would work cohesively together. During phase one, we went through multiple trial and errors with our schematics design, which caused us to continuously revise our approach to our operating system. As we built the operating system, we would find flaws within our approach and discover new approaches that would not only resolve our problems but make it more efficient. As a result, the continuous revisions would solidify our schematics design for our operating system.

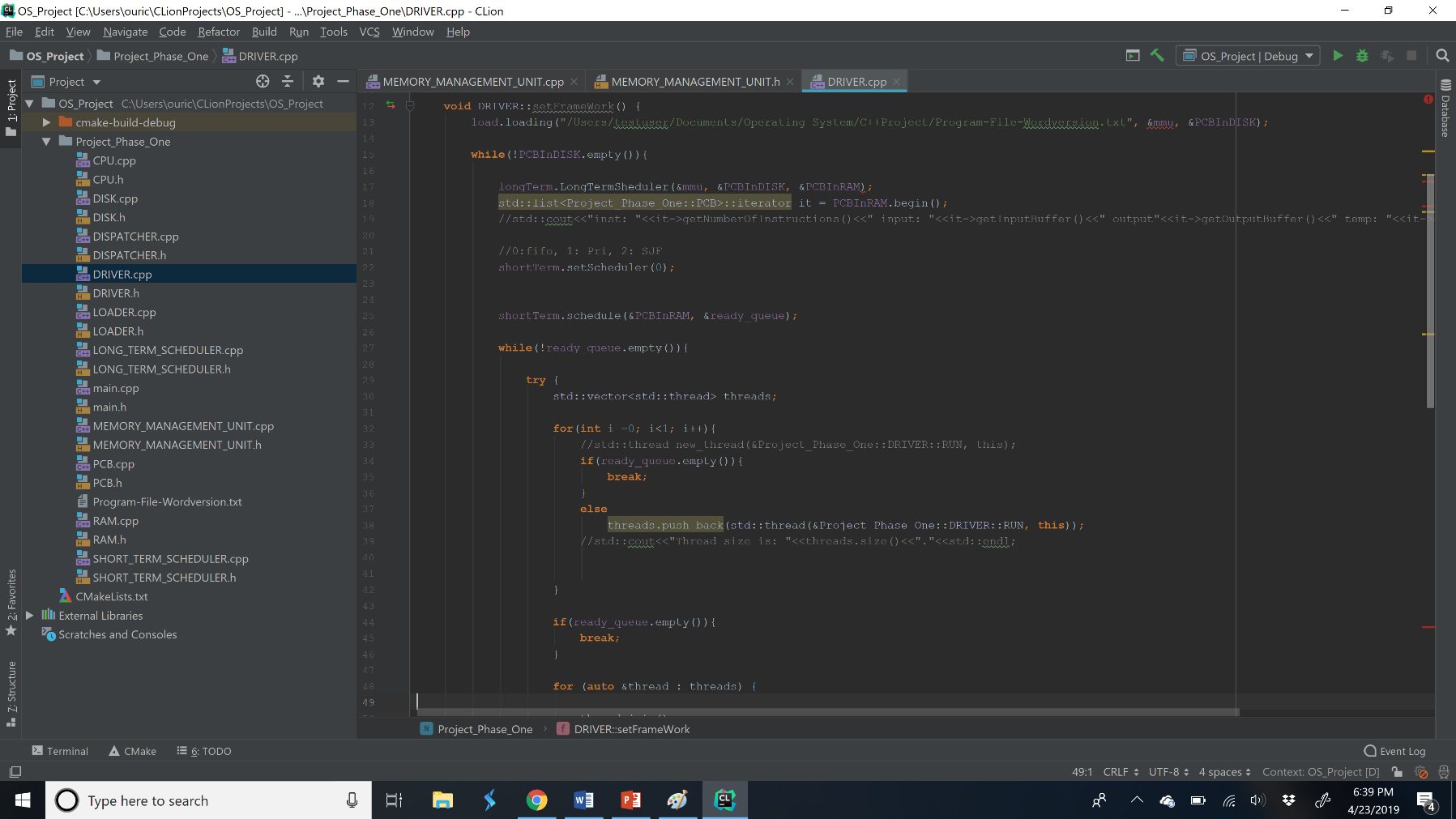


The main components that we pulled out were the CPU, the Short Term Scheduler, the Long Term Scheduler, the MMU, the RAM, the Loader, the Disk, and the Driver. We established the driver as the “engine” of the overall process. From there, we ordered the other components in a flow that takes the information provided in the instruction set and loads them onto the disk through the loader component. From there the disk will be accessed by the Long-Term Scheduler and jobs will be loaded to RAM. After jobs have been loaded to RAM, the MMU, referred to in the Phase 1 Document as Memory, accesses the jobs needed to be scheduled by the Short Term Scheduler which then utilizes the dispatcher to send jobs to the CPU. The CPU then loops a three part cycle of fetch, decode, and execute until it has run out of instructions at which point, the OS terminates.

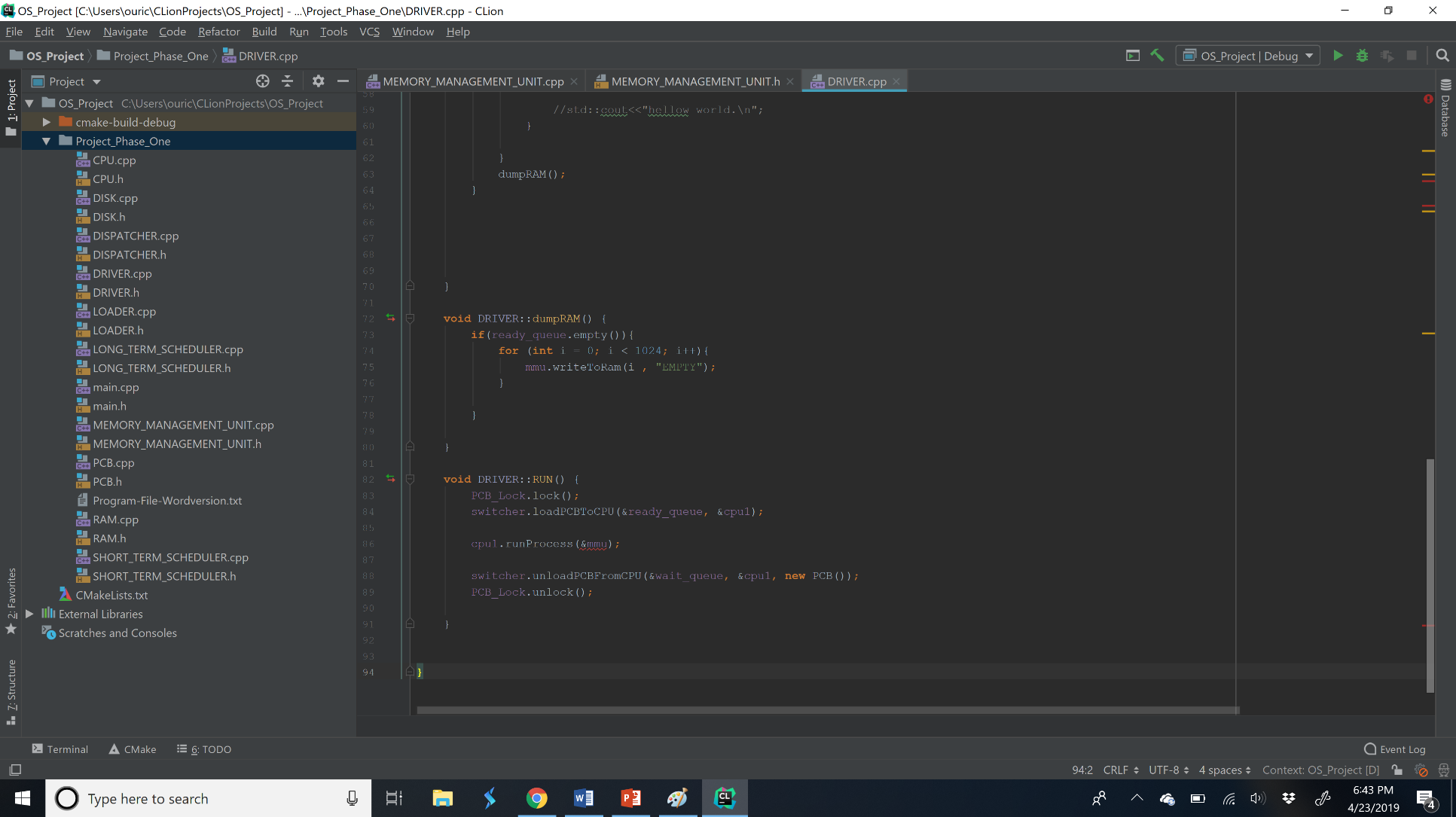
For the multiprocessing portion of our OS, we needed to look at three areas: making support for multiple cores, memory management amongst the processors, and the cache of each processor. In our design approach we took note of this and decided to implement the CPU module to handle all of these items as a single processor and then using threads to handle multiple cores. This approach makes it much easier to scale the CPU cores when appropriate.

For phase 2, we went back and reevaluated the design we had for phase 1. In doing this we addressed areas of concern for the parts that we would implement in phase 2. The general architecture idea hasn’t really changed from phase 1. The Major changes came in the form of separating the Dispatcher into its own individual unit and placing it between the ShortTermScheduler and CPU modules, and the implementation of the cache and paging systems.

## Driver Class

Our driver class has access to all classes and runs when our main class is executed. Once the driver is called from our main class, it then calls our Loader class which will load instructions onto our disk and writes the new PCB into the new PCB que. 

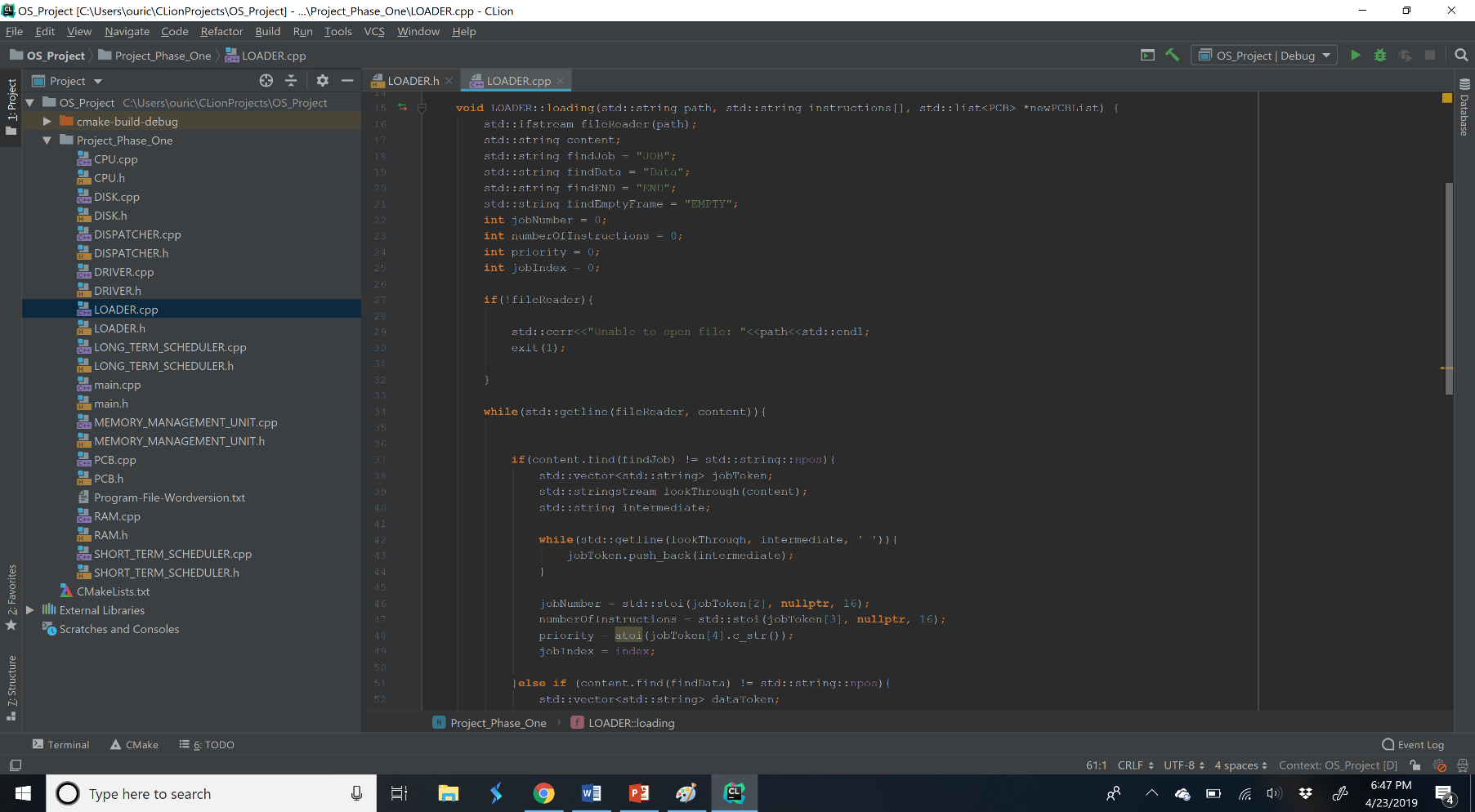
Driver ()



### Driver(run)

## Loader Class

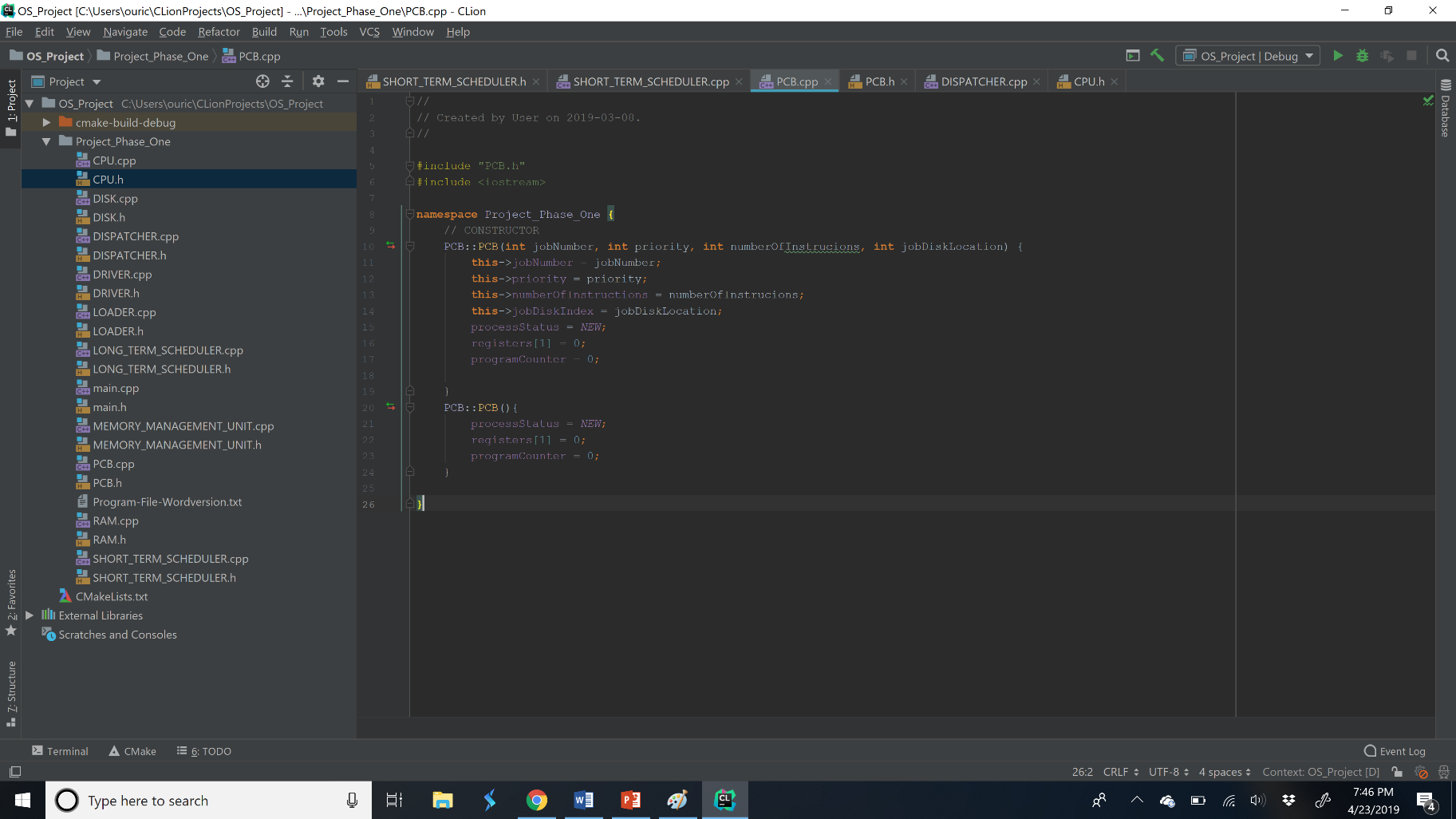
The loader will take in the string for a file path, a pointer MMU, and a pointer to a new PCB list. The first thing that it will do is open a file and searches the first line for the word job. If it finds the word the “JOB” it will push 5 intermediates into a vector. The index in vector 2 will be the job number, the index of vector 3 will be job numbers, and priority will be in vector index 4. The index in disk where the instructions start, will be the current index and will continue to load instructions and increments the index by 1 when instructions are loaded onto the disk. The loader continues to run until it finds the word, “DATA” and will proceed with the same process as JOB, except at index 2 is an INPUT BUFFER, 3 is OUTPUT BUFFER, 4 is TEMP BUFFER. After this it will create a PCB object using job number, priority, number of priorities, and job index and this new PCB will be added to the new PCB list via pointer. If the loader finds the word “END” it does not do anything and will continue to add instructions to the disk.



### Loader(Load)

## Process Control Block

### PCB Class

Our PCB class, is a data structure which, contains all the information needed for each job in the CPU. This information includes, job number, priority, burst time, cache information, etc. Our PCB class has a constructor which takes in jobNumber, priority, numberofInstructions, and the jobDiskLocation for later use.

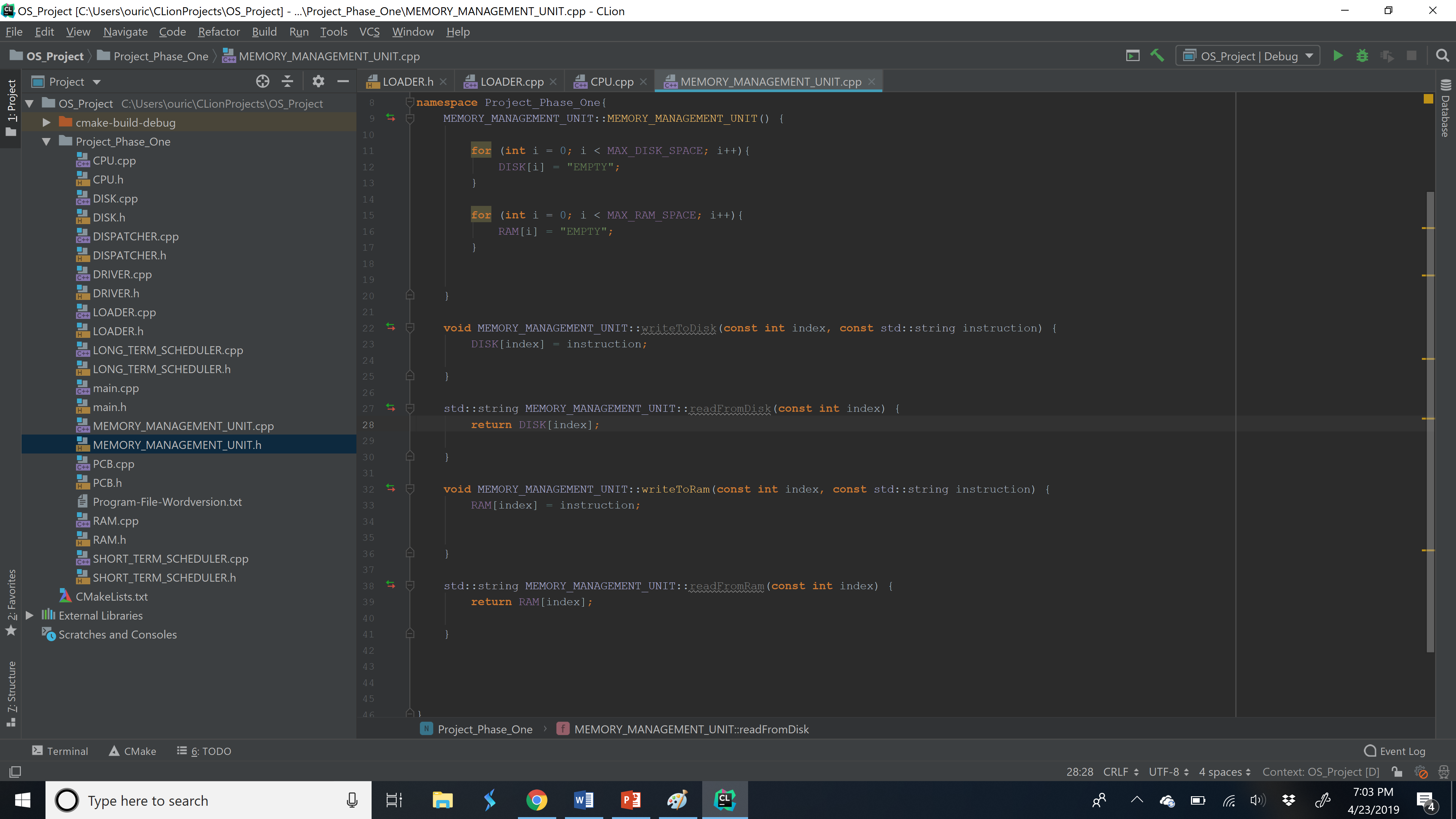
PCB()

## Memory System

### MMU Class

​This class facilitates the communication between other modules and the RAM. We designed our MMU to simulate an actual hardware rather than a separate virtual RAM and disk class. We structured our MMU to take in the loaded instruction on to the disk and simultaneously, add the PCB on to a new queue to be later accessed. Disk and Ram were both placed within the MMU and can ONLY be accessed by the MMU and no other class.

Within the MMU we have methods to read/write into the Disk and Ram.

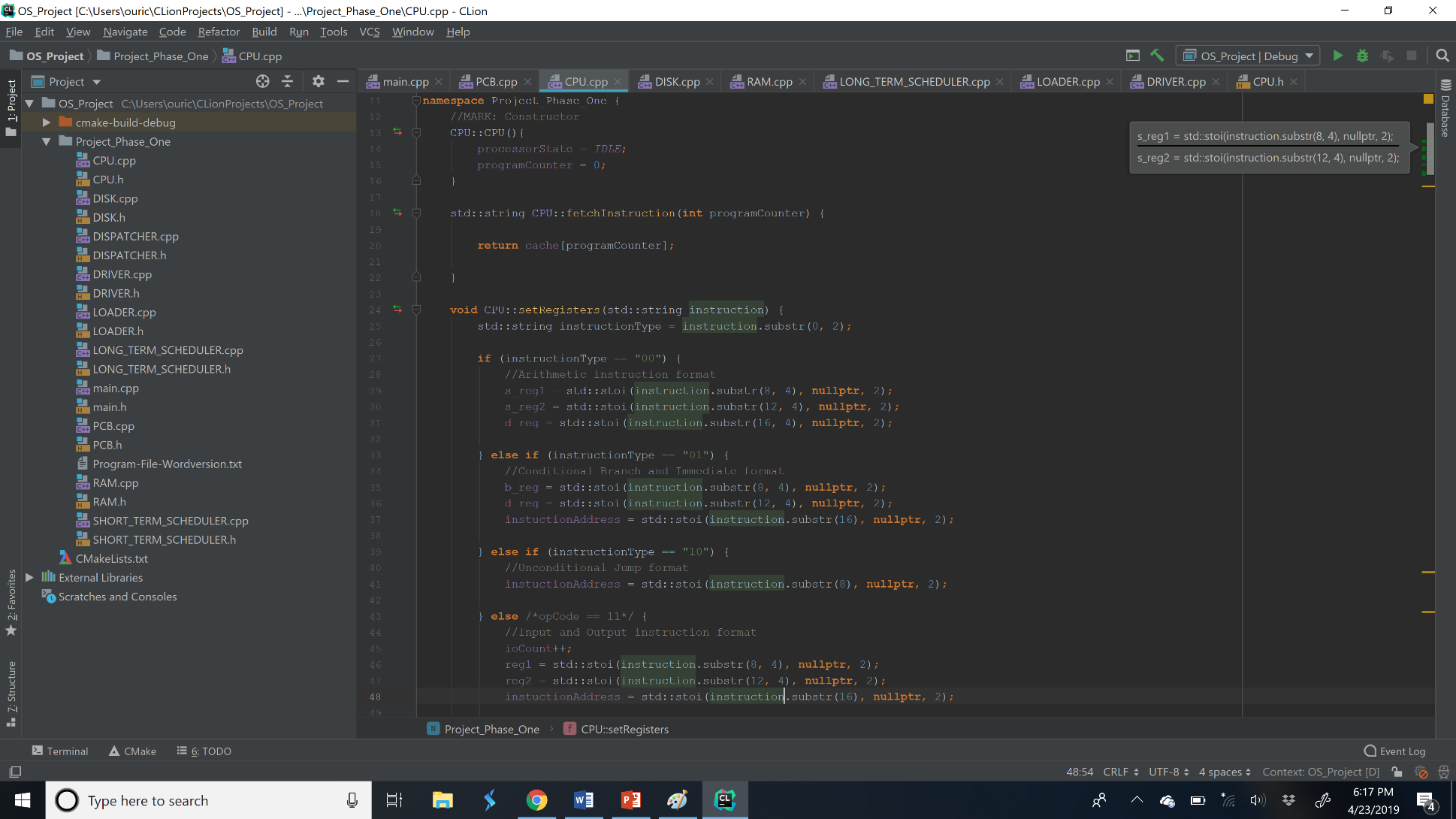


MMU Class (Read/Write function)

## Central Processing Unit

### CPU

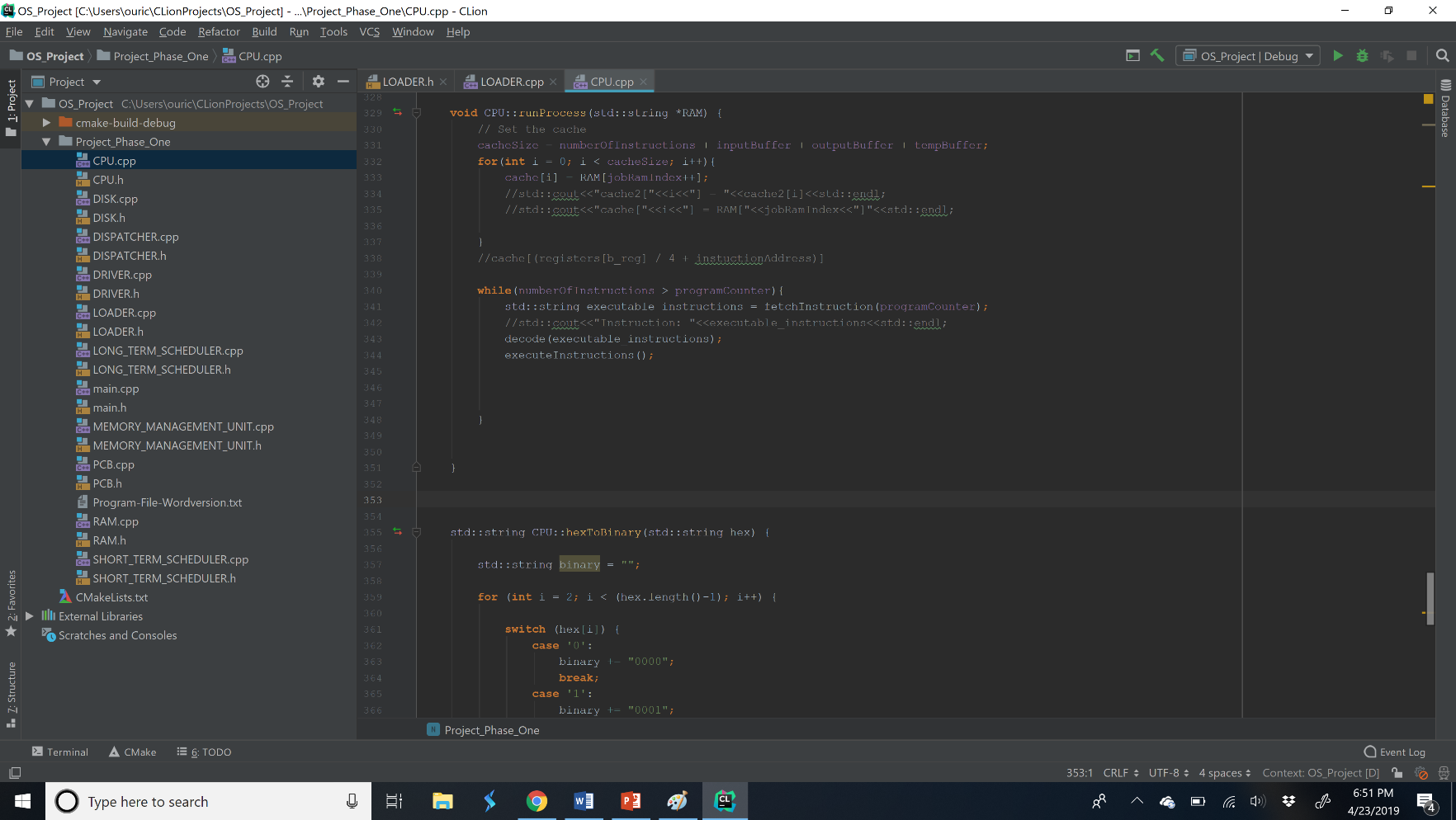
The CPU will first set a cache by adding the number of jobs, input  
buffer, output buffer, and temp buffer to get the cache size. The PCB  
saved the instruction location of the ram in the PCB index, and while  
the job counter is less than the cache size; the instructions will be  
loaded onto the cache. Next, the CPU will call the fetch method, this  
method will retrieve the instructions from the cache using the program  
counter. The program counter will be initialized to zero. Then, the  
CPU will send the hex string to be decoded by into binary string. The  
set register method will read the first two characters and get  
instruction type; this method will also set the registers depending on  
the instruction type. Once the register is set, the CPU will execute  
the instructions until the program counter equals the job counter or  
the process need I/O.



CPU Fetch()



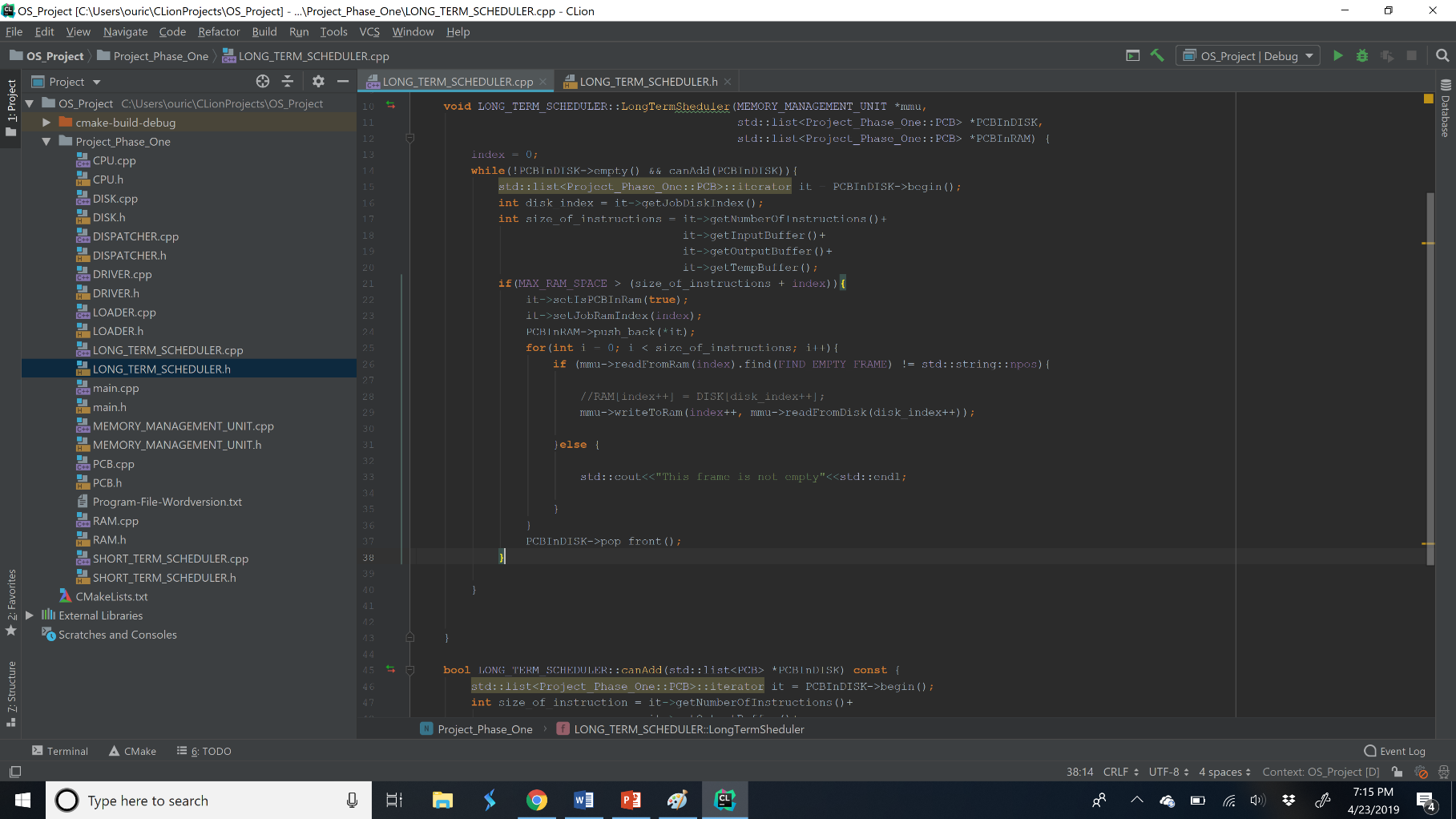
CPU (Decode/Execute)

CPU run process/conversion Hex-> Binary

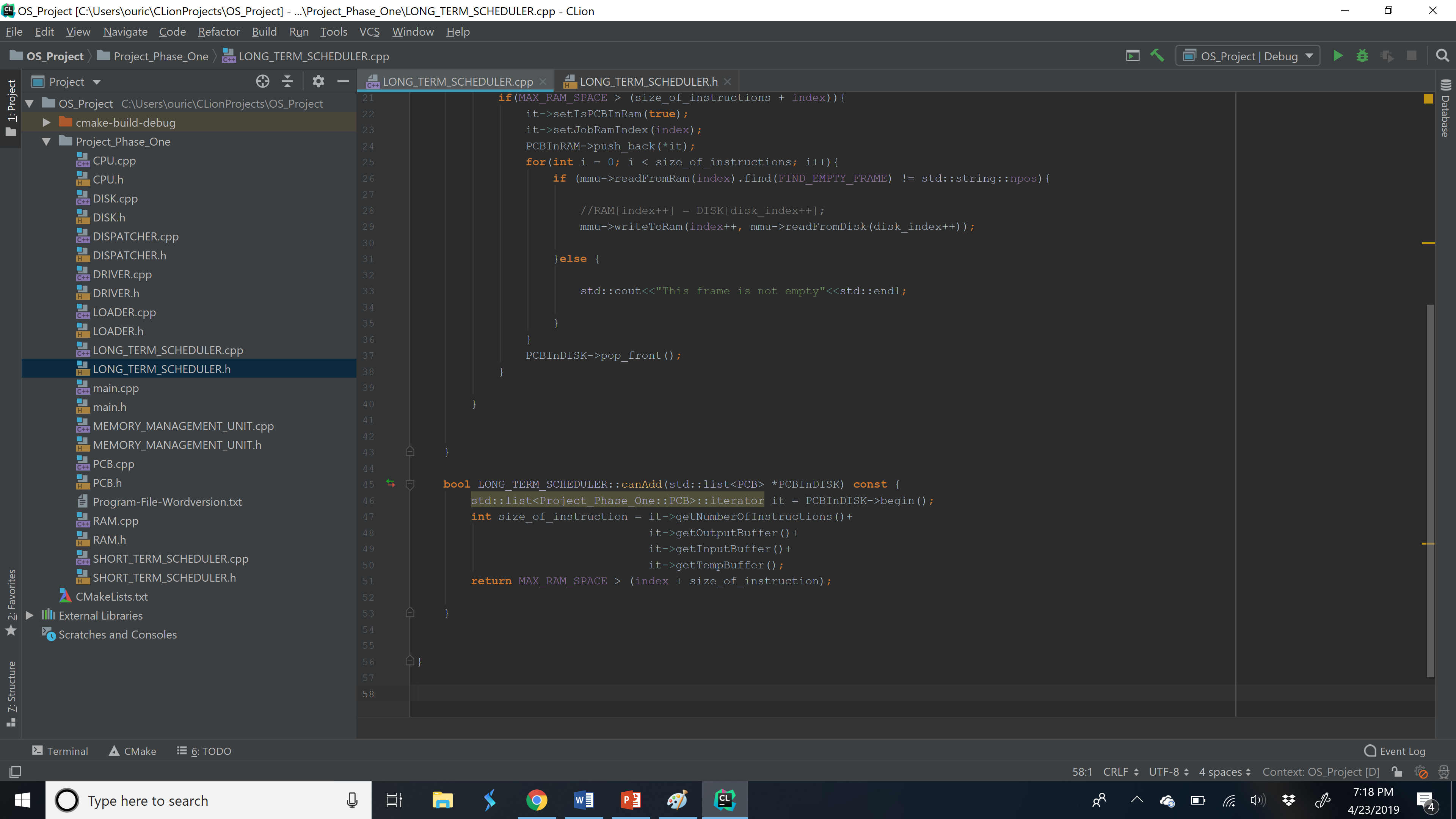
## Schedulers

LongTerm-Scheduler Class

The Long-Term Scheduler loads jobs into RAM. Our Long-term scheduler works by checking if our PCB in disk is empty or not. It must also check if the RAM has enough space by, adding the current process’ number of instructions, input/output/temp buffer to figure out our total instruction count. We’ll add that number into the current index in RAM. If that number is > that max ram size, then it will stop bringing the PCB into RAM and will continue to the short-term scheduler. If not empty and we can add, our Long-term scheduler will look for an empty frame to load our instructions into. Our Long-Term Scheduler will continue to loop until our PCB within the disk is empty.



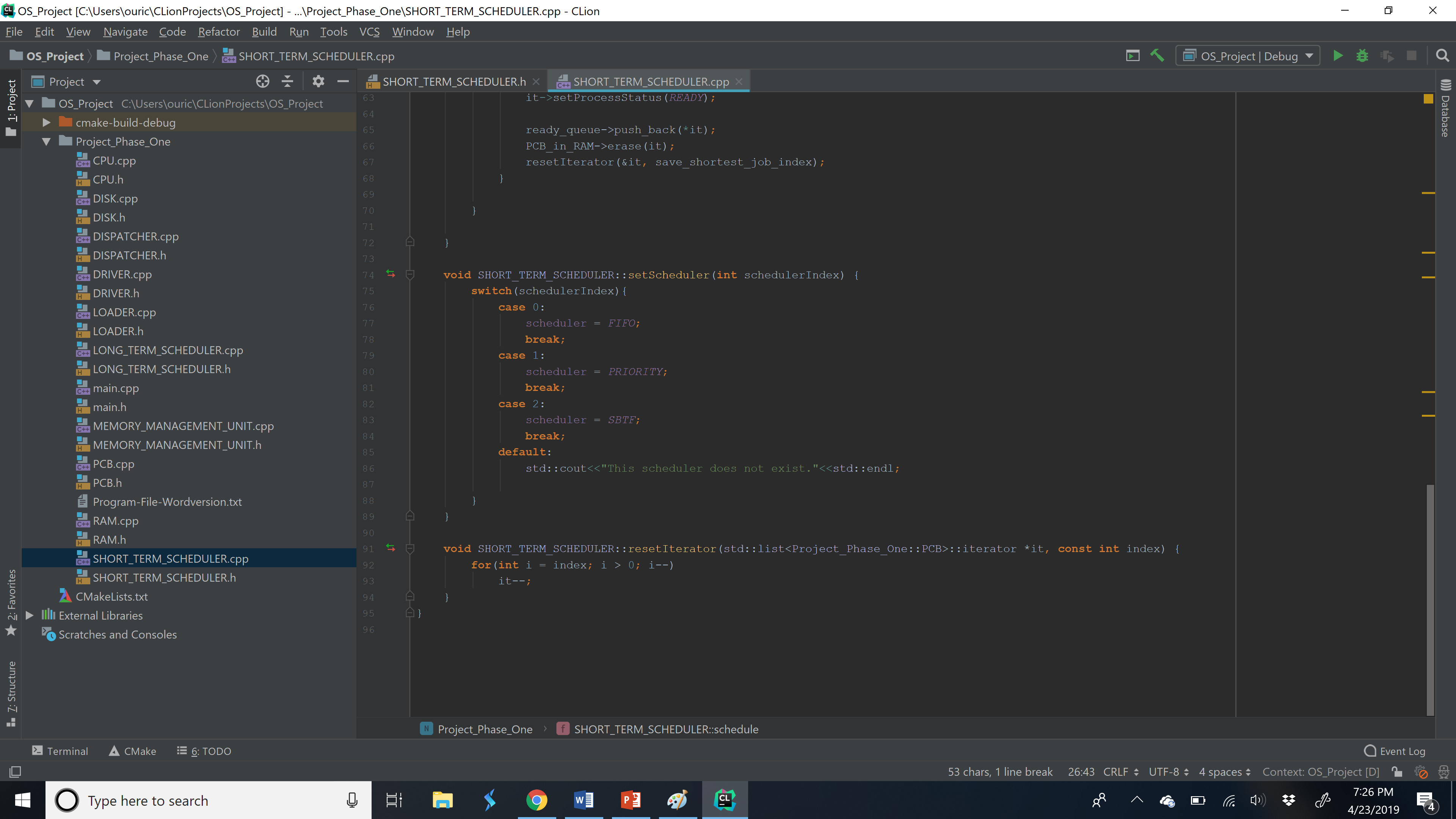
#### LongTermScheduler()

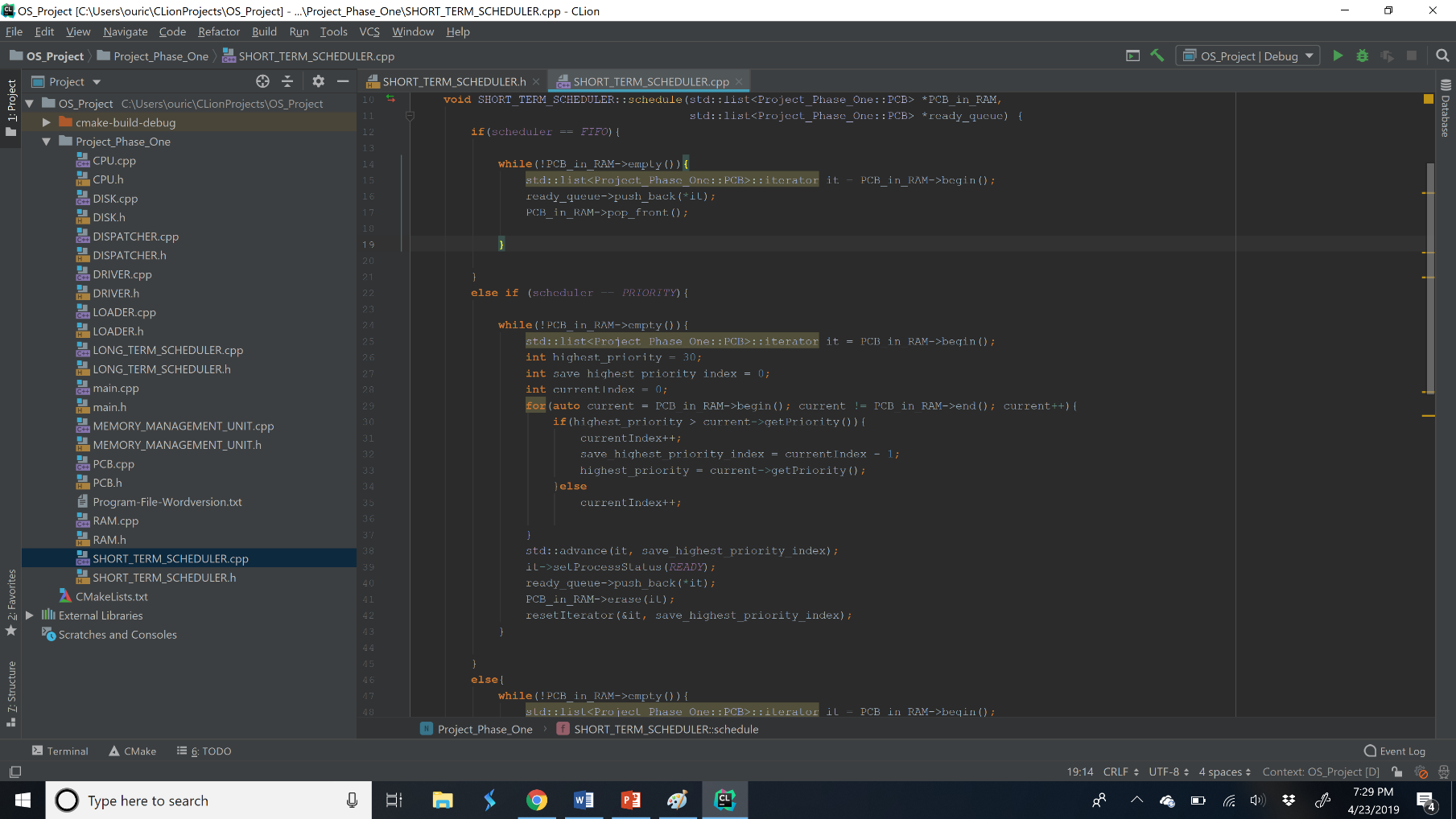


LongTermScheduler(canAdd)

### Short Term Scheduler Class

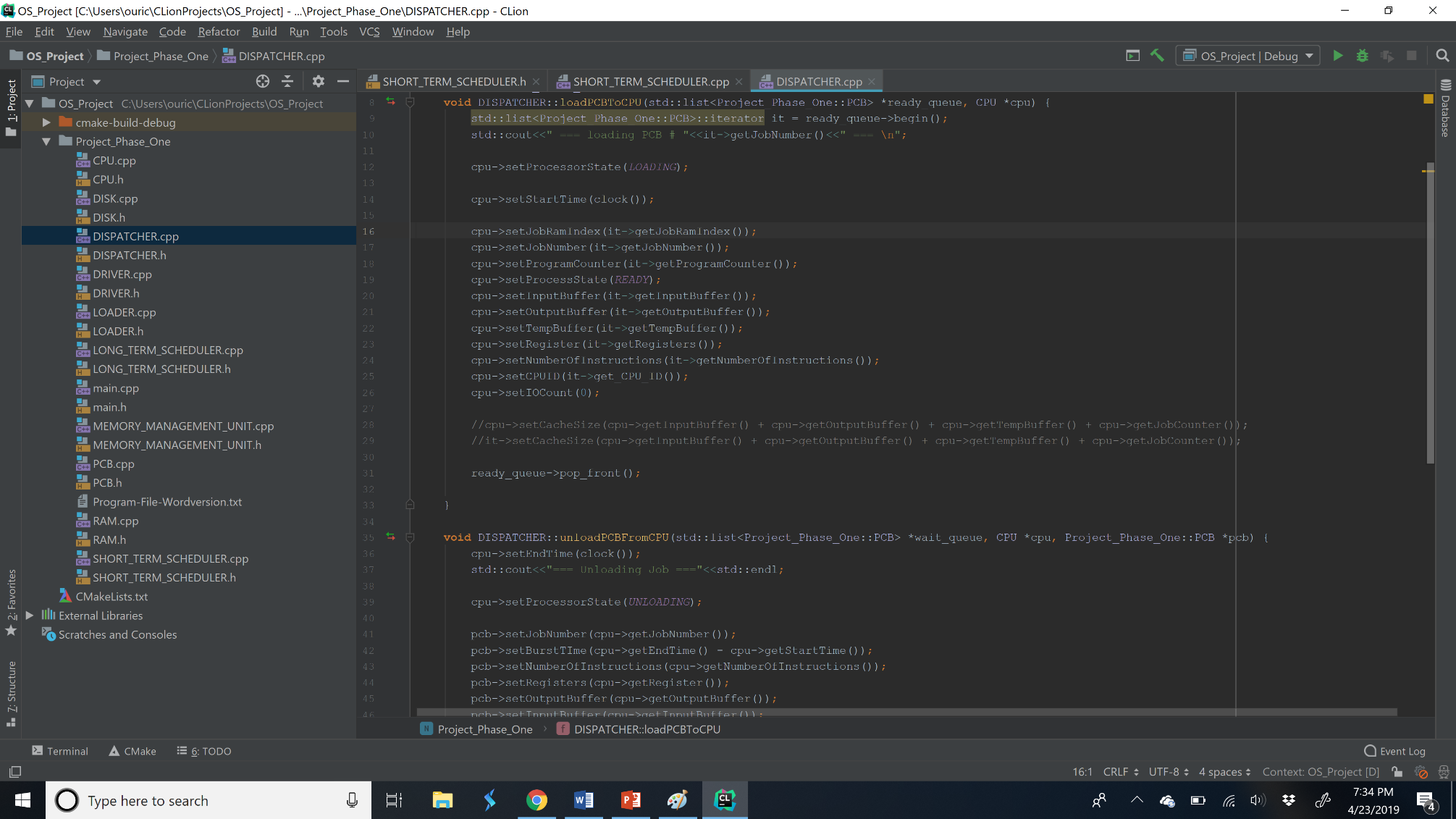
Our Short-Term Scheduler Class loads PCB to be put into the ready queue for execution based on the scheduling methods that we choose. The scheduling methods that we could select from were first in FIFO, priority, and SJF. One of these scheduling methods will take an integer and based on that integer our scheduling method will be ran. The method of scheduling that we choose would determine the order in which the job would be added on to the ready queue to be sent to our dispatcher.

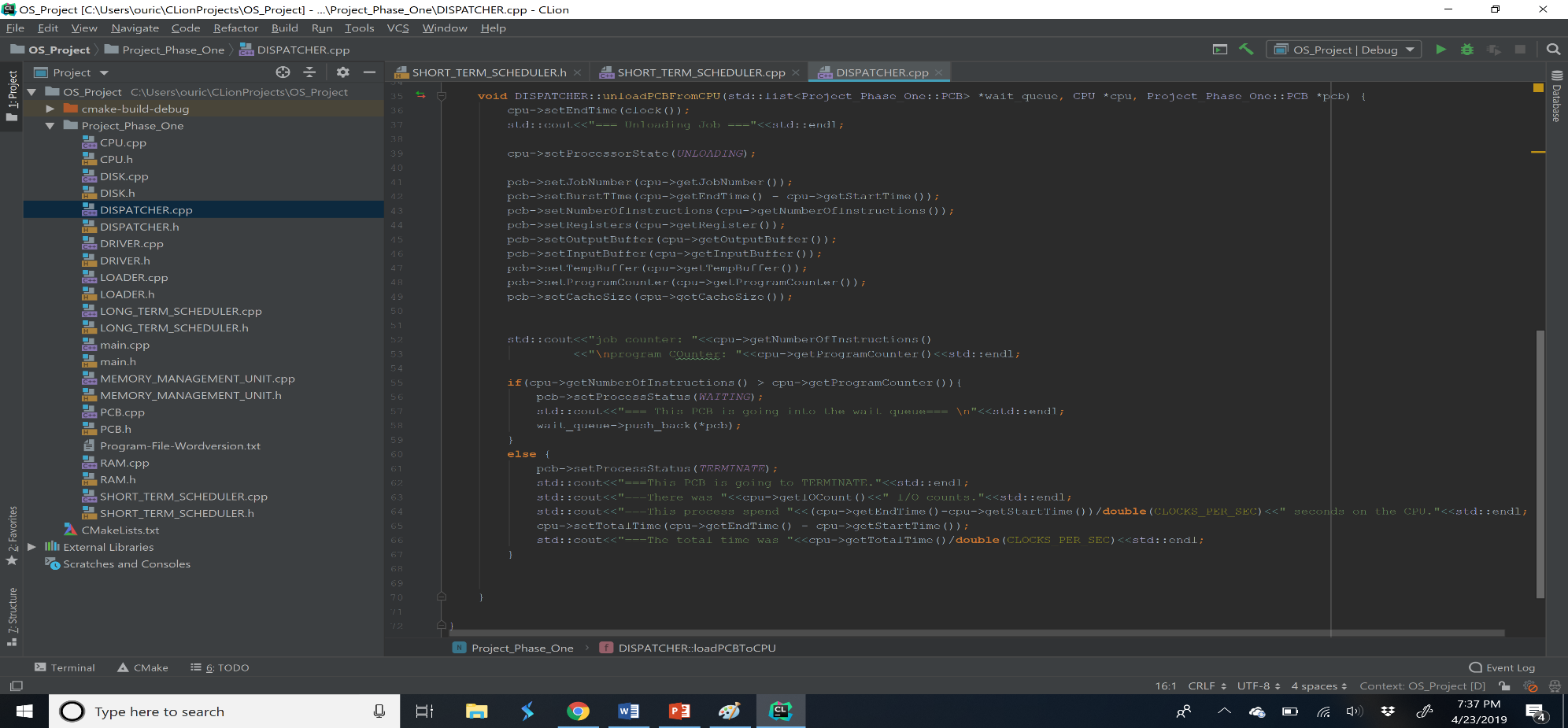


Short Term Scheduler(setScheduler)(resetIterator- used to return to first iter of the PCB list.)

Short Term Scheduler(schedule)

### Dispatcher Class

Dispatcher will load and unload our PCB onto the CPU. For our load method, it will take the information from the PCB and place it onto the CPU, this may include number of instructions, buffers, setting the Register, etc. For our unload method, it will unload the PCB from the CPU, if the program counter is < than the job counter then the PCB status will be placed onto a wait que for I/O. If the job counter equals the program counter then it flips the process status to terminate; it will then move it off the PCB to a completed list.

Dispatcher load()

Dispatcher unload()

# Conclusion

In conclusion, all the jobs spent roughly the same amount of time on  
the CPU, but the wait time varies depending on the scheduler. The FIFO  
scheduler had the longest wait time because the some of the shorter  
jobs had to wait for the longer jobs to finish. The Priority scheduler  
had a lower wait time than the FIFO scheduler, but not by much. This  
might be due to the fact that some job with high priority had a longer  
burst time than jobs with lower burst time. The SJF scheduler was the  
fastest, since the shortest jobs were able to execute before the  
longer jobs the wait time was cut down. The SFJ allow the program to  
wait less but the execution time was not that different from the  
Priority scheduler. Since all the jobs burst time should be the same,  
the total execution time is determined by the wait time and overhead.  
This means the SJF and Priority scheduler wait time is much lower than  
the FIFO and possibly has less overhead. When the CPU is set the  
number of cores from one to four, the total run time decrease by  
approximately 60%. This can be a result of the decrease in wait time.  
With four cores, the instructions will be executed 4 at a time. This  
results in a large decrease in wait time, and as we concluded before,  
the wait time determines the total execution time because each process  
should spends the same amount of time on the CPU no matter the  
scheduler.

# FIFO Statistics

FIFO stats:

Average Wait Time (Minimum 10 Runs): .002883

Average Run Time (Minimum 10 Runs): .00190

Average Total Time(Minimum 10 Runs): .01044

FIFO Table Statistics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| JOB# | WAIT TIME | RUN TIME | Pages Used | I/O Used | Cache Used |
| 1 | 0 | |  | | --- | |  | | .000243 | | 15.75 | 12 | 93.1% |
| 2 | .000243 | .000221 | 18 | 12 | 100% |
| 3 | .000464 | .000216 | 17 | 12 | 94.40% |
| 4 | .000680 | .000156 | 15.75 | 12 | 87.50% |
| 5 | .000836 | .000228 | 15.75 | 12 | 100% |
| 6 | .001064 | .000207 | 17 | 12 | 94.40% |
| 7 | .001271 | .000128 | 18 | 12 | 87.50% |
| 8 | .001399 | .000217 | 16.75 | 12 | 87.50% |
| 9 | .001523 | .000215 | 18 | 12 | 94.40% |
| 10 | .00174 | .000255 | 17 | 12 | 100% |
| 11 | .001955 | .000202 | 16.75 | 12 | 93% |
| 12 | .002157 | .000103 | 18 | 12 | 100% |
| 13 | .002412 | .000293 | 17 | 12 | 94% |
| 14 | .002705 | .000110 | 15.75 | 10 | 88% |
| 15 | .002815 | .000190 | 17 | 11 | 94% |
| 16 | .003005 | .000103 | 16.75 | 5 | 88% |
| 17 | .003108 | .000293 | 18 | 12 | 100% |
| 18 | .003401 | .000211 | 17 | 12 | 94% |
| 19 | .003601 | .000135 | 18.75 | 10 | 100% |
| 20 | .003817 | .000211 | 17 | 12 | 94% |
| 21 | .004028 | .000135 | 15.75 | 13 | 88% |
| 22 | .004163 | .000221 | 17 | 12 | 100% |
| 23 | .004384 | .000188 | 16.75 | 11 | 93% |
| 24 | .004572 | .000224 | 17 | 12 | 94% |
| 25 | .004796 | .000127 | 16.75 | 7 | 93% |
| 26 | .004923 | .00083 | 15.75 | 8 | 88% |
| 27 | .005006 | .00024 | 16.75 | 12 | 93% |
| 28 | .005246 | .000237 | 18 | 12 | 100% |
| 29 | .005483 | .000205 | 17 | 12 | 94% |
| 30 | .005688 | .000111 | 15.75 | 30 | 88% |

# Priority Statistics

Priority stats:

Average Wait Time (Minimum 10 Runs): .006225

Average Run Time (Minimum 10 Runs): .000162

Average Total Time(Minimum 10 Runs): .010503

# Priority Table Statistics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| JOB# | WAIT TIME | RUN TIME | Pages Used | I/O Used | Cache Used |
| 1 | .000446 | .000172 | 15.75 | 12 | 93.1% |
| 2 | .001502 | .000195 | 18 | 12 | 100% |
| 3 | .002423 | .000174 | 17 | 12 | 94.40% |
| 4 | .001828 | .000115 | 15.75 | 12 | 87.50% |
| 5 | .001072 | .000182 | 15.75 | 12 | 100% |
| 6 | .002644 | .000176 | 17 | 12 | 94.40% |
| 7 | .002027 | .000112 | 18 | 12 | 87.50% |
| 8 | .003096 | .000107 | 16.75 | 12 | 87.50% |
| 9 | .000644 | .000174 | 18 | 12 | 94.40% |
| 10 | 0 | .000202 | 17 | 12 | 100% |
| 11 | .001292 | .000194 | 16.75 | 12 | 93% |
| 12 | .002875 | .000188 | 18 | 12 | 100% |
| 13 | .000222 | .000172 | 17 | 12 | 94% |
| 14 | .000934 | .000089 | 15.75 | 10 | 88% |
| 15 | .00221 | .000164 | 17 | 11 | 94% |
| 16 | .005443 | .000073 | 16.75 | 5 | 88% |
| 17 | .004435 | .000187 | 18 | 12 | 100% |
| 18 | .003224 | .000181 | 17 | 12 | 94% |
| 19 | .003637 | .000188 | 18.75 | 10 | 100% |
| 20 | .004026 | .000184 | 17 | 12 | 94% |
| 21 | .004716 | .000116 | 15.75 | 13 | 88% |
| 22 | .003414 | .000183 | 17 | 12 | 100% |
| 23 | .005018 | .000233 | 16.75 | 11 | 93% |
| 24 | .005232 | .000223 | 17 | 12 | 94% |
| 25 | .005642 | .0001 | 16.75 | 7 | 93% |
| 26 | .004902 | .000085 | 15.75 | 8 | 88% |
| 27 | .005787 | .000173 | 16.75 | 12 | 93% |
| 28 | .004233 | .000209 | 18 | 12 | 100% |
| 29 | .003086 | .000208 | 17 | 12 | 94% |
| 30 | .005511 | .000099 | 15.75 | 30 | 88% |

# SJF Statistics

SJF stats:

Average Wait Time (Minimum 10 Runs): .002616

Average Run Time (Minimum 10 Runs): .000186

Average Total Time(Minimum 10 Runs): .010919

# SJF Table Statistics

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| JOB# | WAIT TIME | RUN TIME | Pages Used | I/O Used | Cache Used |
| 1 | .000536 | .000206 | 15.75 | 12 | 93.1% |
| 2 | .002002 | .000219 | 18 | 12 | 100% |
| 3 | .00095 | .000204 | 17 | 12 | 94.40% |
| 4 | 0 | .000155 | 15.75 | 12 | 87.50% |
| 5 | .002221 | .000201 | 15.75 | 12 | 100% |
| 6 | .001154 | .000272 | 17 | 12 | 94.40% |
| 7 | .000115 | .000139 | 18 | 12 | 87.50% |
| 8 | .000294 | .000116 | 16.75 | 12 | 87.50% |
| 9 | .001426 | .000182 | 18 | 12 | 94.40% |
| 10 | .002422 | .000219 | 17 | 12 | 100% |
| 11 | .000742 | .000208 | 16.75 | 12 | 93% |
| 12 | .002641 | .000213 | 18 | 12 | 100% |
| 13 | .001608 | .000203 | 17 | 12 | 94% |
| 14 | .00041 | .000126 | 15.75 | 10 | 88% |
| 15 | .001811 | .000191 | 17 | 11 | 94% |
| 16 | .002854 | .000128 | 16.75 | 5 | 88% |
| 17 | .004765 | .000216 | 18 | 12 | 100% |
| 18 | .003906 | .000211 | 17 | 12 | 94% |
| 19 | .004991 | .0002 | 18.75 | 10 | 100% |
| 20 | .004117 | .00021 | 17 | 12 | 94% |
| 21 | .002982 | .000164 | 15.75 | 13 | 88% |
| 22 | .005191 | .000225 | 17 | 12 | 100% |
| 23 | .003379 | .000189 | 16.75 | 11 | 93% |
| 24 | .004327 | .000223 | 17 | 12 | 94% |
| 25 | .003568 | .000138 | 16.75 | 7 | 93% |
| 26 | .003146 | .000112 | 15.75 | 8 | 88% |
| 27 | .003706 | .0002 | 16.75 | 12 | 93% |
| 28 | .005416 | .00024 | 18 | 12 | 100% |
| 29 | .00455 | .000215 | 17 | 12 | 94% |
| 30 | .003258 | .000121 | 15.75 | 30 | 88% |

# I/O’s per Job

% of Cache used

# Pages Used Per Job