Operating Systems Project Phase 2 Final Report

Nawal Ahmed & Jordan Hasty

CVM++ OS

Abstract & Introduction

This final report of the semester project details the current state of the virtual machine simulation up to Phase 2. The report covers the details of the design approach and the functions and implementations of the modules that constitute the simulation program. Also provided is the data collected during the simulation runs as well as an analysis of the data.

Design Approach

The simulation program is written in C++ as it allows greater control of memory, native conversion between binary, hex, and decimal, and allows the use of pointers. Combined, these elements allow the simulation to perform better than if it were written in another language such as Java.

The group began constructing the design of the virtual machine by first examining the project specification and block diagram that were provided. We decided to use a modularized approach in our design to allow code reuse and to maintain an organized and coherent structure. In deriving the structure of the program, we decided that the main component, the driver, should be contained separately and call upon various functions to run the simulation. These functions that the driver calls are contained in separate modules which correspond to the individual components that constitute the virtual machine such as the CPU, RAM, Disk, etc. (more detail on the specific modules later).

We began with the driver which called upon unimplemented functions as we decided what functions the various modules should perform to give basic structure to the program. This would also ensure that separate parts of the program would fit together as each person worked on their own module. Having figured out the functions of certain modules, we then began our implementation of functions within them. During the program’s development, we individually tested each module upon completion to ensure that they functioned properly.

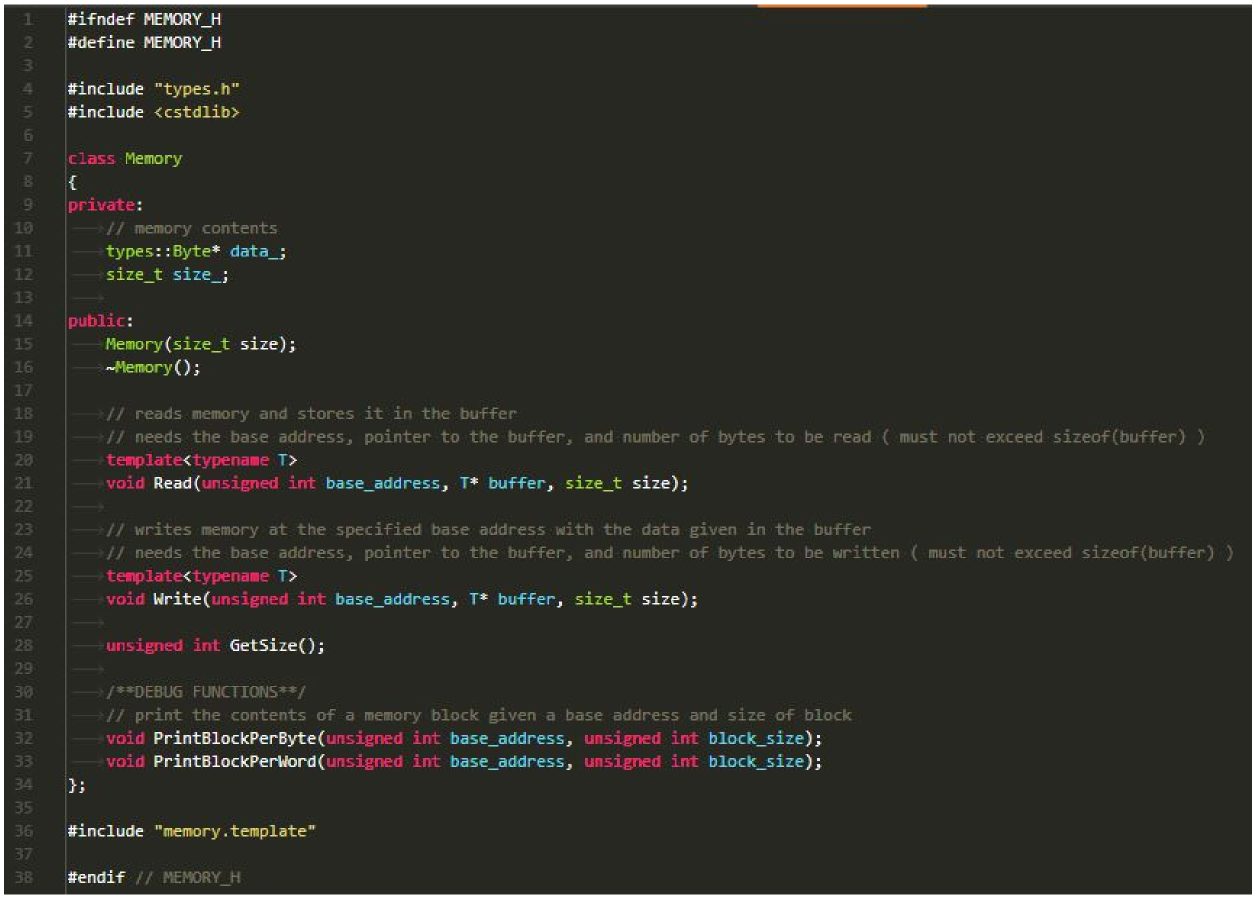
Design Diagram:

Implementation Modules

The following modules represent the individual components that comprise the virtual machine and operating system. Each module has its own functions which all work together to perform the simulation. The following sections describe each module's implementation and functionality.

Memory System

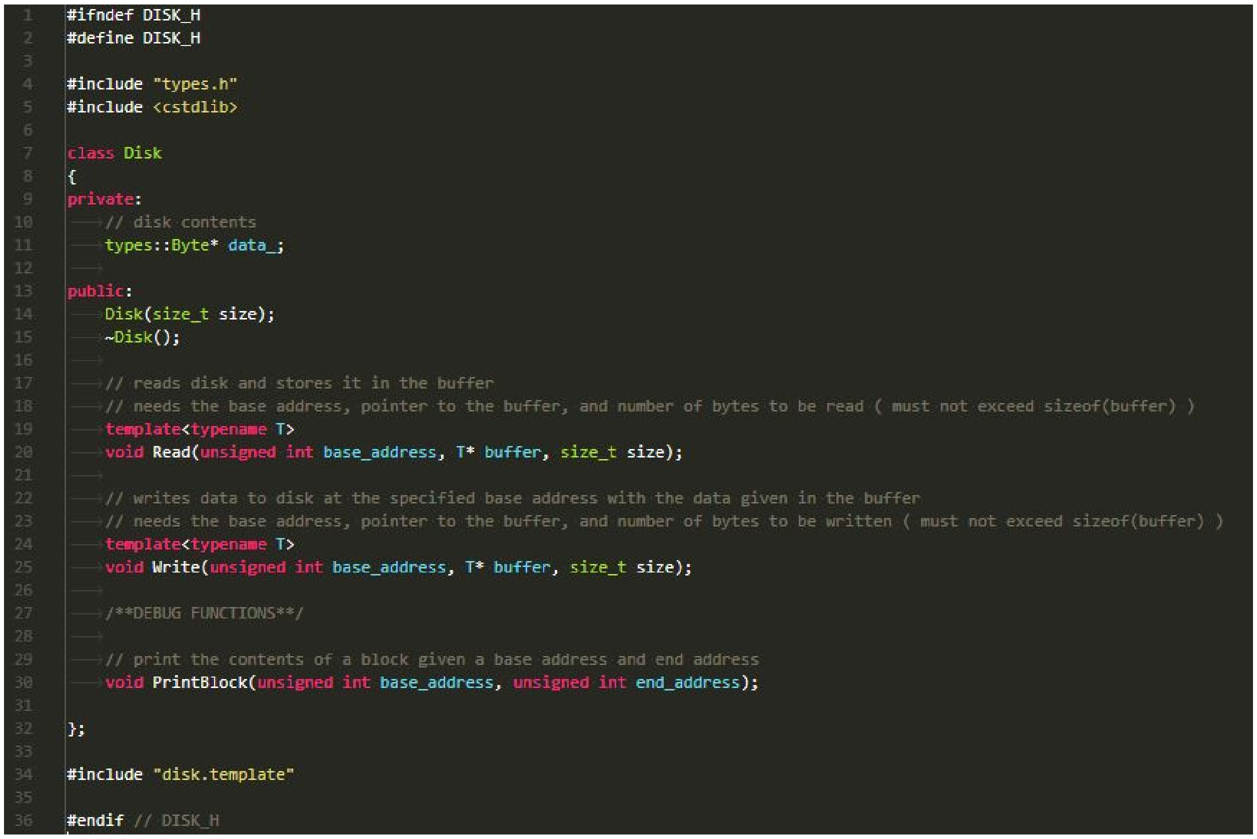
RAM:



```
#ifndef MEMORY_H
#define MEMORY_H

#include "types.h"
#include <cstdlib>

class Memory
{
private:
	// memory contents
	types::Byte* data_;
	size_t size_;
	
public:
	Memory(size_t size);
	~Memory();

	// reads memory and stores it in the buffer
	// needs the base address, pointer to the buffer, and number of bytes to be read ( must not exceed sizeof(buffer) )
	template<typename T>
	void Read(unsigned int base_address, T* buffer, size_t size);
	
	// writes memory at the specified base address with the data given in the buffer
	// needs the base address, pointer to the buffer, and number of bytes to be written ( must not exceed sizeof(buffer) )
	template<typename T>
	void Write(unsigned int base_address, T* buffer, size_t size);
	
	unsigned int GetSize();
	
	/**DEBUG FUNCTIONS**/
	// print the contents of a memory block given a base address and size of block
	void PrintBlockPerByte(unsigned int base_address, unsigned int block_size);
	void PrintBlockPerWord(unsigned int base_address, unsigned int block_size);
};

#include "memory.template"

#endif // MEMORY_H
```

This is the memory module which processes are loaded into. It contains an array of unsigned 8-bit integers which stores the content of the RAM. This allows the RAM module to be byte-addressable. It contains functions through which data is read and written to the data array. By using templated functions that utilize a buffer, any data type and be read/written to/from the data array and bytes are automatically concatenated when returning the specific data type.

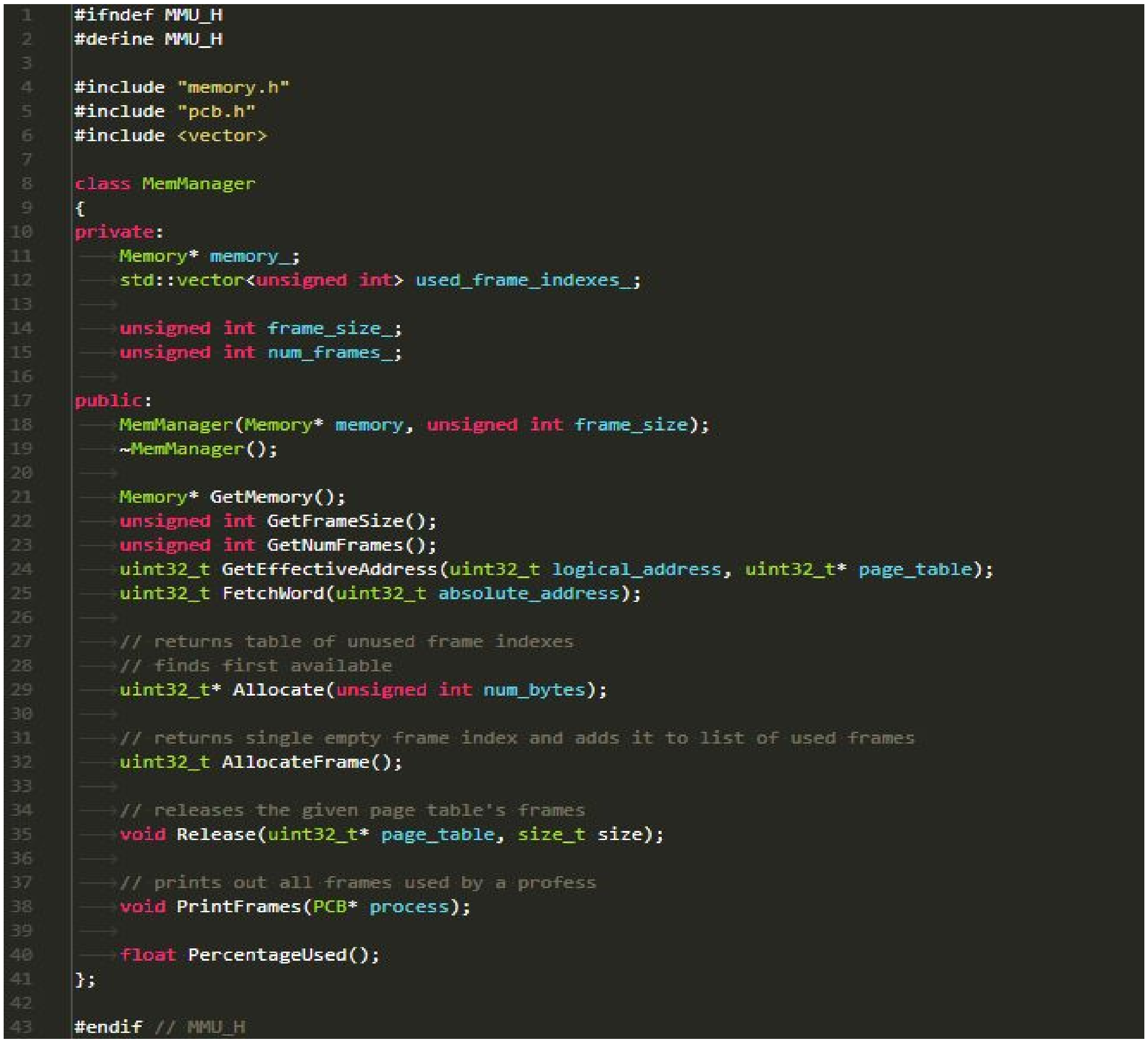
Disk:



```cpp
#ifndef DISK_H
#define DISK_H

#include "types.h"
#include <cstdlib>

class Disk
{
private:
	// disk contents
	types::Byte* data_;
	
public:
	Disk(size_t size);
	~Disk();
	
	// reads disk and stores it in the buffer
	// needs the base address, pointer to the buffer, and number of bytes to be read ( must not exceed sizeof(buffer) )
	template<typename T>
	void Read(unsigned int base_address, T* buffer, size_t size);
	
	// writes data to disk at the specified base address with the data given in the buffer
	// needs the base address, pointer to the buffer, and number of bytes to be written ( must not exceed sizeof(buffer) )
	template<typename T>
	void Write(unsigned int base_address, T* buffer, size_t size);
	
	/**DEBUG FUNCTIONS**/
	
	// print the contents of a block given a base address and end address
	void PrintBlock(unsigned int base_address, unsigned int end_address);

};

#include "disk.template"

#endif // DISK_H
```

This is the memory module which programs are loaded into from the data file. Like the RAM module, it also contains an array of unsigned 8-bit integers which stores the content of the disk and allows the disk to be byte-addressable. It also contains identical read/write functions identical to those used in the RAM module. While it may seem redundant to use a separate implementation for the Disk and RAM modules, it did allow for a more coherent design structure.

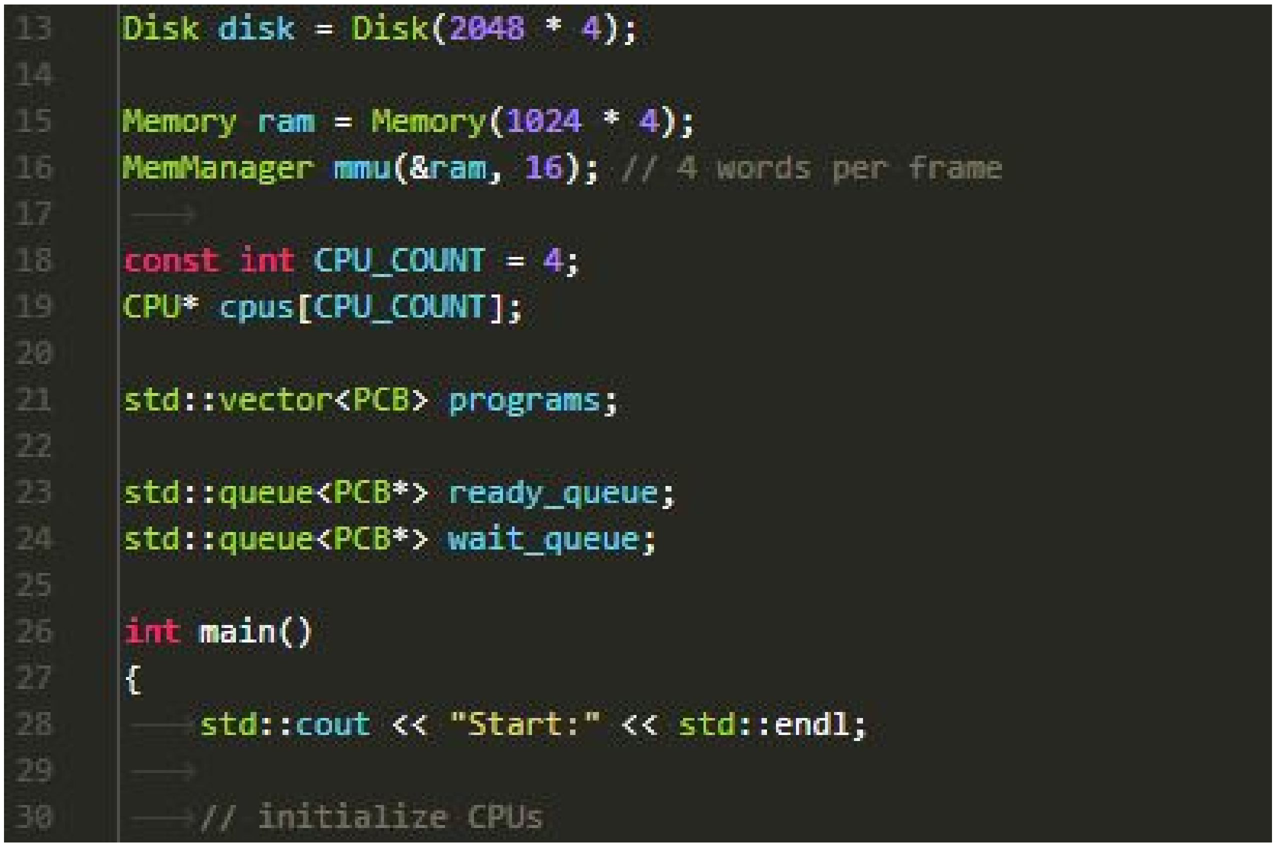
MMU:



```
#ifndef MMU_H
#define MMU_H

#include "memory.h"
#include "pcb.h"
#include <vector>

class MemManager
{
private:
	Memory* memory_;
	std::vector<unsigned int> used_frame_indexes_;
	
	unsigned int frame_size_;
	unsigned int num_frames_;
	
public:
	MemManager(Memory* memory, unsigned int frame_size);
	~MemManager();
	
	Memory* GetMemory();
	unsigned int GetFrameSize();
	unsigned int GetNumFrames();
	uint32_t GetEffectiveAddress(uint32_t logical_address, uint32_t* page_table);
	uint32_t FetchWord(uint32_t absolute_address);
	
	// returns table of unused frame indexes
	// finds first available
	uint32_t* Allocate(unsigned int num_bytes);
	
	// returns single empty frame index and adds it to list of used frames
	uint32_t AllocateFrame();
	
	// releases the given page table's frames
	void Release(uint32_t* page_table, size_t size);
	
	// prints out all frames used by a profess
	void PrintFrames(PCB* process);
	
	float PercentageUsed();
};

#endif // MMU_H
```

The Memory Management Unit module (MMU) acts as an interface between the RAM module and the rest of the virtual machine. It maintains a list of used frame indexes which is necessary for the paging system. It also handles the allocation and release of memory. It contains functions for translating logical addresses to physical as these functions work closely with the paging system.

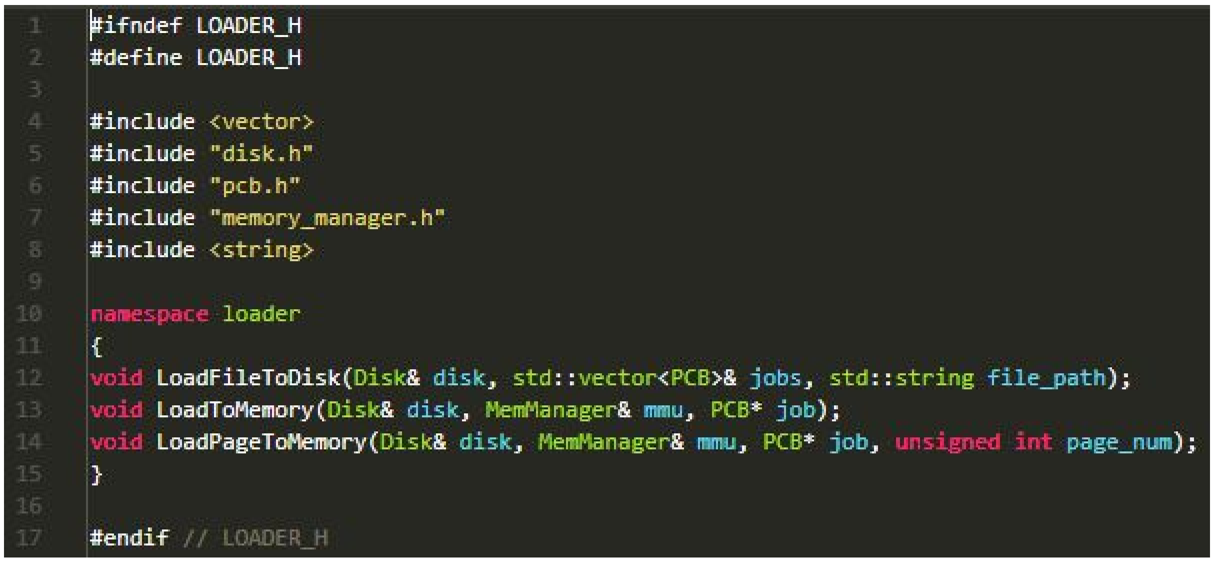
Driver



```cpp
Disk disk = Disk(2048 * 4);

Memory ram = Memory(1024 * 4);
MemManager mmu(&ram, 16); // 4 words per frame


const int CPU_COUNT = 4;
CPU* cpus[CPU_COUNT];

std::vector<PCB> programs;

std::queue<PCB*> ready_queue;
std::queue<PCB*> wait_queue;

int main()
{
    std::cout << "Start:" << std::endl;

    // initialize CPUs
```

The Driver module contains the entry point of the program. It handles the initialization of the various components and maintains the program queues. It also contains the main loop of the program which cycles until all programs have executed.

Loader

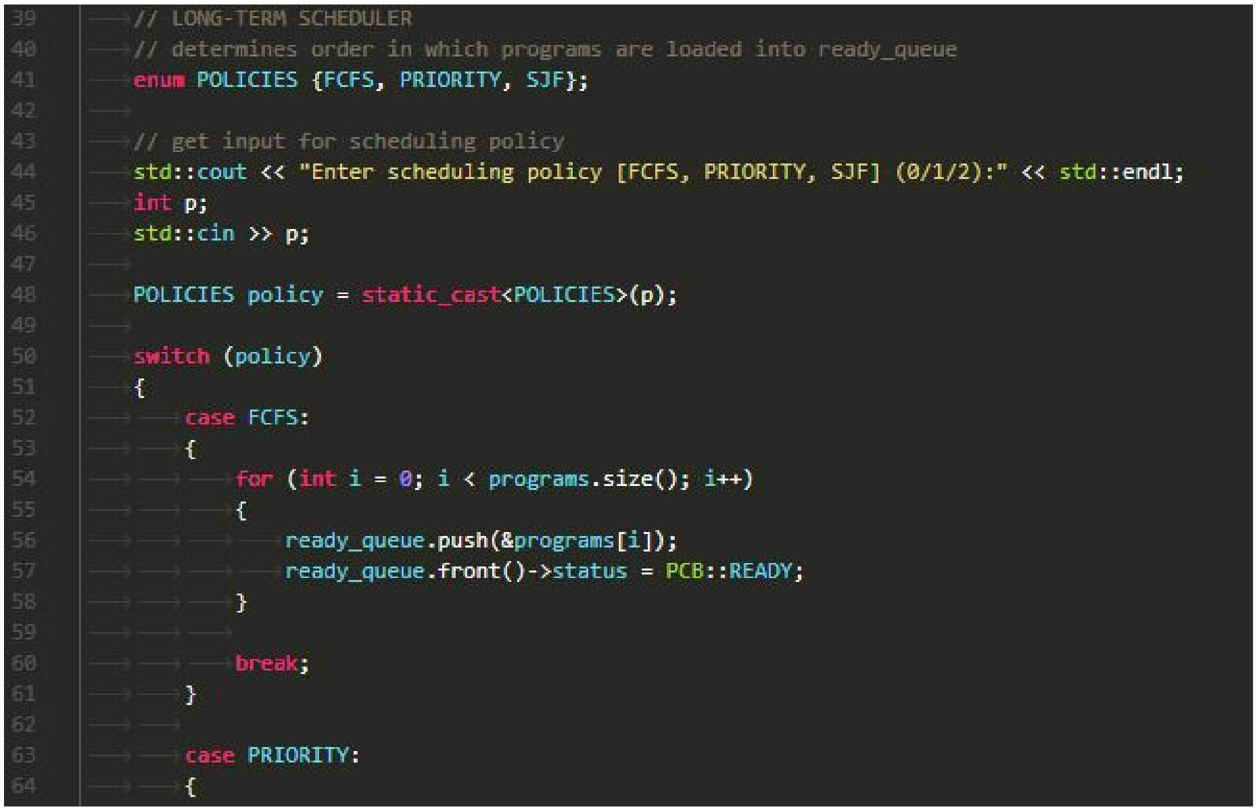


```cpp
#ifndef LOADER_H
#define LOADER_H

#include <vector>
#include "disk.h"
#include "pcb.h"
#include "memory_manager.h"
#include <string>

namespace loader
{
void LoadFileToDisk(Disk& disk, std::vector<PCB>& jobs, std::string file_path);
void LoadToMemory(Disk& disk, MemManager& mmu, PCB* job);
void LoadPageToMemory(Disk& disk, MemManager& mmu, PCB* job, unsigned int page_num);
}

#endif // LOADER_H
```

The Loader module handles loading data from the data file into the Disk module and parses program data into each programs PCB. The module is also responsible for loading process data into RAM.

Schedulers

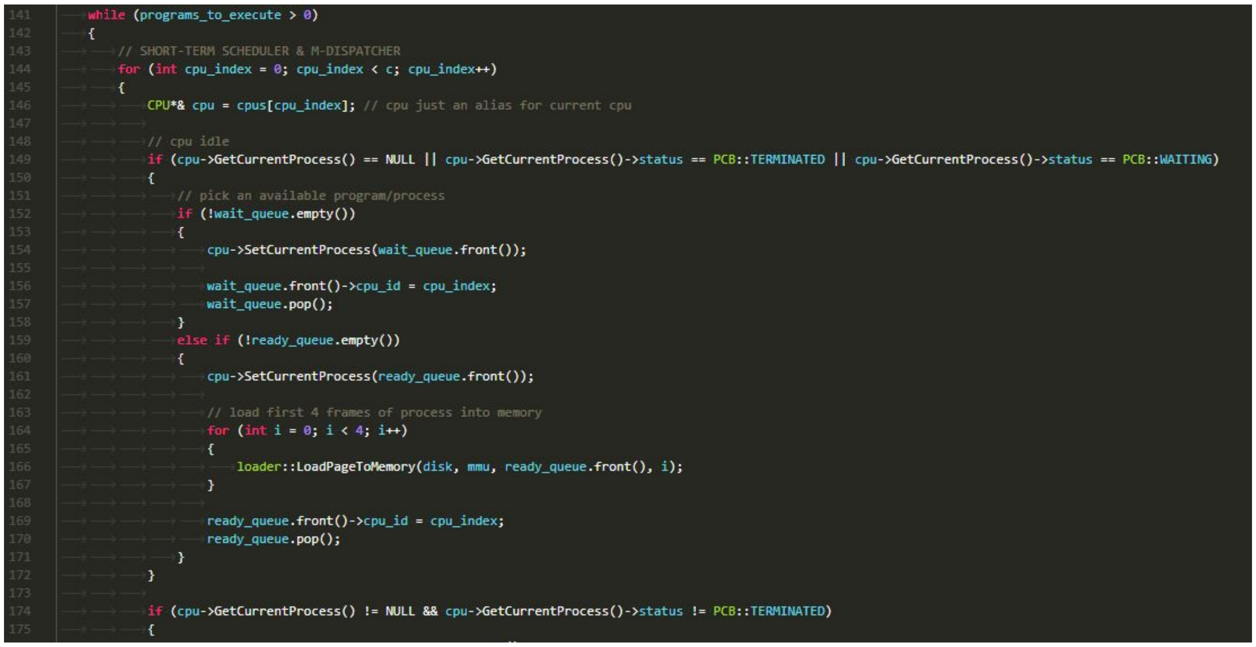
Long-Term Scheduler:



```
	// LONG-TERM SCHEDULER
	// determines order in which programs are loaded into ready_queue
	enum POLICIES {FCFS, PRIORITY, SJF};
	
	// get input for scheduling policy
	std::cout << "Enter scheduling policy [FCFS, PRIORITY, SJF] (0/1/2):" << std::endl;
	int p;
	std::cin >> p;
	
	POLICIES policy = static_cast<POLICIES>(p);
	
	switch (policy)
	{
		case FCFS:
		{
			for (int i = 0; i < programs.size(); i++)
			{
				ready_queue.push(&programs[i]);
				ready_queue.front()->status = PCB::READY;
			}
			
			break;
		}
		
		case PRIORITY:
		{
```

The Long-Term Scheduler module is contained within the Driver module or “main.” It is responsible for loading programs into the ready queue ordered by the given scheduling policy.

Short-Term Scheduler & Dispatcher:



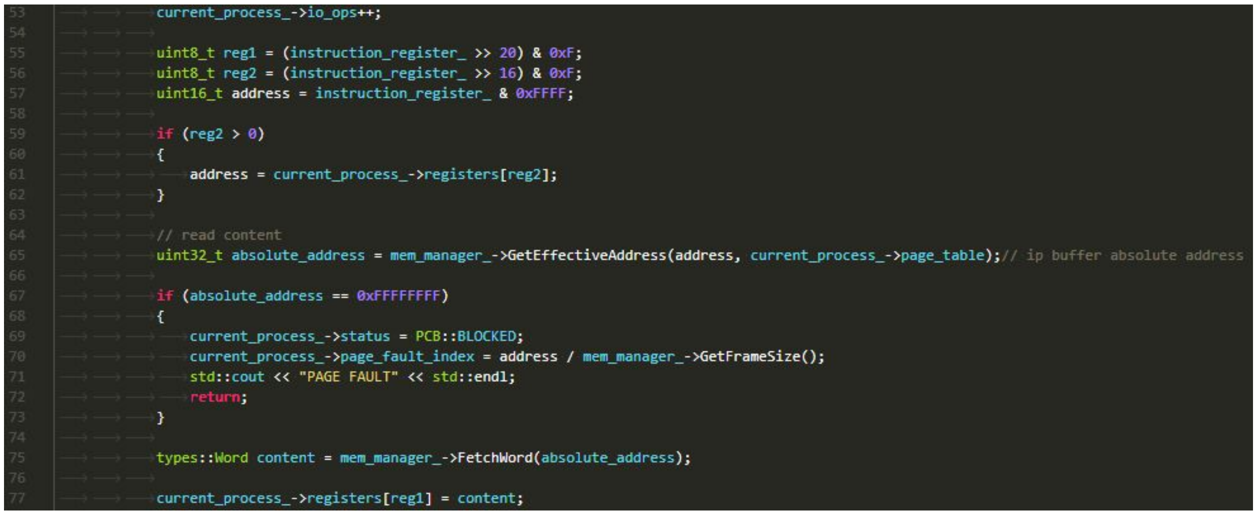
```
    while (programs_to_execute > 0)
    {
        // SHORT-TERM SCHEDULER & M-DISPATCHER
        for (int cpu_index = 0; cpu_index < c; cpu_index++)
        {
            CPU*& cpu = cpus[cpu_index]; // cpu just an alias for current cpu

            // cpu idle
            if (cpu->GetCurrentProcess() == NULL || cpu->GetCurrentProcess()->status == PCB::TERMINATED || cpu->GetCurrentProcess()->status == PCB::WAITING)
            {
                // pick an available program/process
                if (!wait_queue.empty())
                {
                    cpu->SetCurrentProcess(wait_queue.front());

                    wait_queue.front()->cpu_id = cpu_index;
                    wait_queue.pop();
                }
                else if (!ready_queue.empty())
                {
                    cpu->SetCurrentProcess(ready_queue.front());

                    // load first 4 frames of process into memory
                    for (int i = 0; i < 4; i++)
                    {
                        loader::LoadPageToMemory(disk, mmu, ready_queue.front(), i);
                    }

                    ready_queue.front()->cpu_id = cpu_index;
                    ready_queue.pop();
                }
            }

            if (cpu->GetCurrentProcess() != NULL && cpu->GetCurrentProcess()->status != PCB::TERMINATED)
            {
```

The Short-Term Scheduler and Dispatcher are contained within the Driver or “main.” The Short-Term Scheduler picks from either the ready queue or waiting queue. The Dispatcher then chooses an available CPU to assign the process. The Dispatcher is incorporated in this block of code as it works closely with the Short-Term Scheduler.

CPU

The CPU module is central to the decoding and execution of programs. Each CPU object is able to be assigned a process and then executes the next instruction pointed to by the PCB’s program counter. The CPU object itself does not contain the set of 16 32-bit registers as those are implemented in the PCB which the CPU directly interfaces with. The CPU decodes instructions by bit-shifting and masking to extract sections of the instruction word. The Execute method uses a switch statement to perform the correct set of operations depending on the instruction. Multiple CPUs are able to be instantiated and used concurrently in the system.

DMA-Channel:



```
            current_process_->io_ops++;

            uint8_t reg1 = (instruction_register_ >> 20) & 0xF;
            uint8_t reg2 = (instruction_register_ >> 16) & 0xF;
            uint16_t address = instruction_register_ & 0xFFFF;

            if (reg2 > 0)
            {
                address = current_process_->registers[reg2];
            }

            // read content
            uint32_t absolute_address = mem_manager_->GetEffectiveAddress(address, current_process_->page_table);// ip buffer absolute address

            if (absolute_address == 0xFFFFFFFF)
            {
                current_process_->status = PCB::BLOCKED;
                current_process_->page_fault_index = address / mem_manager_->GetFrameSize();
                std::cout << "PAGE FAULT" << std::endl;
                return;
            }

            types::Word content = mem_manager_->FetchWord(absolute_address);

            current_process_->registers[reg1] = content;
```

The DMA-Channel is incorporated directly into the CPU module. This section of code is responsible for handling I/O instructions. It calls upon the MMU to separately handle reading and writing from memory to free up CPU time.

PCB

The Process Control Block (PCB) struct contains all data that pertains to a specific program. Within the system, there only exists a single PCB per program which exists in the Driver’s programs vector. All other parts of the system that interact with a program’s PCB interface through pointers so that there only exists one PCB in memory per program. For instance, the ready queue is only a queue of pointers to programs which are contained in the programs vector.

Simulation Runs

The program begins by interfacing with the user through the command line. The user enters the scheduling priority, the number of CPUs to be used in the simulation, and the number of programs to execute. The program then initializes the components of the virtual machine such as the RAM, disk, MMU, and CPUs. Next, the Loader loads all programs into the disk and creates a PCB for each program which it enters the programs metadata into. The Long-Term Scheduler then orders a list of pointers to the PCBs by the given scheduling policy. Next, the Short-Term Scheduler chooses a program from the ready queue or wait queue. If it chooses a program from the ready queue, it means it has not yet been opened and so the first four frames of the process are loaded into memory. The Dispatcher then assigns the process to an idle CPU. If a page fault is generated, the process is moved to the wait queue while the page is loaded into a frame. The main loop continues to cycle until all programs have been executed and terminated.

During the initial simulation runs, we used the file provided which gave descriptions for the first four programs so we understood how the programs should execute. We found that all of the first four programs executed properly.

Once we determined that our virtual machine was fully functional, we ran 6 simulations of all 30 programs. We tested the following scheduling policies for both 1 CPU and 4 CPUs: FCFS, Priority, and SJF. Following this section is the data that we collected from these simulation runs.

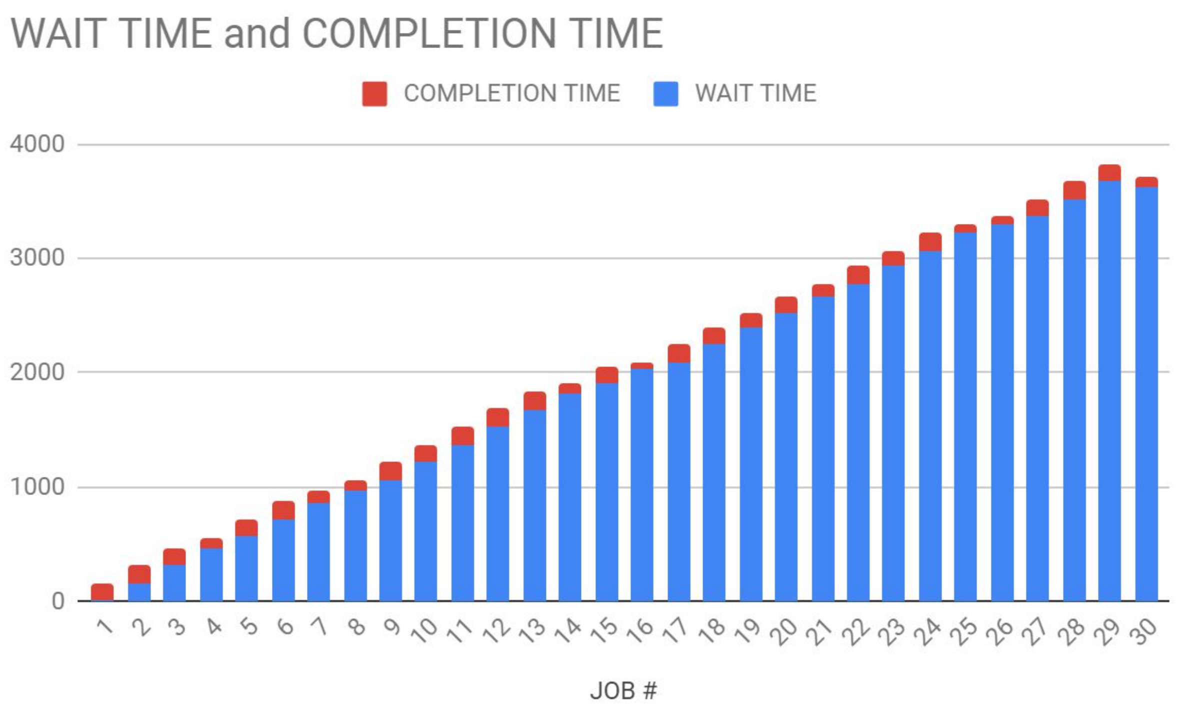
Data Results

Single CPU FCFS

Maximum RAM Usage: 5%

JOB #	Program Size	Priority	WAIT TIME	COMPLETION TIME	IOPS
1	92	2	10	150	17
2	112	4	160	159	16
3	96	6	318	150	16
4	76	5	464	96	17
5	112	3	565	159	16
6	96	7	723	150	16
7	76	5	869	96	17
8	76	16	965	96	17
9	96	2	1065	150	16
10	112	10	1216	159	16
11	92	3	1375	150	17
12	112	9	1525	159	16
13	96	12	1683	150	16
14	76	2	1828	79	14
15	96	5	1912	137	15
16	76	8	2043	38	8
17	112	4	2088	159	16
18	96	12	2246	150	16
19	112	1	2393	129	14
20	96	2	2525	150	16
21	76	4	2671	104	18
22	112	10	2780	159	16
23	92	5	2938	136	16
24	96	7	3074	150	16
25	92	9	3222	82	11
26	76	4	3301	63	12

27	92	9	3370	150	17
28	112	3	3520	159	16
29	96	1	3678	150	16
30	76	8	3624	88	16
		AVG TIME:	1938.366667	130.2333333	

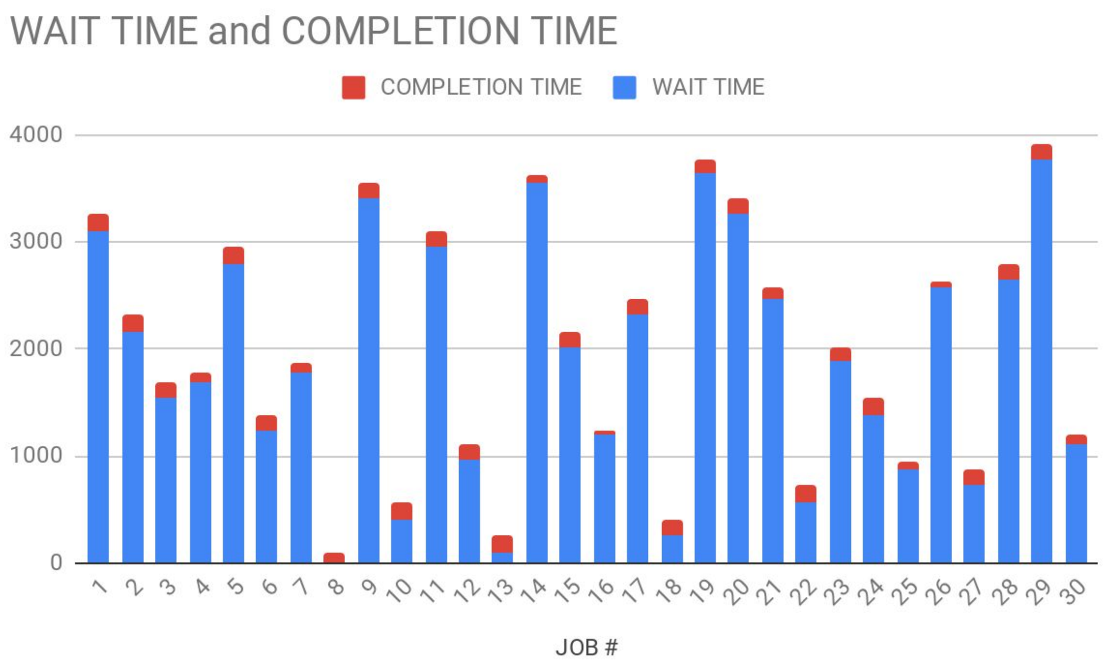


Single CPU Priority

Maximum RAM Usage: 5%

JOB #	Program Size	Priority	WAIT TIME	COMPLETION TIME	IOPS
1	92	2	3109	150	17
2	112	4	2156	159	16
3	96	6	1540	150	16
4	76	5	1686	96	17
5	112	3	2800	159	16
6	96	7	1240	150	16
7	76	5	1782	96	17
8	76	16	5	96	17

9	96	2	3408	150	16
10	112	10	406	159	16
11	92	3	2959	150	17
12	112	9	956	159	16
13	96	12	105	150	16
14	76	2	3553	79	14
15	96	5	2018	137	15
16	76	8	1196	38	8
17	112	4	2315	159	16
18	96	12	255	150	16
19	112	1	3637	129	14
20	96	2	3258	150	16
21	76	4	2469	104	18
22	112	10	565	159	16
23	92	5	1882	136	16
24	96	7	1390	150	16
25	92	9	871	82	11
26	76	4	2572	63	12
27	92	9	724	150	17
28	112	3	2641	159	16
29	96	1	3766	150	16
30	76	8	1110	88	16
		AVG TIME:	1879.13333	130.2333333	

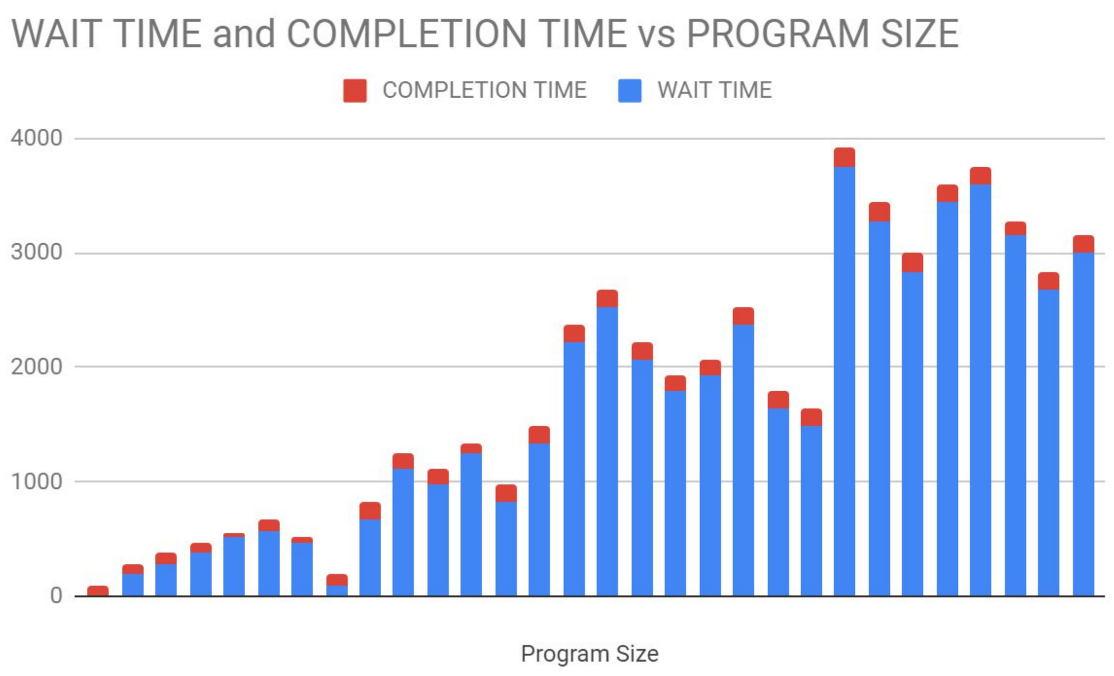


Single CPU SJF

Maximum RAM Usage: 5.46875%

JOB #	Program Size	Priority	WAIT TIME	COMPLETION TIME	IOPS
1	92	2	670	150	17
2	112	4	3758	159	16
3	96	6	1337	150	16
4	76	5	5	96	17
5	112	3	3281	159	16
6	96	7	2224	150	16
7	76	5	189	96	17
8	76	16	285	96	17
9	96	2	2524	150	16
10	112	10	2834	159	16
11	92	3	1106	150	17
12	112	9	3440	159	16
13	96	12	2074	150	16

14	76	2	380	79	14
15	96	5	1787	137	15
16	76	8	521	38	8
17	112	4	3599	159	16
18	96	12	1924	150	16
19	112	1	3151	129	14
20	96	2	2374	150	16
21	76	4	561	104	18
22	112	10	2675	159	16
23	92	5	969	136	16
24	96	7	1637	150	16
25	92	9	1253	82	11
26	76	4	459	63	12
27	92	9	820	150	17
28	112	3	2993	159	16
29	96	1	1487	150	16
30	76	8	101	88	16
		AVG TIME:	1680.6	130.2333333	

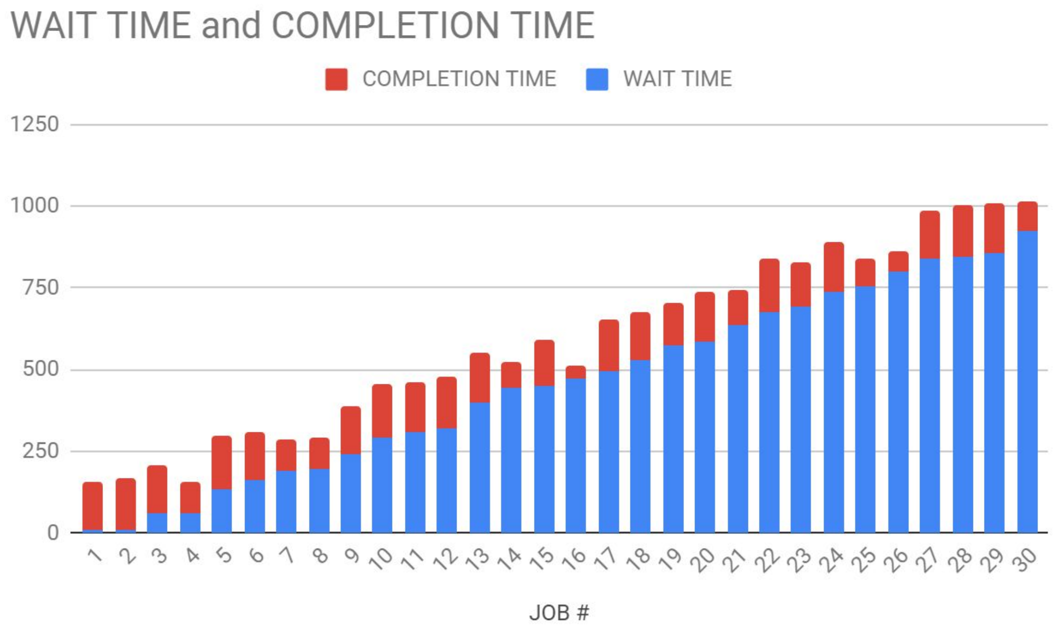


4 CPU FCFS

Maximum RAM Usage: 15.625%

JOB #	Program Size	Priority	WAIT TIME	COMPLETION TIME	IOPS
1	92	2	8	150	17
2	112	4	7	159	16
3	96	6	58	150	16
4	76	5	60	96	17
5	112	3	136	159	16
6	96	7	160	150	16
7	76	5	190	96	17
8	76	16	193	96	17
9	96	2	239	150	16
10	112	10	294	159	16
11	92	3	309	150	17
12	112	9	318	159	16
13	96	12	401	150	16
14	76	2	443	79	14
15	96	5	452	137	15
16	76	8	472	38	8
17	112	4	495	159	16
18	96	12	527	150	16
19	112	1	575	129	14
20	96	2	586	150	16
21	76	4	637	104	18
22	112	10	678	159	16
23	92	5	690	136	16
24	96	7	739	150	16
25	92	9	757	82	11
26	76	4	798	63	12
27	92	9	837	150	17
28	112	3	846	159	16

29	96	1	857	150	16
30	76	8	924	88	16
		AVG TIME:	456.2	130.2333333	



4 CPU Priority

Maximum RAM Usage: 15.625%

JOB #	Program Size	Priority	WAIT TIME	COMPLETION TIME	IOPS
1	92	2	735	150	17
2	112	4	504	159	16
3	96	6	355	150	16
4	76	5	374	96	17
5	112	3	660	159	16
6	96	7	300	150	16
7	76	5	422	96	17
8	76	16	2	96	17
9	96	2	818	150	16
10	112	10	84	159	16

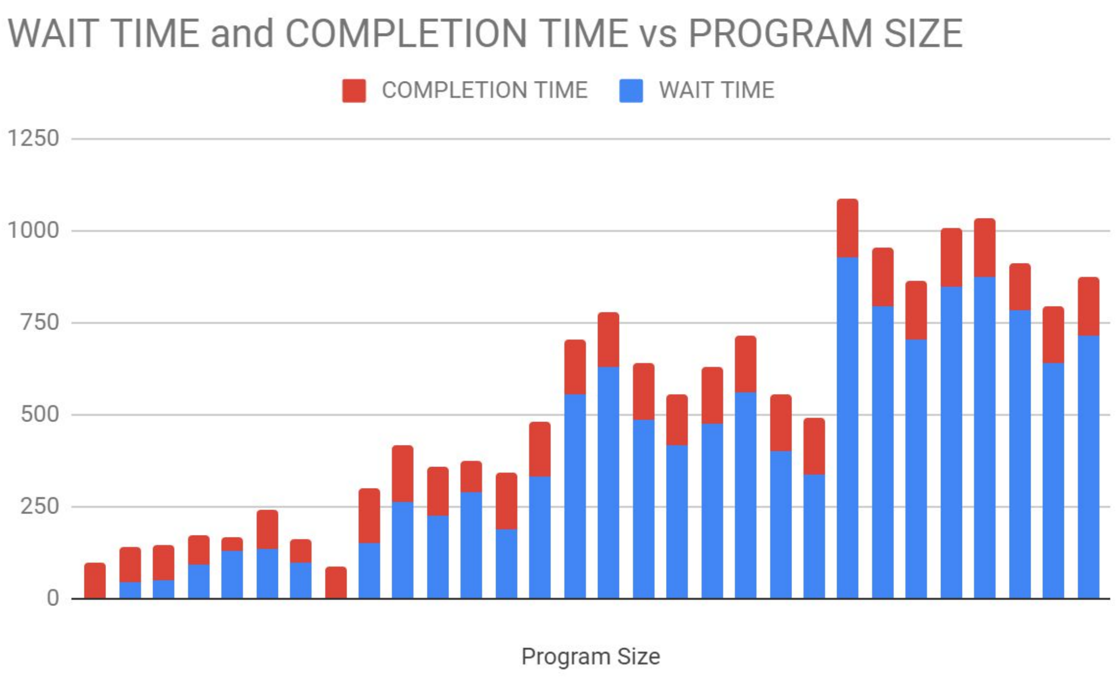
11	92	3	691	150	17
12	112	9	226	159	16
13	96	12	5	150	16
14	76	2	835	79	14
15	96	5	472	137	15
16	76	8	276	38	8
17	112	4	542	159	16
18	96	12	51	150	16
19	112	1	888	129	14
20	96	2	809	150	16
21	76	4	582	104	18
22	112	10	117	159	16
23	92	5	450	136	16
24	96	7	302	150	16
25	92	9	199	82	11
26	76	4	621	63	12
27	92	9	120	150	17
28	112	3	655	159	16
29	96	1	899	150	16
30	76	8	230	88	16
		AVG TIME:	440.8	130.2333333	

4 CPU SJF

Maximum RAM Usage: 10.5469%

JOB #	Program Size	Priority	WAIT TIME	COMPLETION TIME	IOPS
1	92	2	151	150	17
2	112	4	926	159	16
3	96	6	332	150	16
4	76	5	2	96	17
5	112	3	797	159	16
6	96	7	553	150	16
7	76	5	46	96	17
8	76	16	50	96	17
9	96	2	628	150	16
10	112	10	704	159	16
11	92	3	265	150	17
12	112	9	848	159	16
13	96	12	488	150	16
14	76	2	93	79	14

15	96	5	419	137	15
16	76	8	128	38	8
17	112	4	876	159	16
18	96	12	478	150	16
19	112	1	782	129	14
20	96	2	563	150	16
21	76	4	135	104	18
22	112	10	638	159	16
23	92	5	225	136	16
24	96	7	403	150	16
25	92	9	292	82	11
26	76	4	97	63	12
27	92	9	190	150	17
28	112	3	717	159	16
29	96	1	339	150	16
30	76	8	2	88	16
		AVG TIME:	405.56666	130.2333333	



Conclusion

From our simulation runs, we found that the multi-CPU simulation runs had much greater performance and executed significantly faster than the single-CPU runs. For both the single-CPU and multi-CPU runs, we tested 3 scheduling policies: First Come First Serve (FCFS), Priority, and Shortest Job First (SJF). In comparing the total execution time for different scheduling policies while using the same number of CPUs, we found that the total execution time was nearly the same. This makes sense as the number of instructions remains constant and no CPUs are left idle when processes are available to be run. Overall, we found SJF to be best as it completed the greatest number of programs in the shortest amount of time since shorter programs are executed first. In analyzing FCFS scheduling, we saw an almost linear increase in completion time vs job number. We did not find any significant data in analyzing priority scheduling as the priority seemed to be arbitrary and unrelated to the program's properties. We also did not find any significant data in analyzing completion time vs I/O operations since total program size varied greatly.