Yeezus Operating System

Project Report

Version 2.0

Mark Zeagler, Jessica Brummel,

Nick Curtin, Pablo Avecilla

Kennesaw State University  
CS 3502 Operating Systems  
Professor Patrick Bobbie  
4/18/2018

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 4 Feb 2018 | SRS 1.0 | Initial SRS setup | Mark Zeagler |
| 21 Mar 2018 | PR 0.1 | Conversion to Project Report | Jessica Brummel |
| 25 Mar 2018 | PR 1.0 | Project Report 1 finalization | Jessica Brummel |
| 19 Apr 2018 | PR 2.0 | Project Report 2 finalization |  |

Table of Contents

1. Abstract 5

2. Introduction 6

3. Overall Description 7

4. Specific Requirements 8

4.1 The Memory System 8

4.2 The Driver Overview 8

4.2.1 Driver 9

4.2.2 Loader 9

4.2.3 Scheduler 11

4.2.4 Dispatcher 12

4.2.5 Memory 12

4.2.6 Effective-Address 13

4.2.7 Fetch 13

4.2.8 Decode 13

4.2.9 Execute 14

4.3 CPU 14

4.3.1 DMA-Channel 14

4.3.2 ComputeOnly 15

4.4 Multiprocessor Architecture 16

4.4.1 Multiprocessor Memory Management 17

4.4.2 Multiprocessor Program Cache 17

4.5 Shortest Job First 17

4.6 Paging and I/O Blocking 17

4.6.1 Page Table 18

5. Design & Implementation 21

5.1 Memory 21

5.2 Driver 22

5.2.1 Loader 22

5.2.2 Scheduler 23

5.2.3 Dispatcher 25

5.3 CPU 27

5.3.1 Fetch 27

5.3.2 Decode 27

5.3.3 Execute 27

5.3.4 DMA-Channel 27

5.4 Multiprocessor Architecture 28

5.4.1 Multiprocessor Memory Management 28

5.4.2 Multiprocessor Program Cache 28

5.5 Shortest Job First 29

5.6 Paging and I/O Blocking 29

5.6.1 Paging 29

5.6.2 I/O Blocking 29

6. Simulation Discussion 30

7. Data Analysis 31

7.1 FCFS and Priority Comparison 31

7.2 Introduction of Cache and Comparison 31

7.3 Percentage of RAM and Cache Used 32

7.4 Comparison of Performance for 1-CPU vs N-CPU Runs 32

7.5 FCFS, Priority, and SJF Data Comparison 33

7.6 Understanding the Effects of Paging 34

8. Conclusions 35

8.1 Phase 1 Conclusions 35

8.2 Phase 2 Conclusions 35

9. Supporting Information 37

9.1 Table of Figures 37

9.2 Appendix A: Instruction Format 38

9.3 Appendix B: Instruction Set 40

9.4 Appendix C: Process Data Table (FCFS, 1 Thread, No Cache) 41

9.5 Appendix D: Process Data Table (Priority, 1 Thread, No Cache) 42

9.6 Appendix E: Process Data Table (FCFS, 1 Thread, Cache) 43

9.7 Appendix F: Process Data Table (Priority, 1 Thread, Cache) 44

9.8 Appendix G: Process Data Table (FCFS, 4 Threads, Cache) 45

9.9 Appendix H: Process Data Table (Priority, 4 Threads, Cache) 46

9.10 Appendix I: Process Data Table (SJF, 1 Thread, Cache) 47

9.11 Appendix J: Process Data Table (SJF, 4 Threads, Cache) 48

9.12 Appendix K: Process Data Table (SJF, 4 Threads, Cache, Paging) 49

9.13 Appendix L: Process Data Table (Priority, 4 Threads, Cache, Paging) 50

9.14 Appendix M: Process Data Table (FCFS, 4 Threads, Cache, Paging) 51

9.15 Appendix N: Comparison of Page Faults and Average Page Fault Servicing Time in ns for FCFS, Priority, and SJF on One CPU 52

9.16 Appendix O: Comparison of Running Times for FCFS, Priority, and SJF on One CPU 53

9.17 Appendix P: Comparison of Waiting Times for FCFS, Priority, SJF on One CPU 54

9.18 Appendix Q: Comparison of Running Times for FCFS, Priority, SJF on Four CPUs 55

9.19 Appendix R: Comparison of Waiting Times for FCFS, Priority, SJF on Four CPUs 56

9.20 Appendix S: Comparison of Completion Times for FCFS, Priority, SJF on Four CPUs 57

9.21 Appendix T: Comparison of Waiting Times for FCFS and Priority and the Effects of Cache 58

9.22 Appendix U: Comparison of Running Times for FCFS and Priority and the Effects of Cache 59

Appendix V: Comparison of Running Times for FCFS and Priority and the Effects of Cache 60

# Abstract

The OS Project consisted of two phases. Phase 1 consisted of creating a barebones operating system (OS) that consists of a memory system, a driver, a scheduler, a loader, and a dispatcher. The OS also had to be able to use effective-address, fetch, decode, execute instructions, and handle I/O through a DMA-Channel. The OS had to be able to load in 30 jobs, their instructions, their data, execute the instructions, and handle the I/O. Phase 1 also introduced the concepts of a multiprocessor system, the use of cache, and the concepts of two scheduling algorithms: first come first serve (FCFS) and priority scheduling. It was concluded in Phase 1 that priority scheduling had shorter overall wait times and completion times than FCFS for each job. Furthermore, having N-CPUs is better than having a single CPU, as the wait times and completion times overall decrease as the number of CPUs increase. Finally, cache memory, or fast memory, decreased the overall wait times and completion times of each job.

Phase 2 was built on the structures developed in Phase 1. To further the research in how operating systems worked, another scheduling algorithm: shortest job first (SJF) was included. This scheduling algorithm used the number of instructions as an estimate on how long the job would take to run. It was found that SJF had similar waiting times and completion times as Priority, most likely because SJF is an alteration of Priority scheduling, where the priority is how long processes will take to run. The last addition to the project was the implementation of paging and the effects it had on overall throughput of the system. Paging actually had a negative impact on the overall times for all three scheduling algorithms. This can be attributed to the fact that each process had a high number of page faults, and thus the system had to spend a good amount of time servicing each page fault.

# Introduction

The following is the Software Requirements Specification document for the CS3502, Operating Systems Term Project for the Spring 2018 semester under the guidance of our professor, Dr. Patrick Bobbie. This document will feature the explanation of the modules, the design of the system architecture, the data collected by Phase 1 & 2, and an analysis of the data. Additionally, there are several provided graphics or pseudocode which are in figures, and these will be indicated within the figure description.

This project has been broken into 2 phases consisting of 2 parts each. This document features all parts and phases. The goal for Phase 1 Part 1 is to implement an operating system with a single CPU that can load and execute a set of jobs written in hex code. The goal for Phase 2 Part 2 is to implement an operating system featuring four CPUs that can execute the same set of jobs and include a program cache.

The semester course project is on the design and implementation of an OS simulator. The project is divided into two phases. Each Phase has two parts. The requirements are discussed below.

# Overall Description

In this semester long project, you are required to design and implement a complete virtual machine, with its own virtual CPU and a control system (or Operating System), together making a Simulator. The given tasks, or user programs/processes, will run on the virtual CPU. The CPU’s architecture and instruction repertoire are posted in the Instruction Format and Instruction Set files in the Project Folder. You will design and implement a number of program modules in a high-level language of your choice, e.g., Java, C, C#, to complete this OS Project.

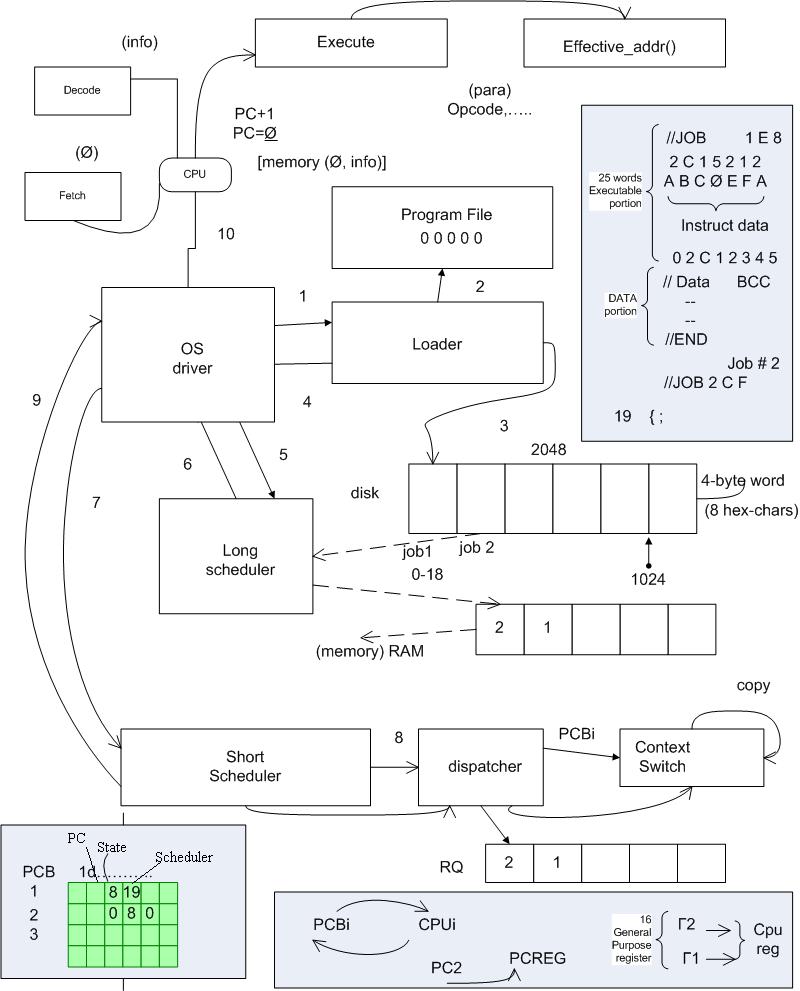


Figure 1 Overview of the Operating System components (provided by Dr. Bobbie).

# Specific Requirements

Note: The elaboration and specification of individual requirements was halted due to time constraints, but the given instructions are included for context.

## The Memory System

The memory hierarchy comprises a set of simulated registers, a program/data simulated RAM, called memory and a simulated hard drive called disk. The contents of the disk and RAM are hex characters; and of sizes 2048 and 1024 words, respectively. (Each word is 4 bytes or 8 hex characters long.)

Ancillary programs to dispatch processes/programs from disk to RAM, to compute effective addresses, to access memory, to fetch instructions and decode instructions will also be needed. Additional support programs for conversions between hex, decimal, and binary numbers will be necessary.

**ID: Memory Requirement (MR) 1**

TITLE: Registers

DESC: The system must have 16 registers of 32-bits each.

**ID: MR 2**

TITLE: RAM

DESC: The system must be able to store 1024 words in RAM.

**ID: MR 3**

TITLE: Disk

DESC: The system must be able to store 2048 words on the disk.

## The Driver Overview

The OS driver constitutes the core, or the main thread, of the simulator. Its function is to simply call the loader to load user programs, or jobs, which are already assembled (given as a stream of hex character) and stored in a ‘program-file.’ The program-file, named as DataFile, is in the Project Folder as well. The Driver calls the Scheduler to select a user program from a list of ‘ready’ jobs for dispatch. The Dispatcher places the selected job or program in an ‘execution context’ to run on the virtual CPU. The CPU executes the programs in the simulated RAM. Generally, if a program/job is interrupted while on the CPU, the interrupt is serviced while the job is suspended. After the service is done, the job must be returned to the ‘ready-state’, at some point in time. When a program completes, the Scheduler and the Dispatcher process the next program. This cycle continues forever or until your simulator is shut down when no more user programs are pending.

The following program sketch should help you design and implement the Driver module (for a single batched system):

*Driver {*

*loader();*

*loop*

*scheduler();*

*dispatcher();*

*CPU();*

*waitforinterrupt();*

*endloop;*

*}*

Figure 2 Overview of the Driver's flow (provided by Dr. Bobbie).

### Driver

**ID: Driver Requirement (DR) 1**

TITLE: Driver

DESC: The System must have a Driver to loop through system processes.

**ID: DR 2**

TITLE: Driver Load

DESC: The Driver must be loaded and executed at system boot.

**ID: DR 3**

TITLE: Call the Loader

DESC: The Driver must call the loader before the loop.

**ID: DR 4**

TITLE: Call the Scheduler

DESC: The Driver must call the Scheduler.

**ID: DR 5**

TITLE: Call the Dispatcher

DESC: The Driver must call the Dispatcher.

**ID: DR 6**

TITLE: Driver Loop

DESC: The Driver must loop through calling the Scheduler, calling the Loader, executing the process, and handling interrupts.

**ID: DR 9**

TITLE: Shutdown

DESC: The Driver must halt its loop when all processes are complete.

### Loader

The loader module opens (once at the start) the ‘program-file’ and performs the loading process. Programs are loaded into disk according to the format of the batched user programs in the program-file. Ancillary programs would be needed to process (strip off) the special control “cards” – which start with ‘//’. For example, the ‘// Job 1 17 2’ control card of Job1 is processed by discarding the ‘//’, noting that the ‘1’ is the ID: of the first job, noting that ‘17’ (or 23 in decimal) is the number of words that constitute the instructions of Job 1, and ‘2’ is the priority-number (to be used for scheduling) of Job 1. All the numbers are in hex. Following the Job-control card are the actual instructions – one instruction per line in the program-file, which must also be extracted and stored in disk.

Similar logic for processing the data-section, following the instructions and proceeded by ‘// Data …’ control cards, also applies. In the case of Job 1, for example, ‘// Data 14 C C’, means Job 1’s input buffer is 20 (14 in hex), its output buffer size is 12 (C in hex) words, and the size of its temporary buffer is 12 (C in hex) words. (In this simulation, the input buffer comes pre-loaded with the input data, for simplicity.) All the data values on the control cards are attributes of each program, must be extracted and stored in the Process Control Block (PCB) (see below).

The basic outline of the loader’s logic looks like the following:

*while (not end-of-program-data-file) do {*

*Read-File();*

*Extract program attributes into the PCB*

*Insert hex-code or instructions into the simulated RAM*

*}*

Figure 3 Outline of the loader’s logic (provided by Dr. Bobbie).

#### **ID: Loader Requirement (LR) 1**

TITLE: Read Files

DESC: The Loader must read files as Strings from the Program-File.txt file.

**ID: LR 1.1**

TITLE: Read Cards

DESC: The Loader must read control data from any line of the source file which begins with “//”.

**ID: LR 1.1.1**

TITLE: Read Cards – Job #

DESC: The Loader must retrieve the Job # from the cards in the source file.

**ID: LR 1.1.2**

TITLE: Read Cards – Job Length

DESC: The Loader must retrieve the Program Length (written in hex) from the source file.

**ID: LR 1.3**

TITLE: Read Cards – Job Priority

DESC: The Loader must retrieve the Job Priority (written in hex) from the source file.

**ID: LR 1.1.4**

TITLE: Read Cards – Data Input Buffer

DESC: The Loader must retrieve the Data Input Buffer (written in hex) from the source file.

**ID: LR 1.1.5**

TITLE: Read Cards – Data Output Buffer

DESC: The Loader must retrieve the Data Output Buffer (written in hex) from the source file.

**ID: LR 1.1.6**

TITLE: Read Cards – Data Temporary Buffer

DESC: The Loader must retrieve the Data Temporary Buffer (written in hex) from the source file.

**ID: LR 1.2**

TITLE: Read Data

DESC: The Loader must read Words from the source file.

**ID: LR 1.2.1**

TITLE: Read from String

DESC: The Loader must read the data as hex written in Strings. Ex: “0x01234567” (one Word per line).

**ID: LR 2**

TITLE: Load to Disk

DESC: The Loader must load the programs from the file into the disk.

**ID: LR 3**

TITLE: Read Sequence

DESC: The Loader will first read the Control Card, the instructions, The Data Card, then the data for each program.

**ID: LR 4**

TITLE: Annotate PCB

DESC: The Loader must fill in data to the PCB as it loads programs.

### Scheduler

The Scheduler loads programs from the disk into RAM according to the given scheduling policy. The scheduler must note in the PCB, which physical addresses in RAM each program/job begins, and ends. This ‘start’ address must be stored in the base-register (or program-counter) of the job). The Scheduler module must also use the Input/Output buffer size information (extracted from the control cards) for allocating spaces in RAM for the input and output data. It may be instructive to store the start addresses of the input-buffer and output-buffer spaces allocated in RAM as well. (Note that a job’s program-counter, which is a component of the PCB, is different from the virtual CPU’s program-counter – see below). The Scheduler module either loads one program or multiple programs at a time (in a multiprogramming system). Thus, the Scheduler works closely with the Memory manager and the Effective-Address method to load jobs into RAM.

**ID: Scheduler Requirement (SR) 1**

TITLE: Determine Process Priority

DESC: The Scheduler must read the PCB to determine the process with the highest priority.

**ID: SR 2**

TITLE: Interpret PCB Data

DESC: The Scheduler must correctly interpret the Process data within the PCB to ensure the correct information is loaded into the RAM.

**ID: SR 2.1**

TITLE: Interpret PCB Data – Program Counter

DESC: The Scheduler must read the Program Counter to determine which Instruction will be the first Instruction to be loaded into the Registers.

**ID: SR 2.2**

TITLE: Interpret PCB Data – Buffer Size

DESC: The Scheduler must determine the Input/Output buffer sizes of the loaded Process.

**ID: SR 2.2.1**

TITLE: Reserve Buffer

DESC: The Scheduler must reserve the appropriate amount of buffer within the RAM for the loaded Process.

**ID: SR 3**

TITLE: Load the Process into RAM

DESC: The Scheduler must load the process of the highest priority into RAM using the Ready Queue and PCB.

**ID: SR 4**

TITLE: Annotate PCB

DESC: The Scheduler will load the status changes in the PCB.

### Dispatcher

The Dispatcher method assigns a process to the CPU. It is also responsible for context switching of jobs when necessary (more on this later!). For now, the dispatcher will extract parameter data from the PCB and accordingly set the CPU’s PC, and other registers, before the OS calls the CPU to execute the job.

**ID: Dispatcher Requirement (DiR) 1**

TITLE: Load the Process

DESC: The Dispatcher must load the Process into the appropriate Register.

### Memory

This method is the only module of your simulator by which RAM can be accessed. A known absolute/physical address must always be passed to this method. The Memory simply fetches an instruction or datum or writes datum into RAM (or cache – more on this later!).

**ID: MR 4**

TITLE: RAM Access

DESC: The RAM must be accessible only via physical address.

**ID: MR 5**

TITLE: RAM Read

DESC: The RAM must be readable at a given physical address.

**ID: MR 6**

TITLE: RAM Write

DESC: The RAM must be writable at a given physical address.

### Effective-Address

This method takes a logical address and returns the corresponding absolute/physical address for the calling unit (e.g., the CPU). The Effective-Address method also supports the Fetch and Decode methods – a part of the CPU, during instruction fetch-decode-execute cycle. The Effective-Address supports two kinds of address translations – direct and indirect, using the index register.

The basic steps for calculating the effective addresses are:

direct addressing: EA = C(base-reg)+ D; // D is the 16-bit offset or displacement

indirect addressing: EA = C(base-reg) + C(I-reg) + D;

**ID: Effective-Address Requirement (ER) 1**

TITLE: Convert Process Address

DESC: The Effective-Address mechanism must convert a Process’s local Logical Address into the corresponding Physical Address in RAM.

### Fetch

With support from the Memory module/method, this method fetches instructions or data from RAM depending on the content of the CPU’s program counter (PC). On instruction fetch, the PC value should point to the next instruction to be fetched. The Fetch method therefore calls the Effective-Address method to translate the logical address to the corresponding absolute address, using the base-register value and a ‘calculated’ offset/address displacement. The Fetch, therefore, also supports the Decode method of the CPU.

**ID: Fetch Requirement (FR) 1**

TITLE: Fill Registers

DESC: The CPU must Fetch instructions as needed to ensure continuous operation.

**ID: FR 2**

TITLE: Read CPU’s PC

DESC: The Fetch mechanism must interpret the CPU’s Program Counter (PC) to ensure that the correct instructions are read into the registers.

**ID: FR 3**

TITLE: Interpret PC

DESC: The Fetch mechanism must work with the Effective-Address mechanism to ensure that the program’s logical PC is interpreted into a usable physical address.

### Decode

The Decode method is a part of the CPU. Its function is to completely decode a fetched instruction – using the different kinds of address translation schemes of the CPU architecture. (See the supplementary information in the file: Instruction Format.) On decoding, the needed parameters must be loaded into the appropriate registers or data structures pertaining to the program/job and readied for the Execute method to function properly.

**ID: Decode Requirement (DeR) 1**

TITLE: Interpret Instructions

DESC: The Decoder must read Words and interpret Instructions from them. Appendices A and B elaborate on the instructions and how they are to be decoded.

**ID: DeR 2**

TITLE: Relay Instructions

DESC: The Decoder must relay operable Instructions to the Execute mechanism.

### Execute

This method is essentially a switch-loop of the CPU. One of its key functions is to increment the PC value on ‘successful’ execution of the current instruction. Note also that if an I/O operation is done via an interrupt, or due to any other preemptive instruction, the job is suspended until the DMA-Channel method completes the read/write operation, or the interrupt is serviced.

**ID: Execute Requirement (XR) 1**

TITLE: Execute

DESC: The Execute mechanism must take the appropriate actions as dictated by the Decoder. Appendix B elaborates on the instructions and how they are to be executed.

## CPU

In this Part of the simulation you are going to separate ‘compute-only’ instructions from ‘I/O’ instructions. Thus, we envision an implementation of two concurrent threads – one to handle each type of instruction. We first discuss the logic of the DMA-Channel for handling I/O instructions.

### DMA-Channel

In small systems with programmed I/O interfaces using interrupts, the CPU can be employed to service slow character-oriented devices since it can service thousands of compute-instructions between any two I/O commands. However, for block-oriented devices, e.g., disk or RAM I/O, it is desirable to delegate a separate device, e.g., the disk channel controller, to work concurrently with the CPU. In this way, the disk device-channel controller works independently on I/O requests, which frees up the CPU to focus on compute-only instructions.

For this to work in DMA-mode, the virtual CPU calls the two routines to perform Read and Write from/to RAM when an I/O instruction is encountered:

Read(ch, next(p\_rec), buf[next\_io]);

Write(ch, next(p\_rec), buf[next\_io]);

where ch is the channel or DMA controller, p\_rec is the RAM address of the physical data to be transferred, and buf is the starting address of a RAM buffer into/from which the data is to be transferred.

The heart of the DMA-channel module/thread will look like the following:

*DMA () {*

*loop*

*switch(type) {*

*case 0: Read(ch, next(p\_rec), buf[next\_io]);*

*case 1: Write(ch, next(p\_rec), buf[next\_io]);*

*}*

*next\_io := next\_io + 4; // assuming 1 word of 4 bytes at a time*

*end; //loop*

*signal(ComputeOnly) // signal the main (virtual) CPU to regain the channel/bus*

*}*

Figure 4 Heart of the DMA-channel module/thread (provided by Dr. Bobbie).

### ComputeOnly

This method, or module, implements a simple instruction cycle algorithm with dynamic relocation of the program (relative to the base-register).

*loop*

*ir : = Fetch(memory[map(PC)]); // fetch instruction at RAM address – mapped PC*

*Decode(ir, oc, addrptr); // part of decoding of the instruction in instr reg (ir), returning the opcode*

*// (oc) and a pointer to a list of significant addresses in ‘ir’ – saved*

*// elsewhere*

*PC := PC + 1; // ready for next instruction, increase PC by 1 (word)*

*Execute(oc) {*

*case 0: // corresponding code using addrptr of operands*

*case 1: // corresponding code or send interrupt*

*…*

*}*

*end; // loop*

Figure 5 Overview of the CPU’s execution (provided by Dr. Bobbie).

**ID: CPU Requirement (CR) 1**

TITLE: Execute Instructions

DESC: The CPU must oversee the decoding and execution of the instructions.

The CPU is supported by a PCB, which may have the following (suggested) structure:

*typedef struct PCB {*

*cpuid: // information the assigned CPU (for multiprocessor system)*

*program-counter // the job’s pc holds the address of the instruction to fetch*

*struct state: // record of environment that is saved on interrupt*

*// including the pc, registers, permissions, buffers, caches, active*

*// pages/blocks*

*code-size; // extracted from the //JOB control line*

*struct registers: // accumulators, index, general*

*struct sched: // burst-time, priority, queue-type, time-slice, remain-time*

*struct accounts: // cpu-time, time-limit, time-delays, start/end times, io-times*

*struct memories: // page-table-base, pages, page-size*

*// base-registers – logical/physical map, limit-reg*

*struct progeny: // child-procid, child-code-pointers*

*parent: ptr; // pointer to parent (if this process is spawned, else ‘null’)*

*struct resources: // file-pointers, io-devices – unitclass, unit#, open-file-tables*

*status; // {running, ready, blocked, new}*

*status\_info: // pointer to ‘ready-list of active processes’ or*

*// ‘resource-list on blocked processes’*

*priority: integer; // of the process, extracted from the //JOB control line*

*}*

Figure 6 The suggested structure of the PCB (provided by Dr. Bobbie).

**ID: PCB Requirement (PR) 1**

TITLE: PCB

DESC: The system must have a Process Control Board (PCB) that stores and organizes information about running processes.

**ID: PR 2**

TITLE: PCB Availability

DESC: The PCB must be made available to be edited or read by the Loader, Scheduler, Dispatcher, Fetch mechanism, and Effective-Address mechanism.

## Multiprocessor Architecture

The virtual CPU for Part 1 is designed with distributed, or parallel, computing architecture in mind. The core simulation system maintains the memory and the various queues as well as the PCB. You will need to clone your single virtual CPU from Part 1 into an N-CPU system, to achieve a N-processor platform. We are considering a four-node architecture in Part 2. A detailed explanation of the workings of the system is given below:

The loader loads all the programs into disk and the scheduler is called to load the programs into the simulated RAM, as discussed in the Loader and Scheduler sections above.

The multiprocessor dispatcher (m-dispatcher), which extends the single-CPU dispatcher described above, accesses the ready queue and assigns the jobs to available CPU’s in order. Any process can be assigned to any available CPU. The m-dispatcher makes note of which segment of RAM or memory space is assigned to each CPU, and the CPU can only access that part of the RAM. Thus, the CPU can access instructions & data specified in only its part of the RAM. Similarly, output can be written to only the assigned part of the RAM for a given process. Care must be taken to ensure that the CPU does not modify or access memory outside its assigned space. If a process tries to access memory outside its assigned range, a trap is generated and the program is aborted.

Each CPU fetches each instruction from its assigned cache, decodes the instruction and executes it. On end of the program (or trap), the CPU signals the scheduler about the end of the program (or trap), and it is assigned the next process from the single/shared ready queue. (We are assuming an ‘asymmetric’ multi-processor scheduling system.)

To facilitate the design and minimize the overhead/delay due to bus contention, caches are to be used in each CPU. Each cache must be equivalent in size to the largest job size. A short-term loader module must be written to support the swapping of instructions/data between the RAM and the caches.

### Multiprocessor Memory Management

As mentioned above, the memory is maintained by the system (simulator). Therefore, appropriate semaphores and locks must be used to access memory. When a process needs to access memory, it must first acquire a lock and while it has the lock, no other process should be able to access even its portion of memory. (This is the price one pays for a shared memory architecture; but it could be implemented more efficiently. For simplicity, we are assuming this architecture.)

### Multiprocessor Program Cache

As indicated, each CPU maintains a program cache (for instructions and data). Thus, as the m-dispatcher assigns a process to a CPU, the whole process is loaded into the (unified) program cache of the CPU by the short-term loader, for simplicity. A CPU fetches each instruction from its assigned cache, decodes the instruction and executes it. On the execution of instructions, a note is made of which addresses in the cache, and the corresponding addresses in memory (your virtual RAM) that are being modified. For example, if an instruction specifies a ‘write’ to memory, the appropriate output-buffer section of the cache is rather changed. Also the address of the output-buffer section to be changed in memory is noted. At the end of the process execution, only the modified words are written back from the cache to memory. The scheduler is signaled about the termination of each process so that the CPU can be assigned the next process from the ready queue. This cycle continues until all the programs on disk are executed or until the ready queue becomes empty.

## Shortest Job First

As done in Phase 1, the processes from the disk would be moved into memory (or your simulated RAM), ordered by SJF policy, and then into the designated caches for execution. Under the SJF, the short-term scheduler will select the shortest process from the ready queue and then dispatch them to the CPU. Both the 1-CPU and N-CPU versions of the simulator are to be run for this Step 1.

## Paging and I/O Blocking

The second step involves the design and implementation of a paging system. This will require the generation of page fault with process-blocking and servicing of faults. This step will also include process-blocking due to I/O and servicing of I/O requests.

The simulated memory must now be broken into a pool of fixed-sized blocks called frames and the disk must also be broken into blocks of same sizes called pages. Thus, pages and frames are of same sizes: 4 words. Each process’s cache is now reorganized as a set of caches: cache-frames for input-data, cache-frames for instructions, and cache-frames for output data.

Each process will be initially assigned four (4) memory frames (4-words per frame), and any additional requests will be honored by a page-manager, depending on availability from the pool. When no more frames are available a process must wait until more frames are returned to the pool (by terminating processes). Remember that memory or RAM is still shared. Process instruction and data must still be loaded into the corresponding caches for the CPU to process them. [We are skipping page replacement implementation in Phase 2 but taking care of page fault servicing.]

### Page Table

You will have to implement a page table. The address generated by the CPU is divided into page number (p) and page offset (d). The page number is used as an index to the page table. The page table contains the base address of each page in physical memory. The base address is combined with the page-offset to define the physical memory that is sent to the memory unit.

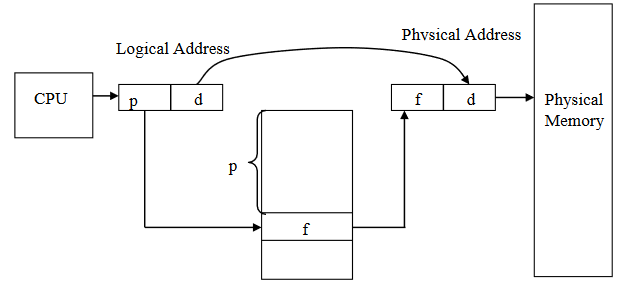


Figure 7 Diagram of a Page Table (provided by Dr. Bobbie).

The page table is kept as a part of the PCB and a page-table base register (PTBR), which is also a component of the PCB, points to the page table. A valid-invalid bit should be associated with every entry in the page table. The status of the bit is used to determine if a page fault interrupt should be generated or not. Thus, access to a page marked invalid causes a page fault.

Three major components of the page fault service overhead are:

1. Service the page fault interrupt
2. Read in the page into the physical memory (and write a modified frame back to disk)
3. Restart the instruction that caused the fault

Things to take care of while servicing the fault

1. Generate an interrupt when a page is not found in physical memory (or invalid bit is set)
2. Save the state and registers of the current process, which caused the interrupt into the PCB
3. Determine the interrupt type, i.e., if it were a page fault or an I/O request
4. Check if the page address is legal, then determine the location of the page in memory
5. The process moves into the wait state, issues a read for the desired page from memory and waits until the page is transferred into a free frame (allocated form the pool)
6. While waiting, the CPU can be allocated to the next process according to the CPU scheduling algorithm. When a process moves into the wait state, the appropriate PCB info must be changed or modified [Note: Generate an interrupt to bring in a new process and restore its PCB state]. The following state diagram depicts the logical flow of process transitions in the system.

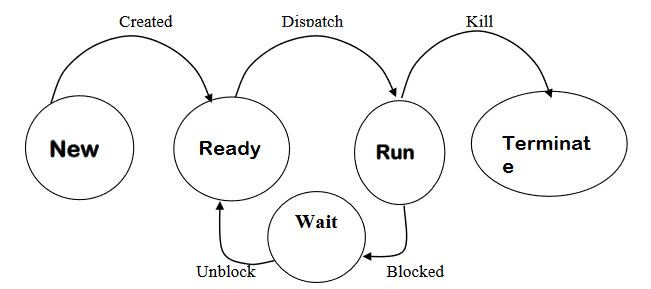


Figure 8 Diagram of process states (provided by Dr. Bobbie).

1. Determine the interrupt type (I/O or page fault service completion)
2. Update the page table, showing that the page is now in memory
3. Change the appropriate PCB info and add this faulting process to the ready queue

Context switching in this step will require

1. Interrupt generation
2. Swapping of pages, if required.

The short-term loader module must be written to support the updating and swapping of instructions/data between memory and the caches during context-switches or temporary blocking on interrupts.

Each CPU can be assigned any process such that when the process is blocked or context-switched, it may not be reassigned to the same CPU. Therefore, on context switch, it is imperative to write back the modified words of the process into its cache – particularly, the output-buffer cache. At all times, the PCB must be updated to reflect the state of the process currently on the CPU.

# Design & Implementation

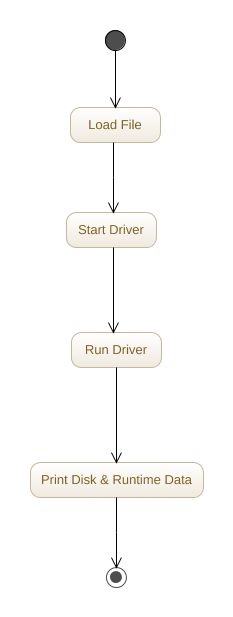


Figure 9 System overview diagram.

The design and implementation processes were not discrete and experienced a lot of back-and-forth interaction, so the two sections have been combined. Many times, the implementation was initiated before the design was complete in order to explore options that would then dictate design choices, and other times a design was found to be impossible or inadvisable to implement and would have to be adapted.

The early consensus was that our operating system would be developed in the Java programming language, due to its familiarity to all on the team. This decision helped drive the system design as a modular operating system, made up of several objects communicating and working together. Each major component was made into its own object, with the heart of the system, the driver, controlling every component of the system. The driver would load the file, start the Scheduler, followed by the Dispatcher, then would signal the CPU to execute. When instantiating the representative objects for each of these components, the driver would ensure that they had the necessary memory objects to correctly accomplish their job. Finally, once the system had completed, the driver ensures that each component is signaled, or controlled into their terminal tasks before finally relinquishing control to its parent process for output of the data.

## Memory

The basic memory system is broken down into 2 parts, the data itself and the storage. The data is given to the system as hexadecimal, 4-byte values written in text. In order to turn this into something that can be easily used by the operating system, these text values are read in as strings, then passed to the Word objects, which convert the strings into long integers for more memory-efficient storage. The long integer was chosen over the regular integer due to constraints of the Java programming language. While an integer is large enough to contain all of the data necessary, Java’s integers are signed, and are therefore 1 bit short. The long integer, though far too large, was the next best option. The stored integer can easily be converted back to a string when printed, overriding the toString() method in Java, or it can be retrieved as its integer value for easy manipulation and calculation.

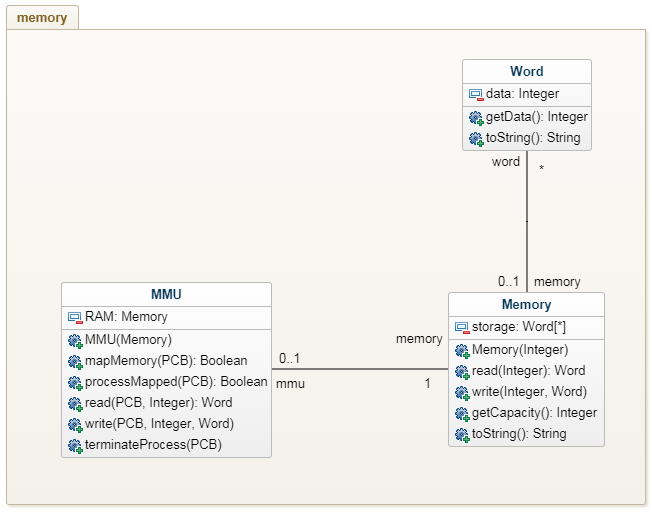


Figure 10 UML class diagram of the memory system.

The storage of the system uses an array of these Word objects. The array was chosen because it allows random access, and its fixed size would not be an issue in our system, where the size of the various memory devices does not change after it has been set.

## Driver

The driver, from a distance, is very simple. It initiates the loader, and cycles through the systems required to power the operating system. However, there are many nuances that had to be added as the system was implemented. Early designs revolved around developing a symmetrical system, where instead of creating multiple CPUs, multiple drivers would be created, and each CPU would be created and controlled by a driver. However, this was adjusted to an asymmetric system, though many design choices of the symmetrical system remained. For instance, the Driver.loadFile() static method was developed to ensure that only 1 driver attempted to load the file into the disk, and also the driver constructors are meant to check if the file has been loaded and throw an exception if not. These artifacts remain, though made irrelevant in the new system design.

Furthermore, there are intricacies that had to be added concerning the shutting down of the system and writing the data back to the disk. A task manager was developed to aid the driver in controlling the PCBs, and quickly became essential to the operation of the Loader, Scheduler, and Dispatcher.

### Loader

The nature of the loader changed several times through the design process and could still perhaps use further improvements. The loader is vitally important in its relationship to the driver and the system’s threading architecture. If it is called by the driver during the driver’s instantiation, or perhaps during later setup, in a symmetric system, there’s a potential for every driver to call the loader, resulting in wasted executions, or even in corrupted data on the disk. In the initial design pass, the loader would be called within the driver constructor, and checked against a static flag within the driver to determine whether or not the loader had already been run. To simplify things, the loader was removed from the driver constructor, and was instead called via a static class method that must be run prior to a Driver’s instantiation.

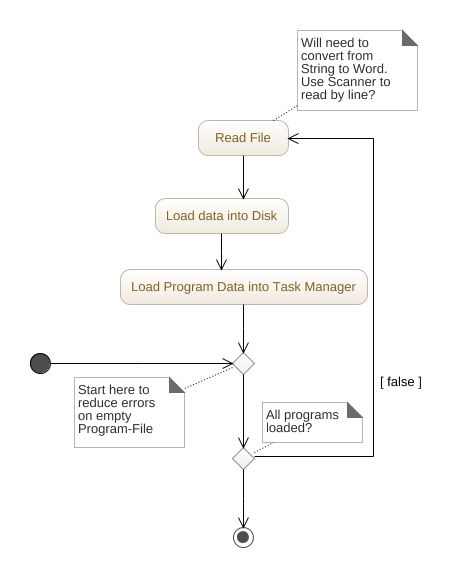


Figure 11 Activity diagram of the loader.

Other than that, the design of the loader includes very little in addition to the requirements specifications. The loader reads in the file, turns each hexadecimal value into a Word object, and stores it in the appropriate address in the disk. Using the addresses, and the cards contained within the program file, the loader creates a PCB in the task manager to store data about the loaded process.

### Scheduler

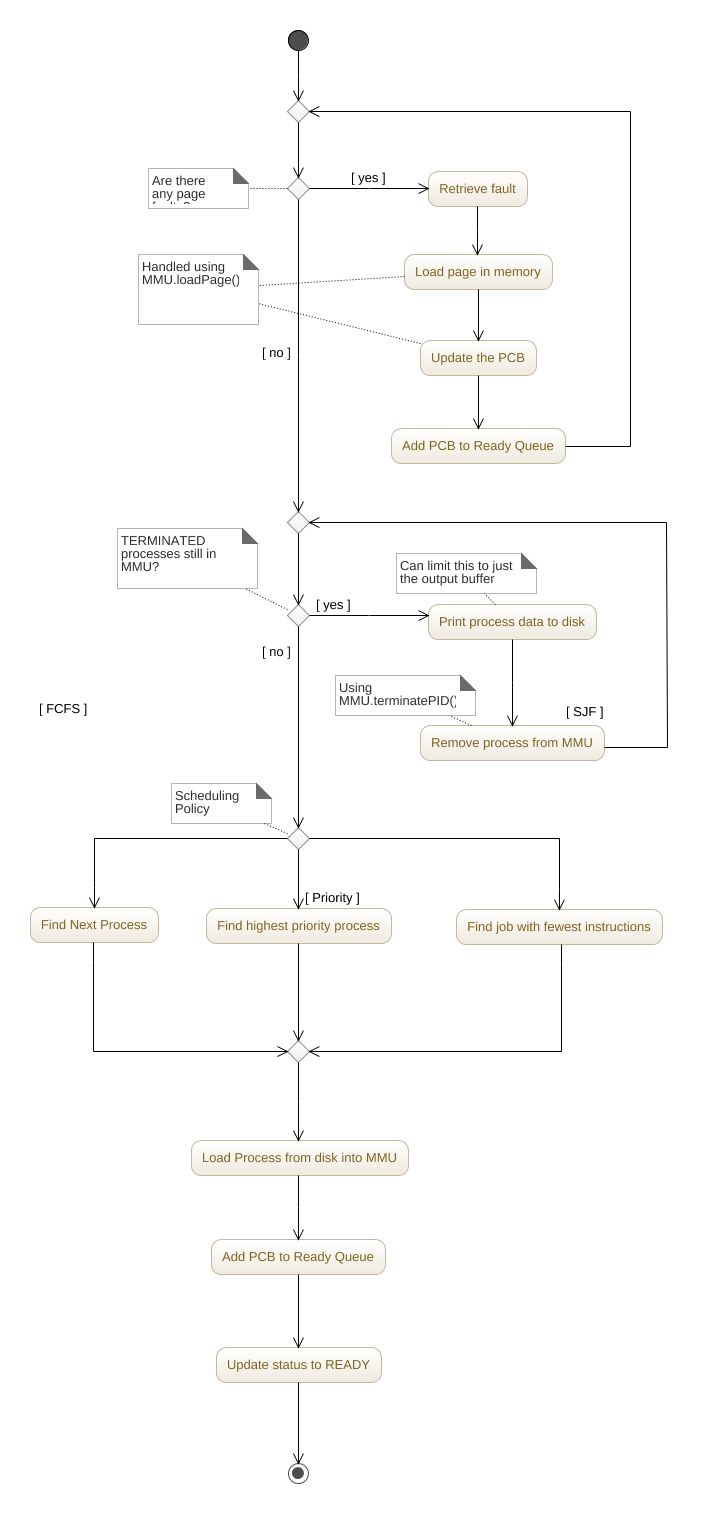


Figure 12 Activity diagram of the scheduler.

The scheduler was one of the first continuously-operating components to be designed. Its responsibilities are to load the appropriate process into RAM based on the chosen scheduling policy, to load that process’s PCB into the ready queue, and to write any terminated processes from RAM back to the disk.

The design of the scheduler closely coincided in time with the design of the CPU, and actually underwent several changes as the CPU design was refined. Most notably, the method of loading the program data was changed from a direct load to RAM to an indirect load using the MMU.

As the design of the MMU was further adjusted and improved, so too did the process by which the scheduler loaded the data into it. The original design was that for each word of data that the scheduler wished to write for the process, it would have to go to the MMU to register a physical address with the process’s logical address for that word and write the word through the MMU if the address could be mapped.

This design resulted in a number of issues in implementation, such as when a the MMU runs out of data part-way through the loading of a process, resulting in a lengthy process of un-mapping the data that had successfully been written.

The final design calls for the scheduler to map all of the addresses with the MMU that the process needs and begins loading the process data through the MMU once it has been notified that the process addresses have been successfully mapped.

With the addition of paging and page faults, the it was determined that it would be the scheduler that would handle the page faults. At the beginning of each of its iterations, the scheduler retrieves the page faults from the MMU and attempts to load each of them. If the page is loaded, the appropriate PCB is updated and added back to the ready queue. There is no method for handling a failure to load a page, and the scheduler will attempt to load it each iteration until the page successfully loads.

### Dispatcher

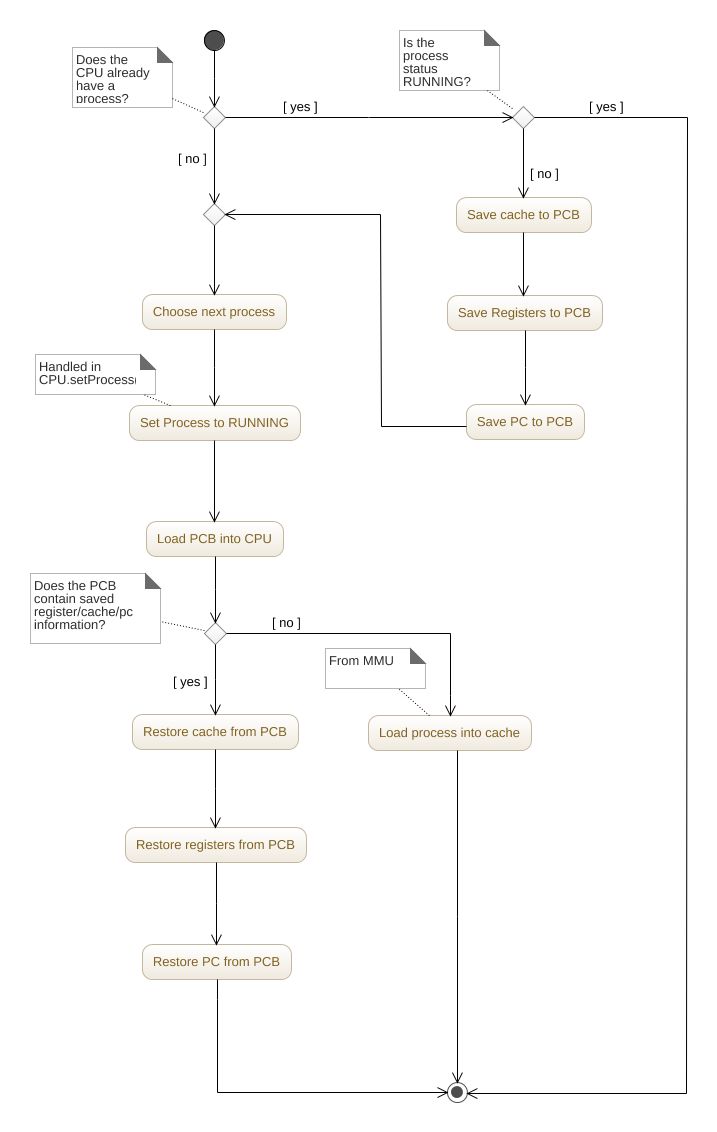


Figure 13 Activity diagram of the dispatcher.

For Phase 1 Part 1, the dispatcher started off very simple. Its sole job was to tell the CPU which process to execute. In Phase 1 Part 2, the dispatcher had to be changed to load the process into the cache, and to write the cache of terminated processes back to memory. Since the cache at the time was designed to hold whole processes, this added little to the complexity of the dispatcher. To accommodate the multiple CPUs, the dispatcher simply iterated through each of them, performing the same set of operations on each.

However, with the introduction of paging and I/O blocking, along with changes to the nature of the cache, the dispatcher had to be heavily modified. Process blocking was easy to handle for the dispatcher. When checking CPUs, instead of checking to see if the process was terminated, it would check to see if the process was in any other state than “RUNNING”. If the process was not running, it would be removed and replaced. Furthermore, if the process were waiting, a status reserved for both page and I/O blocking, the process’s program counter, registers, and temp buffer data would be saved to the process’s PCB. When loading a new process, the dispatcher retrieves these from the PCB to fill in the correct data for the CPU to resume (or even begin a new process) correctly.

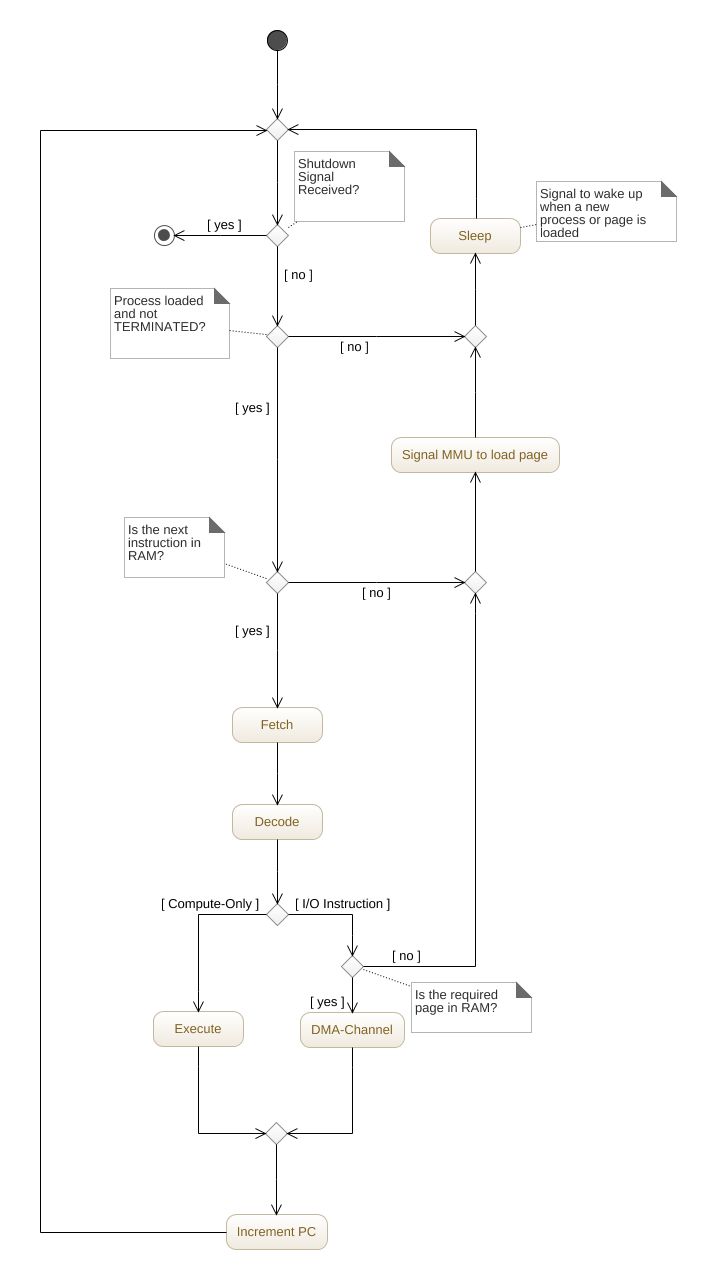


Figure 14 Activity diagram of the CPU.

## CPU

The design of the CPU has changed very little over the course of the project. It started out by being called as part of the driver loop in Phase 1 Part 1, when it could assume that it had a process already loaded and would begin execution until the process was completed. In Phase 1 Part 2, with the addition of multi-processing, the CPU was initiated without a process loaded. Instead, it just loops until there’s an available process, at which point it begins executing it, or until it receives a shutdown signal and terminates. In Phase 2 Part 2, considerations had to be made for page faults and I/O blocking, but each of those are were extremely simple additions for the CPU.

### Fetch

The fetch operation was designed to be a set of lines within the CPU’s execution loop instead of a separate component due to its simplicity. It was originally designed to read from the MMU but was later adapted to utilize the cache. Even with the further addition of the paged cache, nothing had to be changed. The cache itself interpreted the logical addresses and handled the loading of pages from the RAM when a requested instruction was not available. If there were any page faults, the CPU was able to handle them the same as it would with faults from any of its other components.

### Decode & Execute

The decoder is a mix of different components in this implementation. It was decided that the decoder would create executable objects from the instruction data that would carry out each instruction’s operations. The executable objects are provided with all of the data they need to correctly complete their operations and simply need to be called during the Execute cycle, unless they’re for an I/O instruction. I/O executable objects are passed to the DMA Channel for execution, along with any further data the DMA Channel needs to complete its operations.

### DMA Channel

The DMA Channel started out as a separate class with a single method that handled the I/O operations. Even through the earlier implementations of multi-processing, the DMA Channel was run in the calling CPU’s thread. However, once process swapping was implemented, it became necessary to separate the DMA Channel from any single CPU. If a process sends an I/O request from CPU A and gets re-assigned to CPU B after the I/O operation has been completed, then it would create issues if the DMA Channel it had requested the operation through were only accessible to CPU B. Therefore, the DMA Channel was given its own thread in which to operate.

Running as its own thread, the DMA Channel constantly checks to see if it has any pending jobs. If it has any, it will attempt to execute them, adding the PCB back to the ready queue if the job was accomplished successfully, or discarding the job if there is a page fault. If the operation is an output operation, the DMA Channel pulls the necessary data from the registers during the job notification process. If the operation was an input operation, the DMA Channel will hold onto the read-in data until the process is resumed and requests it.

## Multiprocessor Architecture

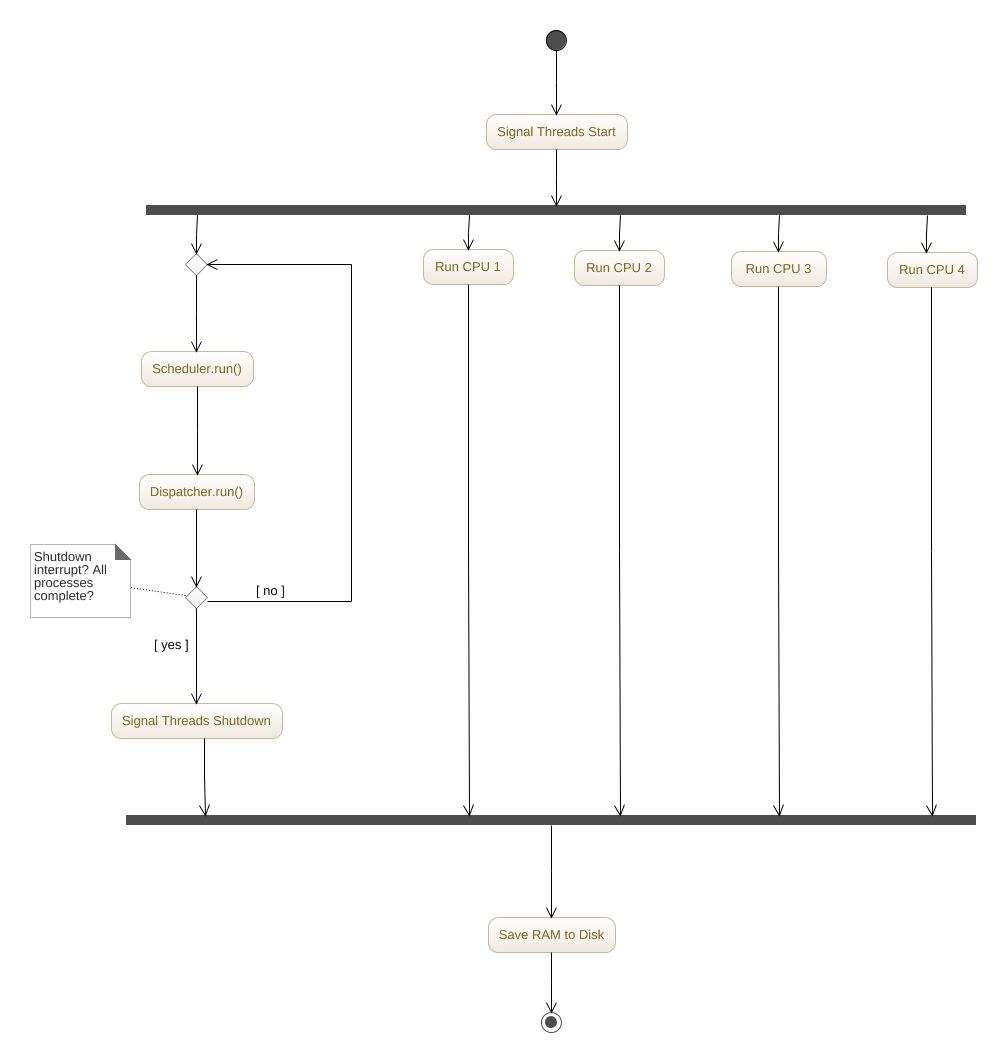


Figure 15 Activity Diagram of the Multiprocessor Driver.

### Multiprocessor Memory Management

### Multiprocessor Program Cache

## Shortest Job First

This was perhaps the easiest section to design, and actually allowed for some changes to the prior design to streamline the various components. In anticipation of the complex interactions with the ready queue that would come from process swapping, it was determined that the ready queue should be switched from a FIFO queue to a priority queue, which re-orders the entire queue when a new entry is added. This was because, when a process was added back to the queue after waiting for an I/O request or a page to be loaded, it needed to be added back in its correct position in the queue in regard to the scheduling method, and not just in the back. Since both the dispatcher and scheduler would need to use the scheduling algorithm, the algorithm was moved into the *enum* used to determine the scheduling policy via a comparator. Also, instead of having the dispatcher handle this directly, the comparator is added to the ready queue so the dispatcher only needs to add it, and the ready queue will handle the positioning internally.

As for the actual shortest job algorithm itself, it was determined that prioritizing based of the current remaining instructions, instead of the total instructions, would lead to the best result. Therefore, the algorithm subtracts the PC from the total instructions, and chooses the process with the lowest result. Since the other scheduling algorithms were already values within an *enum*, adding the SJF to it was simple.

## Paging and I/O Blocking

The addition of paging, I/O blocking, page faults, and the servicing of page faults resulted in small changes in the design that ultimately lead to sweeping changes in the implementation. While the concepts seemed quite simple, they included, unknown to us, a large number of race conditions and opportunities for deadlocks that plagued the system. Eventually, it was determined that a slower, more stable system was more desirable than a much faster system that crashed at a greater frequency. As a result, there are many instances where a thread will sit and do nothing rather than risk the opportunity of interfering with another thread’s data.

### Paging

The original design for the MMU already included a form of rudimentary paging that was partially adapted to the later paging design. In the original design, every single logical address was mapped 1-to-1 to a physical address, not necessarily in order. With the included requirement of pages and frames of 4 words each, this design had to be altered.

The MMU’s address mapping tables of the earlier design were morphed into the PCBs’ page tables in the later design. This design helped further protect each process’s data, since its PCB was required to access it (the MMU wholly controlled access to the RAM and required the PCB and the process’s logical address for any read or write operation). While this design sounds good in theory, there have been persistent and intermittent memory corruption issues whose cause has not been completely identified and might be an upshot flaw in this design change.

Paging was also introduced into the cache, which underwent more extensive design changes than even the MMU or PCB. The physical cache was also removed and enclosed in a similar manner to the RAM, with all access requiring the PCB and the process’s logical address. The cache was then trimmed down to include room for just the instructions and the temp buffer. This was determined to be necessary due to the swapping that was required for the page faulting and I/O blocking. And to help ensure that page faulting would occur during the process’s execution, the size of memory allocated for the process’s instructions were reduced to just 2 frames, 8 instructions. This helped minimize the wasted cache space that existed in the earlier designs.

Currently there are 2 ways that a page fault can be generated: during the Fetch cycle of the CPU and when the DMA Channel is servicing an I/O request. During a Fetch operation, the CPU attempts to read the cache for the requested instruction. If the cache does not have the correct page, it will automatically attempt to load it from RAM through the MMU. If the correct page is again not loaded into RAM, it will generate a page fault which the MMU will store for later servicing and the CPU will catch. When the DMA Channel is servicing an I/O request, the CPU has already put the process to sleep. So when the DMA Channel attempts to read an invalid page from RAM and receives a page fault, it simply discards the I/O job and moves onto the next one. The MMU automatically stores all page faults, which the Scheduler checks for, in its cycle and commands their servicing. Once any page fault has been serviced, the Scheduler will add the process’s PCB back into the ready queue, which automatically sorts it into the correct position.

### I/O Blocking

I/O in itself was a very simple system to design and implement. Once the CPU had notified the DMA Channel of an I/O job, the CPU would update the status of the PCB to ‘WAITING’ and would end its loop. Since each loop of the CPU checked to ensure that the PCB status was ‘RUNNING’, this meant that the CPU would stop executing the process and go to sleep. Once the dispatcher notices that a CPU’s process is no longer in the ‘RUNNING’ status the dispatcher pulls the old process, saves the data if necessary, sends a new process to the CPU, and wakes up the CPU to resume execution.

The DMA Channel is constantly looping, constantly checking for new I/O jobs. Once it has one, it detects that it has one, it begins to complete it. For an output request, the notification process initiated by the CPU pulls data from the instruction and registers and stores it in an output cache located within the DMA Channel (the term “cache” applies very loosely). Using this stored data, the DMA Channel attempts to write to RAM through the MMU. For an input request, the DMA Channel attempts to read the appropriate RAM address through the MMU into an input cache (this actually is a cache). If either an input or output operation generates a page fault, the job is immediately halted and discarded, with no further action taken by the DMA Channel. If there is a page fault, the Scheduler will take care of loading the page, and the process will re-attempt the I/O request once it has been resumed.

Once an I/O request has been completed, the DMA Channel will update the status of the PCB back to ‘READY’ and will add it back into the ready queue. If the I/O request was an input, the process will resume by executing the same instruction again, which will result in the data from the input cache being pulled into the appropriate register. If the I/O request was an output, the process will resume on the next instruction.

# Simulation Discussion

For some reason, the operating system requires Java 8 for correct execution. Even if you’re using Java 9 or 10 and tell them to run like Java 8, it won’t work correctly. At least that’s the case when executing it through Intellij IDEA. It has something to do with the annotations used, they were likely removed between Java 8 and 9.

Furthermore, the program needs the 3 included libraries, “poi-3.17” (all contained JARs), “commons-collections4-4.1.jar”, and “xmlbeans-2.5.0.jar”. The libraries are needed for the program to print the runtime data into an Excel spreadsheet for easy reading. In order for the Excel printer to appropriately output the data, it must have the “template.xlsx” file in the project’s resources directory. If the file is no there, there will be an error, and some of the runtime data will be printed to the terminal. Furthermore, if the “data.xlsx” file in the output directory has had its structure modified, or is open, the program will experience a similar error. If there is an error while printing, the best thing to do is to delete the old data file. If that does not resolve the issue, ensure that the template file is in the appropriate location (if running through an IDE like Intellij, it may be necessary to mark the directory as the ‘Resources’ directory).

There is a GUI that was developed for the system, however it was never fully integrated into Phase 2 due to time constraints. Phase 2 executes via the terminal, though it doesn’t require any inputs, and only gives status update outputs. When executing, the program will automatically execute 6 operating system simulations, one right after the other. It iterates through the 3 scheduling methods and will execute them in both a 1-CPU system and a 4-CPU system. If everything works correctly, all 6 will be located in different tabs within the same data output file. However, there has been mixed success in getting all 6 simulations to complete correctly. For the most part, if there is an issue with one, it will be abandoned, and the program will begin the next simulation. However, this has not been tested, and may produce incorrect data. If there are persistent issues with this, the main is located in the Main class, and could be edited to only execute one system at a time, if necessary.

# Data Analysis

## FCFS and Priority Comparison

The waiting, run time, and completion time data collected for both FCFS and Priority are shown in Figures 16 and 17. On a single CPU, the average waiting time for FCFS was 505.23 ms, the average run time was 10.7ms, and the average completion time was 515.92ms.  On a single CPU, the average waiting time for Priority was 113.9 ms, the average run time was 4.6ms, and the average completion time was 118.5ms. On a single CPU the priority scheduling algorithm had a waiting time average that was a fifth of FCFS, a run time average that’s half of FCFS, and a completion time that was a fifth of FCFS. This shows that priority scheduling is better than FCFS in terms of throughput. This is further supported by the data in Table 2, where priority wait times and completion times were a half of wait and completion times in FCFS. In FCFS, processes with a large number of executions may execute before shorter processes, which increases the average wait time and completion time. The priority scheduling algorithm most likely assigned priority based on a factor that estimates how long a process will take to execute.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Wait Time (ms) | Run Time (ms) | Completion Time (ms) |
| FCFS | 505.23 | 10.7 | 515.93 |
| Priority | 113.9 | 4.6 | 118.5 |
| Average | 309.56 | 7.65 | 317.21 |

Table 1 Comparison of Metric Averages for FCFS and Priority on 1 Thread with Cache.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Wait Time (ms) | Run Time (ms) | Completion Time (ms) |
| FCFS | 73.16 | 15.8 | 88.96 |
| Priority | 35.7 | 5.33 | 41.03 |
| Average | 54.43 | 10.56 | 64.99 |

Table 2 Comparison of Metric Averages for FCFS and Priority on Four Threads with Cache.

## Introduction of Cache and Comparison

Table 1 and Table 2 illustrated how the priority scheduling and FCFS scheduling algorithms compare, however Table 1 can also be compared with Table 3 below to analyze the addition of Cache and how it affected the wait times, run times, and completion times of the 30 jobs. With the introduction of Cache, the average wait times decreased by roughly 80ms, the average run time decreased by 12ms, and the average completion time decreased by 90ms. The main purpose of the cache is to decrease run time and completion time. When a process needs a particular piece of information, first it checks the cache and if found, it uses that piece of information. If it does not find the information in the cache, it then checks the information from the source and makes a copy of it in the cache for later use, assuming that it will be needed again sometime in the near future. Upon analyzing the data, a conclusion can be reached that this is exactly the case. For further data points, please see Appendix C-F for additional data points for each process and cache comparisons.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Wait Time (ms) | Run Time (ms) | Completion Time (ms) |
| FCFS | 424.03 | 19.73 | 443.76 |
| Priority | 348.56 | 18.5 | 367.06 |
| Average | 386.29 | 19.11 | 405.41 |

Table 3 Comparison of Metric Averages FCFS, Priority One CPU Without Cache.

## Percentage of RAM and Cache Used

The FCFS and the priority scheduling algorithms both had the same data for percentage of RAM, percentage of Cache, and Number of I/O. This is due to the fact that the processes only differ in the order that they are run and there is no preemption. The average percentage of RAM used was: 67.56%, and the average percentage of cache used was 6.59%.

Figure 16 Percent Cache and RAM Used on One CPU.

## Comparison of Performance for 1-CPU vs N-CPU Runs

The two tables in Table 4 and Table 5 hold the averages in wait times, run times, and completion times for 1-CPU and 4-CPUs for FCFS and Priority scheduling. The times of the two scheduling algorithms were averaged and used for this analysis, as it shows a good basis point for how long 1-CPU or 4-CPUs will take to go through the 30 jobs. For 1-CPU, the average wait time was 309.56ms, the average run time was 7.65ms, and the average completion time was 317.21ms. For 4-CPUs, the average wait time was 54.43ms, the average run time was 10.56ms, and the average completion time was 64.99ms. The average wait times on 1-CPU were roughly 6 times the average wait time for 4-CPUS, the average run times were roughly the same, and the average completion times on 1-CPU were 5 times as long as those on 4-CPU. This proves that using more CPUs allow for processes to execute more quickly. Not only can the processes run concurrently on the different CPUs, the 4-CPUs can each grab a process from the ready queue, shortening the average waiting times and completion times overall. Furthermore, for FCFS, a process that takes a long time to run will only hold up one CPU while the other CPUs can continue grabbing and executing jobs.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Wait Time (ms) | Run Time (ms) | Completion Time (ms) |
| FCFS | 505.23 | 10.7 | 515.93 |
| Priority | 113.9 | 4.6 | 118.5 |
| Average | 309.56 | 7.65 | 317.21 |

Table 4 Comparison of Metric Averages for FCFS, Priority One CPU with Cache.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Wait Time (ms) | Run Time (ms) | Completion Time (ms) |
| FCFS | 73.16 | 15.8 | 88.96 |
| Priority | 35.7 | 5.33 | 41.03 |
| Average | 54.43 | 10.56 | 64.99 |

Table 5 Comparison of Metric Averages for FCFS, Priority on Four CPUs with Cache.

Table 6 and Table 7 illustrate the number of processes each CPU executed for FCFS and Priority. Each CPU grabbed roughly 20-30% of the jobs, splitting the workload among each other. This corresponds with the large decrease in average wait times and completion times illustrated in Table 4 and Table 5. For further data, please check Appendixes G and H that illustrate the individual run, completion, and wait times for 4-CPUs as well as records what CPU each process was grabbed by.

|  |  |  |
| --- | --- | --- |
| **CPU ID:** | **Number Processes** | **Percent Processes** |
| 0 | 9 | 30% |
| 1 | 7 | 23% |
| 2 | 7 | 23% |
| 3 | 7 | 23% |

Table 6 Process Distribution for CPUs using FCFS.

|  |  |  |
| --- | --- | --- |
| **CPU ID:** | **Number Processes** | **Percent Processes** |
| 0 | 9 | 30% |
| 1 | 10 | 33% |
| 2 | 6 | 20% |
| 3 | 5 | 16.77% |

Table 7 Process Distribution for CPUs using Priority.

## FCFS, Priority, and SJF Data Comparison

Phase 2 featured the introduction of the shortest job first (SJF) scheduling algorithm. The SJF scheduling algorithm assigns a priority to processes based on the number of instructions the process has. The processes with the least number of instructions are scheduled to execute first. This should overall decrease the wait and completion times of each process, and not have an effect on the number of I/O operations or the number of executions.

In comparing FCFS, Priority, and SJF scheduling algorithms, FCFS was found to have the longest completion and wait times, and Priority and SJF were found to have similar completion and wait times. The data for comparing the three scheduling algorithms on 4-CPUs is found below in Table 9. On four CPUs, the average wait time for FCFS was 73.16 ms, the average run time was 15.8ms, and the average completion time was 88.96ms.  On four CPUs, the average waiting time for Priority was 35.7 ms, the average run time was 5.33ms, and the average completion time was 41.03ms. On four CPUs, the average waiting time for SJF was 38.96 ms, the average run time was 10.96ms, and the average completion time was 49.93ms. FCFS had a wait time that was twice as long as both SJF and Priority. This is most likely due to FCFS scheduling algorithm scheduling on a first come first serve basis, so processes that take a long time may end up being taken off the ready queue and given a CPU before shorter processes. This increases the overall waiting times and completion times. SJF and Priority most likely have similar wait, run, and completion times because SJF is a version of Priority scheduling, where the number of instructions is used as the priority.

The data for the comparisons on all three scheduling algorithms on 1-CPU can be found below in Table 8 and show the same trends described above.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Wait Time (ms) | Run Time (ms) | Completion Time (ms) |
| SJF | 75.86 | 3.23 | 79.1 |
| FCFS | 505.23 | 10.7 | 515.93 |
| Priority | 113.9 | 4.6 | 118.5 |
| Average | 231.66 | 6.17 | 237.84 |

Table 8 Comparison of Metric Averages for FCFS, Priority, and SJF on One CPU.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Wait Time (ms) | Run Time (ms) | Completion Time (ms) |
| SJF | 38.96 | 10.96 | 49.93 |
| FCFS | 73.16 | 15.8 | 88.96 |
| Priority | 35.7 | 5.33 | 41.03 |
| Average | 49.27 | 10.69 | 59.97 |

Table 9 Comparison of Metric Averages for FCFS, Priority, and SJF on Four CPUs.

## Understanding the Effects of Paging

The main purpose of Paging is speed up the context switch time. Paging allows the physical memory to be broken down into frames and the logical memory to be broken down into pages. The separation of the physical and logical memory allows a process with a larger address space then the physical memory, to still run if stored in a page and from there called into the available frame. However sometimes a page fault can occur, and the system has to stop and service this page fault. This has the potential to take a large amount of overhead and defeat the benefits of minimizing context switch times. This is evident in the comparison of Table 9 above and Table 10 below. With the introduction of paging, the average wait time increased by a factor of around five times, the run time almost doubled, and the completion time increased by a factor of six. This is most likely the result of a large number of page faults and the time it takes the OS to service these faults. This is illustrated in Table 11, where its shown that the average number of page faults per process, regardless of scheduling algorithm, is 7.6, and the average time spent servicing each page fault is 19436 (ns). Appendix N further showcases the number of page faults for each process and the amount of time it took to service the page fault for each process for each scheduling algorithm. From this data, it’s concluded that the overhead costs of paging had a negative effect on total completion and waiting times.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Wait Time (ms) | Run Time (ms) | Completion Time (ms) |
| SJF | 156.36 | 5.36 | 161.73 |
| FCFS | 202.23 | 26.73 | 228.96 |
| Priority | 478.63 | 27.53 | 506.16 |
| Average | 279.07 | 19.87 | 298.95 |

Table 10 Comparison of Metric Averages for FCFS, Priority, and SJF on Four CPUs With Paging.

|  |  |  |
| --- | --- | --- |
|  | Number of Page Faults | Average Page Fault Servicing Times (ns) |
| FCFS | 7.533 | 22054.5 |
| Priority | 7.43 | 13873.88 |
| SJF | 7.86 | 22379.88 |
| Average | 7.607 | 19436.08 |

Table 11 Comparison of Page Faults and Average Page Fault Servicing Times for FCFS, Priority, and SJF on Four CPUs With Paging.

# Conclusions

## Phase 1 Conclusions

In Phase 1, the main concepts that were supported by the data surround the concepts of scheduling algorithms, the use of cache memory, and the perks of using a multiprocessor. In Phase 1 we compared priority scheduling and FCFS scheduling. On both 1-CPU and 4-CPUs priority scheduling resulted in shorter wait and completion times than FCFS scheduling. It was concluded that this is because FCFS was most likely to have longer processes execute before shorter processes, which increases overall wait and completion times for each process. Priority appeared to circumvent this issue and had less longer processes executed before shorter processes.

The introduction of cache memory did affect the overall performance of the Operating System. The introduction of cache resulted in a large decrease in completion time, run time, and waiting times. This supports the concept that cache acts as a form of fast memory and can increase the throughput of the system.

In terms of the perks for using a multiprocessor, having N-CPUs resulted in a better throughput than having a singular CPU. As illustrated in Table 4 and Table 5, with an additional 3 CPUs, the wait time for FCFS scheduling is cut from 505.23ms down to 73.16ms, and the completion time is cut from 515.93ms down to 118.5ms. For Priority scheduling, having 4-CPUs cut wait time and completion time in half. It was concluded that as the number of CPUs increase, the number of processes that can run concurrently increases. Furthermore, with 4-CPUs, if one processor grabbed a long process, the other 3 CPUs would be unaffected and can continue to grab processes from the shared ready queue to execute. Table 6 and Table 7 indicate that with both FCFS and Priority scheduling, the CPUs each managed to grab 20-30% of the 30 jobs. This indicates that the general workload was split near equally between the 4 CPUs.

In conclusion, Phase 1 findings support the facts that priority scheduling is more efficient in terms of lessening wait and completion times than FCFS, cache decreases overall times, and that having N-CPUs also decreased overall wait and completion times.

## Phase 2 Conclusions

Phase 2 built further on top of the modules and algorithms developed in Phase 1. Phase 2 featured an additional scheduling algorithm: shortest job first (SJF). SJF used the number of instructions as an estimate on how long the job would take to run. Using the instructions was only an estimate because it did not include the possible number of times the system looped through I/O servicing. In comparing the three scheduling algorithms, it was found that FCFS resulted in the longest waiting and completion times regardless of how many CPUs were used. Furthermore, the data shows that SJF had similar waiting times and completion times as Priority, most likely because SJF is an alteration of Priority scheduling, where the priority is an estimation on how long processes will take to run. The last addition to the project was the implementation of paging and the effects it had on overall throughput of the system.

Paging proved to have a negative impact on the completion and waiting times for all three scheduling algorithms. Each process tended to have a large number of page faults, and it took the system a sizeable portion of time to service all of these faults. This ended up causing an increase of wait times by a factor of five, an increase in run times by a factor of two, and an increase of completion time by a factor of six.

# Supporting Information

## Table of Figures

[Figure 1 Overview of the Operating System components (provided by Dr. Bobbie). 7](#_Toc511881803)

[Figure 2 Overview of the Driver's flow (provided by Dr. Bobbie). 9](#_Toc511881804)

[Figure 3 Outline of the loader’s logic (provided by Dr. Bobbie). 10](#_Toc511881805)

[Figure 4 Heart of the DMA-channel module/thread (provided by Dr. Bobbie). 15](#_Toc511881806)

[Figure 5 Overview of the CPU’s execution (provided by Dr. Bobbie). 15](#_Toc511881807)

[Figure 6 The suggested structure of the PCB (provided by Dr. Bobbie). 16](#_Toc511881808)

[Figure 7 Diagram of a Page Table (provided by Dr. Bobbie). 18](file:///E:\Users\Mark%20Zeagler\Desktop\CS3502%20Project%20Phase%202%20Report.docx#_Toc511881809)

[Figure 8 Diagram of process states (provided by Dr. Bobbie). 19](file:///E:\Users\Mark%20Zeagler\Desktop\CS3502%20Project%20Phase%202%20Report.docx#_Toc511881810)

[Figure 9 System overview diagram. 21](file:///E:\Users\Mark%20Zeagler\Desktop\CS3502%20Project%20Phase%202%20Report.docx#_Toc511881811)

[Figure 10 UML class diagram of the memory system. 21](file:///E:\Users\Mark%20Zeagler\Desktop\CS3502%20Project%20Phase%202%20Report.docx#_Toc511881812)

[Figure 11 Activity diagram of the loader. 22](file:///E:\Users\Mark%20Zeagler\Desktop\CS3502%20Project%20Phase%202%20Report.docx#_Toc511881813)

[Figure 12 Activity diagram of the scheduler. 23](file:///E:\Users\Mark%20Zeagler\Desktop\CS3502%20Project%20Phase%202%20Report.docx#_Toc511881814)

[Figure 13 Activity diagram of the dispatcher. 25](file:///E:\Users\Mark%20Zeagler\Desktop\CS3502%20Project%20Phase%202%20Report.docx#_Toc511881815)

[Figure 14 Activity diagram of the CPU. 26](file:///E:\Users\Mark%20Zeagler\Desktop\CS3502%20Project%20Phase%202%20Report.docx#_Toc511881816)

[Figure 15 Activity Diagram of the Multiprocessor Driver. 28](file:///E:\Users\Mark%20Zeagler\Desktop\CS3502%20Project%20Phase%202%20Report.docx#_Toc511881817)

[Figure 16 Percent Cache and RAM Used on One CPU. 32](#_Toc511881818)

## Table of Tables

[Table 1 Comparison of Metric Averages for FCFS and Priority on 1 Thread with Cache. 31](#_Toc511881681)

[Table 2 Comparison of Metric Averages for FCFS and Priority on Four Threads with Cache. 31](#_Toc511881682)

[Table 3 Comparison of Metric Averages FCFS, Priority One CPU Without Cache. 31](#_Toc511881683)

[Table 4 Comparison of Metric Averages for FCFS, Priority One CPU with Cache. 32](#_Toc511881684)

[Table 5 Comparison of Metric Averages for FCFS, Priority on Four CPUs with Cache. 32](#_Toc511881685)

[Table 6 Process Distribution for CPUs using FCFS. 33](#_Toc511881686)

[Table 7 Process Distribution for CPUs using Priority. 33](#_Toc511881687)

[Table 8 Comparison of Metric Averages for FCFS, Priority, and SJF on One CPU. 33](#_Toc511881688)

[Table 9 Comparison of Metric Averages for FCFS, Priority, and SJF on Four CPUs. 34](#_Toc511881689)

[Table 10 Comparison of Metric Averages for FCFS, Priority, and SJF on Four CPUs With Paging. 34](#_Toc511881690)

[Table 11 Comparison of Page Faults and Average Page Fault Servicing Times for FCFS, Priority, and SJF on Four CPUs With Paging. 34](#_Toc511881691)

## Instruction Format

All instructions are 32 bits long. There are four types of instruction format.

* Arithmetic instruction format
* Conditional Branch and Immediate format
* Unconditional Jump format
* Input and Output instruction format

Arithmetic instruction format

2 bits 6 bits 4 bits 4 bits 4 bits 12 bits

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 00 | OPCODE | S-reg | S-reg | D-reg 000 |

The first two bits are always 00, indicating that the instruction is an Arithmetic or Register transfer type of instruction. S-reg is the source register. D-reg is the destination register. The last 12 bits are always 0, as they are not used.

Conditional Branch and Immediate format

2 bits 6 bits 4 bits 4 bits 16 bits

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 01 | OPCODE | B-reg | D-reg | Address |

The first two bits are always 01, indicating that the instruction is a Conditional Branch and Immediate type of instruction. B-reg is the base register. D-reg is the destination register. The last 16 bits may be an address or an immediate data.

* When the last 16 bits contain data, the D-reg is always 0000.
* The Address may at times be treated as data, which is direct addressing.
* An indirect Address is calculated as:

Effective Address = Content (B-reg) + Address

* Conditional Branch checks for B and D reg to cause a branch, to a specified Address, or not

Unconditional Jump format

2 bits 6 bits 24 bits

|  |  |  |
| --- | --- | --- |
| 10 | OPCODE | Address |

The first two bits are always 10, indicating that the instruction is an Unconditional Jump type of instruction, with a jump to the specified Address.

Input and Output instruction format

2 bits 6 bits 4 bits 4 bit 16 bits

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | OPCODE | Reg 1 | Reg 2 | Address |

The first two bits are always 11, indicating that the instruction is an Input and Output type of instruction.

* The instruction may read the content of Address/Reg 2 into Reg 1.
* The instruction may write the content of Reg 1 into a specified Address/Reg 2.

Registers

There are 16 registers; each of 32-bit long.

* Reg-0 (0000) being the Accumulator.
* Reg-1(0001) being the Zero register, which contains the value 0.
* All other registers are general purpose register.

Buffers

* Input buffer – containing data read by the program
* Output buffer – containing data produced by the program
* Temp buffer – area in memory to store/retrieve the data temporarily.

## Appendix B: Instruction Set

OPCD INSTRCT TYPE COMMENT

|  |  |  |  |
| --- | --- | --- | --- |
| 00 | RD | I/O | Reads content of I/P buffer into a accumulator |
| 01 | WR | I/O | Writes the content of accumulator into O/P buffer |
| 02 | ST | I | Stores content of a reg. into an address |
| 03 | LW | I | Loads the content of an address into a reg. |
| 04 | MOV | R | Transfers the content of one register into another |
| 05 | ADD | R | Adds content of two S-regs into D-reg |
| 06 | SUB | R | Subtracts content of two S-regs into D-reg |
| 07 | MUL | R | Multiplies content of two S-regs into D-reg |
| 08 | DIV | R | Divides content of two S-regs into D-reg |
| 09 | AND | R | Logical AND of two S-regs into D-reg |
| 0A | OR | R | Logical OR of two S-regs into D-reg |
| 0B | MOVI | I | Transfers address/data directly into a register |
| 0C | ADDI | I | Adds a data value directly to the content of a register |
| 0D | MULI | I | Multiplies a data value directly with the content of a register |
| 0E | DIVI | I | Divides a data directly to the content of a register |
| 0F | LDI | I | Loads a data/address directly to the content of a register |
| 10 | SLT | R | Sets the D-reg to 1 if first S-reg is less than the B-reg; 0 otherwise |
| 11 | SLTI | I | Sets the D-reg to 1 if first S-reg is less than a data; 0 otherwise |
| 12 | HLT | J | Logical end of program |
| 13 | NOP | - | Does nothing and moves to next instruction |
| 14 | JMP | J | Jumps to a specified location |
| 15 | BEQ | I | Branches to an address when content of B-reg = D-reg |
| 16 | BNE | I | Branches to an address when content of B-reg <> D-reg |
| 17 | BEZ | I | Branches to an address when content of B-reg = 0 |
| 18 | BNZ | I | Branches to an address when content of B-reg <> 0 |
| 19 | BGZ | I | Branches to an address when content of B-reg > 0 |
| 1A | BLZ | I | Branches to an address when content of B-reg < 0 |

## Appendix C: Process Data Table (FCFS, 1 Thread, No Cache)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of RAM** |
| 1 | 37 | 124 | 161 | 140 | 12 | 0 | 6.54 |
| 2 | 159 | 33 | 192 | 151 | 12 | 0 | 7.03 |
| 3 | 191 | 40 | 231 | 141 | 12 | 0 | 6.64 |
| 4 | 232 | 3 | 235 | 91 | 12 | 0 | 6.15 |
| 5 | 233 | 8 | 241 | 151 | 12 | 0 | 7.03 |
| 6 | 241 | 35 | 276 | 141 | 12 | 0 | 6.64 |
| 7 | 277 | 12 | 289 | 91 | 12 | 0 | 6.15 |
| 8 | 288 | 6 | 294 | 91 | 12 | 0 | 6.15 |
| 9 | 294 | 9 | 303 | 141 | 12 | 0 | 6.64 |
| 10 | 303 | 19 | 322 | 151 | 12 | 0 | 7.03 |
| 11 | 321 | 9 | 330 | 140 | 12 | 0 | 6.54 |
| 12 | 330 | 14 | 344 | 152 | 12 | 0 | 7.03 |
| 13 | 344 | 20 | 364 | 141 | 12 | 0 | 6.64 |
| 14 | 365 | 3 | 368 | 75 | 10 | 0 | 6.15 |
| 15 | 367 | 12 | 379 | 128 | 11 | 0 | 6.64 |
| 16 | 379 | 1 | 380 | 35 | 5 | 0 | 6.15 |
| 17 | 381 | 8 | 389 | 154 | 12 | 0 | 7.03 |
| 18 | 389 | 20 | 409 | 141 | 12 | 0 | 6.64 |
| 19 | 408 | 8 | 416 | 121 | 10 | 0 | 7.03 |
| 20 | 417 | 50 | 467 | 141 | 12 | 0 | 6.64 |
| 21 | 467 | 83 | 550 | 99 | 13 | 0 | 6.15 |
| 22 | 550 | 9 | 559 | 151 | 12 | 0 | 7.03 |
| 23 | 559 | 5 | 564 | 127 | 11 | 0 | 6.54 |
| 24 | 700 | 10 | 710 | 141 | 12 | 0 | 6.64 |
| 25 | 711 | 38 | 749 | 75 | 7 | 0 | 6.54 |
| 26 | 750 | 1 | 751 | 59 | 8 | 0 | 6.15 |
| 27 | 751 | 3 | 754 | 140 | 12 | 0 | 6.54 |
| 28 | 754 | 5 | 759 | 152 | 12 | 0 | 7.03 |
| 29 | 760 | 3 | 763 | 141 | 12 | 0 | 6.64 |
| 30 | 763 | 1 | 764 | 83 | 11 | 0 | 6.15 |
| Average | 424.03 | 19.73 | 443.76 | 122.83 | 11.26 |  | 6.59 |

## Appendix D: Process Data Table (Priority, 1 Thread, No Cache)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of RAM** |
| 1 | 757 | 18 | 775 | 140 | 12 | 0 | 6.54 |
| 2 | 427 | 3 | 430 | 151 | 12 | 0 | 7.03 |
| 3 | 409 | 2 | 411 | 141 | 12 | 0 | 6.64 |
| 4 | 411 | 3 | 414 | 91 | 12 | 0 | 6.15 |
| 5 | 490 | 8 | 498 | 151 | 12 | 0 | 7.03 |
| 6 | 401 | 3 | 404 | 141 | 12 | 0 | 6.64 |
| 7 | 414 | 2 | 416 | 91 | 12 | 0 | 6.15 |
| 8 | 185 | 21 | 206 | 91 | 12 | 0 | 6.15 |
| 9 | 774 | 10 | 784 | 141 | 12 | 0 | 6.64 |
| 10 | 245 | 9 | 254 | 151 | 12 | 0 | 7.03 |
| 11 | 497 | 245 | 742 | 140 | 12 | 0 | 6.54 |
| 12 | 85 | 32 | 117 | 152 | 12 | 0 | 7.03 |
| 13 | 31 | 31 | 62 | 141 | 12 | 0 | 6.64 |
| 14 | 609 | 1 | 610 | 75 | 10 | 0 | 6.15 |
| 15 | 242 | 6 | 248 | 128 | 11 | 0 | 6.64 |
| 16 | 211 | 8 | 219 | 35 | 5 | 0 | 6.15 |
| 17 | 254 | 11 | 265 | 154 | 12 | 0 | 7.03 |
| 18 | 61 | 6 | 67 | 141 | 12 | 0 | 6.64 |
| 19 | 623 | 3 | 626 | 121 | 10 | 0 | 7.03 |
| 20 | 609 | 11 | 620 | 141 | 12 | 0 | 6.64 |
| 21 | 264 | 1 | 265 | 99 | 13 | 0 | 6.15 |
| 22 | 77 | 5 | 82 | 151 | 12 | 0 | 7.03 |
| 23 | 246 | 2 | 248 | 127 | 11 | 0 | 6.54 |
| 24 | 221 | 3 | 224 | 141 | 12 | 0 | 6.64 |
| 25 | 110 | 1 | 111 | 75 | 7 | 0 | 6.54 |
| 26 | 304 | 1 | 305 | 59 | 8 | 0 | 6.15 |
| 27 | 111 | 92 | 203 | 140 | 12 | 0 | 6.54 |
| 28 | 559 | 12 | 571 | 152 | 12 | 0 | 7.03 |
| 29 | 619 | 3 | 622 | 141 | 12 | 0 | 6.64 |
| 30 | 211 | 2 | 213 | 83 | 11 | 0 | 6.15 |
| Average | 348.56 | 18.5 | 367.06 | 122.83 | 11.26 |  | 6.59 |

## Appendix E: Process Data Table (FCFS, 1 Thread, Cache)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of Cache** | **% of RAM** |
| 1 | 270 | 55 | 325 | 140 | 12 | 0 | 67 | 6.54 |
| 2 | 348 | 9 | 357 | 151 | 12 | 0 | 72 | 7.03 |
| 3 | 357 | 8 | 365 | 141 | 12 | 0 | 68 | 6.64 |
| 4 | 366 | 4 | 370 | 91 | 12 | 0 | 63 | 6.15 |
| 5 | 370 | 9 | 379 | 151 | 12 | 0 | 72 | 7.03 |
| 6 | 379 | 49 | 428 | 141 | 12 | 0 | 68 | 6.64 |
| 7 | 429 | 4 | 433 | 91 | 12 | 0 | 63 | 6.15 |
| 8 | 434 | 4 | 438 | 91 | 12 | 0 | 63 | 6.15 |
| 9 | 439 | 6 | 445 | 141 | 12 | 0 | 68 | 6.64 |
| 10 | 444 | 6 | 450 | 151 | 12 | 0 | 72 | 7.03 |
| 11 | 450 | 5 | 455 | 140 | 12 | 0 | 67 | 6.54 |
| 12 | 453 | 33 | 486 | 152 | 12 | 0 | 72 | 7.03 |
| 13 | 516 | 7 | 523 | 141 | 12 | 0 | 68 | 6.64 |
| 14 | 526 | 2 | 528 | 75 | 10 | 0 | 63 | 6.15 |
| 15 | 528 | 4 | 532 | 128 | 11 | 0 | 68 | 6.64 |
| 16 | 533 | 2 | 535 | 35 | 5 | 0 | 63 | 6.15 |
| 17 | 535 | 8 | 543 | 154 | 12 | 0 | 72 | 7.03 |
| 18 | 540 | 4 | 544 | 141 | 12 | 0 | 68 | 6.64 |
| 19 | 545 | 5 | 550 | 121 | 10 | 0 | 72 | 7.03 |
| 20 | 551 | 38 | 589 | 141 | 12 | 0 | 68 | 6.64 |
| 21 | 591 | 3 | 594 | 99 | 13 | 0 | 63 | 6.15 |
| 22 | 595 | 6 | 601 | 151 | 12 | 0 | 72 | 7.03 |
| 23 | 601 | 4 | 605 | 127 | 11 | 0 | 67 | 6.54 |
| 24 | 604 | 4 | 608 | 141 | 12 | 0 | 68 | 6.64 |
| 25 | 609 | 2 | 611 | 75 | 7 | 0 | 67 | 6.54 |
| 26 | 609 | 9 | 618 | 59 | 8 | 0 | 63 | 6.15 |
| 27 | 618 | 3 | 621 | 140 | 12 | 0 | 67 | 6.54 |
| 28 | 622 | 24 | 646 | 152 | 12 | 0 | 72 | 7.03 |
| 29 | 646 | 3 | 649 | 141 | 12 | 0 | 68 | 6.64 |
| 30 | 649 | 1 | 650 | 83 | 11 | 0 | 63 | 6.15 |
| Average | 505.23 | 10.7 | 515.93 | 122.83 | 11.26 |  | 67.67 | 6.59 |

## Appendix F: Process Data Table (Priority, 1 Thread, Cache)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of Cache** | **% of RAM** |
| 1 | 146 | 2 | 148 | 140 | 12 | 0 | 67 | 6.54 |
| 2 | 129 | 2 | 131 | 151 | 12 | 0 | 72 | 7.03 |
| 3 | 118 | 2 | 120 | 141 | 12 | 0 | 68 | 6.64 |
| 4 | 120 | 1 | 121 | 91 | 12 | 0 | 63 | 6.15 |
| 5 | 137 | 2 | 139 | 151 | 12 | 0 | 72 | 7.03 |
| 6 | 112 | 2 | 114 | 141 | 12 | 0 | 68 | 6.64 |
| 7 | 122 | 1 | 123 | 91 | 12 | 0 | 63 | 6.15 |
| 8 | 4 | 29 | 33 | 91 | 12 | 0 | 63 | 6.15 |
| 9 | 147 | 3 | 150 | 141 | 12 | 0 | 68 | 6.64 |
| 10 | 46 | 49 | 95 | 151 | 12 | 0 | 72 | 7.03 |
| 11 | 139 | 2 | 141 | 140 | 12 | 0 | 67 | 6.54 |
| 12 | 100 | 2 | 102 | 152 | 12 | 0 | 72 | 7.03 |
| 13 | 33 | 2 | 35 | 141 | 12 | 0 | 68 | 6.64 |
| 14 | 151 | 2 | 153 | 75 | 10 | 0 | 63 | 6.15 |
| 15 | 122 | 2 | 124 | 128 | 11 | 0 | 68 | 6.64 |
| 16 | 108 | 0 | 108 | 35 | 5 | 0 | 63 | 6.15 |
| 17 | 130 | 3 | 133 | 154 | 12 | 0 | 72 | 7.03 |
| 18 | 35 | 10 | 45 | 141 | 12 | 0 | 68 | 6.64 |
| 19 | 157 | 1 | 158 | 121 | 10 | 0 | 72 | 7.03 |
| 20 | 153 | 3 | 156 | 141 | 12 | 0 | 68 | 6.64 |
| 21 | 132 | 1 | 133 | 99 | 13 | 0 | 63 | 6.15 |
| 22 | 95 | 3 | 98 | 151 | 12 | 0 | 72 | 7.03 |
| 23 | 124 | 2 | 126 | 127 | 11 | 0 | 67 | 6.54 |
| 24 | 113 | 2 | 115 | 141 | 12 | 0 | 68 | 6.64 |
| 25 | 101 | 2 | 103 | 75 | 7 | 0 | 67 | 6.54 |
| 26 | 134 | 0 | 134 | 59 | 8 | 0 | 63 | 6.15 |
| 27 | 104 | 2 | 106 | 140 | 12 | 0 | 67 | 6.54 |
| 28 | 140 | 2 | 142 | 152 | 12 | 0 | 72 | 7.03 |
| 29 | 158 | 2 | 160 | 141 | 12 | 0 | 68 | 6.64 |
| 30 | 107 | 2 | 109 | 83 | 11 | 0 | 63 | 6.15 |
| Average | 113.9 | 4.6 | 118.5 | 122.83 | 11.26 |  | 67.56 | 6.59 |

## Appendix G: Process Data Table (FCFS, 4 Threads, Cache)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of Cache** | **% of RAM** |
| 1 | 15 | 17 | 32 | 140 | 12 | 0 | 67 | 6.54 |
| 2 | 15 | 21 | 36 | 151 | 12 | 1 | 72 | 7.03 |
| 3 | 15 | 34 | 49 | 141 | 12 | 2 | 68 | 6.64 |
| 4 | 15 | 27 | 42 | 91 | 12 | 3 | 63 | 6.15 |
| 5 | 32 | 16 | 48 | 151 | 12 | 0 | 72 | 7.03 |
| 6 | 36 | 15 | 51 | 141 | 12 | 1 | 68 | 6.64 |
| 7 | 48 | 7 | 55 | 91 | 12 | 0 | 63 | 6.15 |
| 8 | 48 | 24 | 72 | 91 | 12 | 3 | 63 | 6.15 |
| 9 | 48 | 22 | 70 | 141 | 12 | 2 | 68 | 6.64 |
| 10 | 54 | 4 | 58 | 151 | 12 | 1 | 72 | 7.03 |
| 11 | 56 | 20 | 76 | 140 | 12 | 0 | 67 | 6.54 |
| 12 | 53 | 22 | 75 | 152 | 12 | 1 | 72 | 7.03 |
| 13 | 63 | 8 | 71 | 141 | 12 | 2 | 68 | 6.64 |
| 14 | 65 | 9 | 74 | 75 | 10 | 3 | 63 | 6.15 |
| 15 | 69 | 7 | 76 | 128 | 11 | 0 | 68 | 6.64 |
| 16 | 71 | 36 | 107 | 35 | 5 | 2 | 63 | 6.15 |
| 17 | 72 | 37 | 109 | 154 | 12 | 1 | 72 | 7.03 |
| 18 | 73 | 35 | 108 | 141 | 12 | 3 | 68 | 6.64 |
| 19 | 75 | 36 | 111 | 121 | 10 | 0 | 72 | 7.03 |
| 20 | 107 | 6 | 113 | 141 | 12 | 2 | 68 | 6.64 |
| 21 | 108 | 4 | 112 | 99 | 13 | 3 | 63 | 6.15 |
| 22 | 113 | 3 | 116 | 151 | 12 | 0 | 72 | 7.03 |
| 23 | 113 | 9 | 122 | 127 | 11 | 1 | 67 | 6.54 |
| 24 | 113 | 6 | 119 | 141 | 12 | 3 | 68 | 6.64 |
| 25 | 114 | 7 | 121 | 75 | 7 | 2 | 67 | 6.54 |
| 26 | 116 | 6 | 122 | 59 | 8 | 0 | 63 | 6.15 |
| 27 | 122 | 2 | 124 | 140 | 12 | 3 | 67 | 6.54 |
| 28 | 122 | 8 | 130 | 152 | 12 | 0 | 72 | 7.03 |
| 29 | 122 | 14 | 136 | 141 | 12 | 1 | 68 | 6.64 |
| 30 | 122 | 12 | 134 | 83 | 11 | 2 | 63 | 6.15 |
| Average | 73.16 | 15.8 | 88.96 | 122.83 | 11.26 |  | 67.56 | 6.59 |

## Appendix H: Process Data Table (Priority, 4 Threads, Cache)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of Cache** | **% of RAM** |
| 1 | 59 | 5 | 64 | 140 | 12 | 0 | 67 | 6.54 |
| 2 | 43 | 6 | 49 | 151 | 12 | 2 | 72 | 7.03 |
| 3 | 33 | 4 | 37 | 141 | 12 | 1 | 68 | 6.64 |
| 4 | 33 | 2 | 35 | 91 | 12 | 2 | 63 | 6.15 |
| 5 | 52 | 4 | 56 | 151 | 12 | 2 | 72 | 7.03 |
| 6 | 28 | 5 | 33 | 141 | 12 | 3 | 68 | 6.64 |
| 7 | 36 | 3 | 39 | 91 | 12 | 3 | 63 | 6.15 |
| 8 | 5 | 7 | 12 | 91 | 12 | 0 | 63 | 6.15 |
| 9 | 62 | 4 | 66 | 141 | 12 | 1 | 68 | 6.64 |
| 10 | 6 | 17 | 23 | 151 | 12 | 3 | 72 | 7.03 |
| 11 | 53 | 4 | 57 | 140 | 12 | 0 | 67 | 6.54 |
| 12 | 12 | 9 | 21 | 152 | 12 | 0 | 72 | 7.03 |
| 13 | 5 | 4 | 9 | 141 | 12 | 1 | 68 | 6.64 |
| 14 | 63 | 11 | 74 | 75 | 10 | 0 | 63 | 6.15 |
| 15 | 37 | 3 | 40 | 128 | 11 | 0 | 68 | 6.64 |
| 16 | 23 | 1 | 24 | 35 | 5 | 1 | 63 | 6.15 |
| 17 | 42 | 4 | 46 | 154 | 12 | 3 | 72 | 7.03 |
| 18 | 5 | 13 | 18 | 141 | 12 | 2 | 68 | 6.64 |
| 19 | 65 | 11 | 76 | 121 | 10 | 1 | 72 | 7.03 |
| 20 | 64 | 5 | 69 | 141 | 12 | 2 | 68 | 6.64 |
| 21 | 47 | 2 | 49 | 99 | 13 | 0 | 63 | 6.15 |
| 22 | 8 | 6 | 14 | 151 | 12 | 1 | 72 | 7.03 |
| 23 | 37 | 3 | 40 | 127 | 11 | 1 | 67 | 6.54 |
| 24 | 26 | 4 | 30 | 141 | 12 | 0 | 68 | 6.64 |
| 25 | 14 | 7 | 21 | 75 | 7 | 1 | 67 | 6.54 |
| 26 | 49 | 2 | 51 | 59 | 8 | 1 | 63 | 6.15 |
| 27 | 22 | 4 | 26 | 140 | 12 | 0 | 67 | 6.54 |
| 28 | 54 | 4 | 58 | 152 | 12 | 1 | 72 | 7.03 |
| 29 | 66 | 4 | 70 | 141 | 12 | 3 | 68 | 6.64 |
| 30 | 22 | 2 | 24 | 83 | 11 | 2 | 63 | 6.15 |
| Average | 35.7 | 5.33 | 41.03 | 122.83 | 11.26 |  | 67.57 | 6.59 |

## Appendix I: Process Data Table (SJF, 1 Thread, Cache)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of Cache** | **% of RAM** |
| 1 | 60 | 2 | 62 | 140 | 12 | 0 | 67 | 6.54 |
| 2 | 98 | 2 | 100 | 151 | 12 | 0 | 72 | 7.03 |
| 3 | 72 | 2 | 74 | 141 | 12 | 0 | 68 | 6.64 |
| 4 | 4 | 46 | 50 | 91 | 12 | 0 | 63 | 6.15 |
| 5 | 100 | 2 | 102 | 151 | 12 | 0 | 72 | 7.03 |
| 6 | 75 | 2 | 77 | 141 | 12 | 0 | 68 | 6.64 |
| 7 | 50 | 1 | 51 | 91 | 12 | 0 | 63 | 6.15 |
| 8 | 51 | 1 | 52 | 91 | 12 | 0 | 63 | 6.15 |
| 9 | 77 | 3 | 80 | 141 | 12 | 0 | 68 | 6.64 |
| 10 | 102 | 2 | 104 | 151 | 12 | 0 | 72 | 7.03 |
| 11 | 61 | 2 | 63 | 140 | 12 | 0 | 67 | 6.54 |
| 12 | 104 | 2 | 106 | 152 | 12 | 0 | 72 | 7.03 |
| 13 | 80 | 2 | 82 | 141 | 12 | 0 | 68 | 6.64 |
| 14 | 52 | 1 | 53 | 75 | 10 | 0 | 63 | 6.15 |
| 15 | 82 | 3 | 85 | 128 | 11 | 0 | 68 | 6.64 |
| 16 | 53 | 0 | 53 | 35 | 5 | 0 | 63 | 6.15 |
| 17 | 107 | 2 | 109 | 154 | 12 | 0 | 72 | 7.03 |
| 18 | 86 | 2 | 88 | 141 | 12 | 0 | 68 | 6.64 |
| 19 | 110 | 2 | 112 | 121 | 10 | 0 | 72 | 7.03 |
| 20 | 88 | 2 | 90 | 141 | 12 | 0 | 68 | 6.64 |
| 21 | 53 | 1 | 54 | 99 | 13 | 0 | 63 | 6.15 |
| 22 | 111 | 2 | 113 | 151 | 12 | 0 | 72 | 7.03 |
| 23 | 63 | 2 | 65 | 127 | 11 | 0 | 67 | 6.54 |
| 24 | 90 | 2 | 92 | 141 | 12 | 0 | 68 | 6.64 |
| 25 | 65 | 1 | 66 | 75 | 7 | 0 | 67 | 6.54 |
| 26 | 54 | 1 | 55 | 59 | 8 | 0 | 63 | 6.15 |
| 27 | 67 | 2 | 69 | 140 | 12 | 0 | 67 | 6.54 |
| 28 | 114 | 2 | 116 | 152 | 12 | 0 | 72 | 7.03 |
| 29 | 92 | 2 | 94 | 141 | 12 | 0 | 68 | 6.64 |
| 30 | 55 | 1 | 56 | 83 | 11 | 0 | 63 | 6.15 |
| Average | 75.86 | 3.23 | 79.1 | 122.83 | 11.26 |  | 67.56 | 6.59 |

## Appendix J: Process Data Table (SJF, 4 Threads, Cache)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of Cache** | **% of RAM** |
| 1 | 16 | 2 | 18 | 140 | 12 | 2 | 67 | 6.54 |
| 2 | 75 | 6 | 81 | 151 | 12 | 0 | 72 | 7.03 |
| 3 | 21 | 36 | 57 | 141 | 12 | 3 | 68 | 6.64 |
| 4 | 3 | 3 | 6 | 91 | 12 | 0 | 63 | 6.15 |
| 5 | 75 | 9 | 84 | 151 | 12 | 1 | 72 | 7.03 |
| 6 | 23 | 38 | 61 | 141 | 12 | 0 | 68 | 6.64 |
| 7 | 3 | 4 | 7 | 91 | 12 | 1 | 63 | 6.15 |
| 8 | 5 | 1 | 6 | 91 | 12 | 2 | 63 | 6.15 |
| 9 | 24 | 29 | 53 | 141 | 12 | 1 | 68 | 6.64 |
| 10 | 76 | 9 | 85 | 151 | 12 | 2 | 72 | 7.03 |
| 11 | 15 | 5 | 20 | 140 | 12 | 3 | 67 | 6.54 |
| 12 | 77 | 11 | 88 | 152 | 12 | 3 | 72 | 7.03 |
| 13 | 54 | 4 | 58 | 141 | 12 | 1 | 68 | 6.64 |
| 14 | 6 | 2 | 8 | 75 | 10 | 0 | 63 | 6.15 |
| 15 | 54 | 6 | 60 | 128 | 11 | 2 | 68 | 6.64 |
| 16 | 6 | 2 | 8 | 35 | 5 | 2 | 63 | 6.15 |
| 17 | 81 | 10 | 91 | 154 | 12 | 0 | 72 | 7.03 |
| 18 | 61 | 14 | 75 | 141 | 12 | 2 | 68 | 6.64 |
| 19 | 83 | 9 | 92 | 121 | 10 | 1 | 72 | 7.03 |
| 20 | 60 | 15 | 75 | 141 | 12 | 3 | 68 | 6.64 |
| 21 | 5 | 7 | 12 | 99 | 13 | 3 | 63 | 6.15 |
| 22 | 85 | 12 | 97 | 151 | 12 | 2 | 72 | 7.03 |
| 23 | 14 | 7 | 21 | 127 | 11 | 0 | 67 | 6.54 |
| 24 | 60 | 12 | 72 | 141 | 12 | 0 | 68 | 6.64 |
| 25 | 13 | 8 | 21 | 75 | 7 | 1 | 67 | 6.54 |
| 26 | 5 | 7 | 12 | 59 | 8 | 1 | 63 | 6.15 |
| 27 | 16 | 34 | 50 | 140 | 12 | 2 | 67 | 6.54 |
| 28 | 87 | 10 | 97 | 152 | 12 | 3 | 72 | 7.03 |
| 29 | 60 | 11 | 71 | 141 | 12 | 1 | 68 | 6.64 |
| 30 | 6 | 6 | 12 | 83 | 11 | 0 | 63 | 6.15 |
| Average | 38.96 | 10.96 | 49.93 | 122.83 | 11.26 |  | 67.56 | 6.59 |

## Appendix K: Process Data Table (SJF, 4 Threads, Cache, Paging)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of Cache** | **% of RAM** |
| 1 | 166 | 3 | 169 | 277 | 21 | 3 | 100 | 3.90625 |
| 2 | 198 | 2 | 200 | 208 | 39 | 0 | 100 | 4.296875 |
| 3 | 182 | 6 | 188 | 222 | 34 | 0 | 100 | 3.90625 |
| 4 | 93 | 16 | 109 | 816 | 657 | 0 | 100 | 3.515625 |
| 5 | 195 | 2 | 197 | 146 | 12 | 0 | 100 | 4.296875 |
| 6 | 176 | 3 | 179 | 143 | 12 | 0 | 100 | 3.90625 |
| 7 | 74 | 11 | 85 | 680 | 522 | 2 | 100 | 3.515625 |
| 8 | 53 | 14 | 67 | 840 | 661 | 2 | 100 | 3.125 |
| 9 | 175 | 6 | 181 | 226 | 17 | 2 | 100 | 3.90625 |
| 10 | 190 | 3 | 193 | 149 | 15 | 0 | 100 | 4.296875 |
| 11 | 168 | 6 | 174 | 603 | 128 | 3 | 100 | 3.90625 |
| 12 | 195 | 2 | 197 | 159 | 17 | 0 | 100 | 4.296875 |
| 13 | 173 | 3 | 176 | 144 | 13 | 3 | 100 | 3.90625 |
| 14 | 40 | 0 | 40 | 30 | 12 | 3 | 100 | 2.734375 |
| 15 | 180 | 5 | 185 | 186 | 12 | 0 | 100 | 3.90625 |
| 16 | 61 | 7 | 68 | 213 | 176 | 3 | 100 | 2.734375 |
| 17 | 190 | 4 | 194 | 223 | 21 | 2 | 100 | 4.296875 |
| 18 | 181 | 5 | 186 | 157 | 16 | 0 | 100 | 3.90625 |
| 19 | 191 | 5 | 196 | 406 | 73 | 1 | 100 | 4.296875 |
| 20 | 174 | 2 | 176 | 146 | 16 | 2 | 100 | 3.90625 |
| 21 | 124 | 22 | 146 | 1258 | 991 | 0 | 100 | 3.515625 |
| 22 | 191 | 2 | 193 | 151 | 17 | 3 | 100 | 4.296875 |
| 23 | 165 | 2 | 167 | 157 | 16 | 0 | 100 | 3.90625 |
| 24 | 176 | 2 | 178 | 197 | 12 | 0 | 100 | 3.90625 |
| 25 | 164 | 1 | 165 | 91 | 10 | 0 | 100 | 3.515625 |
| 26 | 152 | 10 | 162 | 450 | 318 | 2 | 100 | 3.125 |
| 27 | 163 | 1 | 164 | 155 | 14 | 3 | 100 | 3.90625 |
| 28 | 189 | 3 | 192 | 147 | 12 | 1 | 100 | 4.296875 |
| 29 | 178 | 6 | 184 | 145 | 14 | 3 | 100 | 3.90625 |
| 30 | 134 | 7 | 141 | 274 | 187 | 3 | 100 | 2.734375 |
| Average | 156.36 | 5.36 | 161.73 | 299.96 | 135.5 |  | 100 | 3.7890625 |

## Appendix L: Process Data Table (Priority, 4 Threads, Cache, Paging)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of Cache** | **% of RAM** |
| 1 | 661 | 4 | 665 | 198 | 21 | 1 | 100 | 3.90625 |
| 2 | 613 | 12 | 625 | 155 | 14 | 0 | 100 | 4.296875 |
| 3 | 565 | 6 | 571 | 157 | 14 | 1 | 100 | 3.90625 |
| 4 | 591 | 29 | 620 | 196 | 144 | 1 | 100 | 3.125 |
| 5 | 638 | 2 | 640 | 111 | 11 | 3 | 100 | 3.90625 |
| 6 | 562 | 9 | 571 | 148 | 17 | 3 | 100 | 3.90625 |
| 7 | 589 | 31 | 620 | 304 | 178 | 0 | 100 | 3.125 |
| 8 | 57 | 96 | 153 | 593 | 472 | 1 | 100 | 2.734375 |
| 9 | 647 | 3 | 650 | 192 | 19 | 3 | 100 | 3.90625 |
| 10 | 161 | 18 | 179 | 169 | 14 | 2 | 100 | 4.296875 |
| 11 | 635 | 4 | 639 | 218 | 25 | 1 | 100 | 3.90625 |
| 12 | 221 | 8 | 229 | 232 | 17 | 2 | 100 | 4.296875 |
| 13 | 112 | 63 | 175 | 157 | 14 | 1 | 100 | 3.90625 |
| 14 | 652 | 12 | 664 | 489 | 274 | 1 | 100 | 3.125 |
| 15 | 580 | 14 | 594 | 255 | 67 | 2 | 100 | 3.90625 |
| 16 | 507 | 52 | 559 | 1741 | 1627 | 3 | 100 | 2.734375 |
| 17 | 621 | 6 | 627 | 180 | 17 | 3 | 100 | 4.296875 |
| 18 | 83 | 91 | 174 | 235 | 17 | 3 | 100 | 3.90625 |
| 19 | 667 | 4 | 671 | 119 | 12 | 2 | 100 | 4.296875 |
| 20 | 664 | 7 | 671 | 225 | 30 | 2 | 100 | 3.90625 |
| 21 | 616 | 14 | 630 | 200 | 88 | 2 | 100 | 3.125 |
| 22 | 136 | 85 | 221 | 245 | 15 | 3 | 100 | 4.296875 |
| 23 | 585 | 8 | 593 | 176 | 59 | 0 | 100 | 3.90625 |
| 24 | 557 | 12 | 569 | 165 | 23 | 2 | 100 | 3.90625 |
| 25 | 186 | 35 | 221 | 77 | 8 | 1 | 100 | 3.515625 |
| 26 | 625 | 4 | 629 | 127 | 62 | 0 | 100 | 3.125 |
| 27 | 219 | 7 | 226 | 169 | 17 | 2 | 100 | 3.90625 |
| 28 | 636 | 6 | 642 | 319 | 48 | 3 | 100 | 4.296875 |
| 29 | 665 | 5 | 670 | 238 | 20 | 0 | 100 | 3.90625 |
| 30 | 308 | 179 | 487 | 2039 | 1969 | 1 | 100 | 2.734375 |
| Average | 478.63 | 27.53 | 506.16 | 327.63 | 177.1 |  | 100 | 3.7369792 |

## Appendix M: Process Data Table (FCFS, 4 Threads, Cache, Paging)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **Wait Time (ms):** | **Run Time (ms):** | **Completion Time (ms):** | **Execution Count:** | **I/O Count:** | **CPU** | **% of Cache** | **% of RAM** |
| 1 | 35 | 113 | 148 | 256 | 26 | 0 | 100 | 3.90625 |
| 2 | 37 | 112 | 149 | 148 | 14 | 2 | 100 | 4.29688 |
| 3 | 57 | 91 | 148 | 144 | 13 | 2 | 100 | 3.90625 |
| 4 | 53 | 134 | 187 | 493 | 285 | 3 | 100 | 3.51563 |
| 5 | 164 | 13 | 177 | 152 | 18 | 2 | 100 | 4.29688 |
| 6 | 174 | 11 | 185 | 203 | 72 | 3 | 100 | 3.90625 |
| 7 | 183 | 15 | 198 | 391 | 229 | 3 | 100 | 3.51563 |
| 8 | 178 | 27 | 205 | 303 | 187 | 3 | 100 | 3.51563 |
| 9 | 187 | 24 | 211 | 144 | 13 | 3 | 100 | 3.90625 |
| 10 | 190 | 28 | 218 | 224 | 19 | 0 | 100 | 4.29688 |
| 11 | 189 | 30 | 219 | 148 | 19 | 3 | 100 | 3.90625 |
| 12 | 207 | 14 | 221 | 153 | 18 | 2 | 100 | 4.29688 |
| 13 | 217 | 7 | 224 | 220 | 28 | 0 | 100 | 3.90625 |
| 14 | 221 | 9 | 230 | 276 | 129 | 0 | 100 | 3.125 |
| 15 | 225 | 8 | 233 | 152 | 18 | 3 | 100 | 3.90625 |
| 16 | 231 | 4 | 235 | 80 | 29 | 1 | 100 | 2.73438 |
| 17 | 232 | 5 | 237 | 159 | 15 | 1 | 100 | 4.29688 |
| 18 | 232 | 5 | 237 | 144 | 13 | 0 | 100 | 3.90625 |
| 19 | 236 | 5 | 241 | 272 | 45 | 3 | 100 | 4.29688 |
| 20 | 238 | 3 | 241 | 147 | 16 | 1 | 100 | 3.90625 |
| 21 | 243 | 7 | 250 | 374 | 182 | 0 | 100 | 3.51563 |
| 22 | 248 | 5 | 253 | 114 | 17 | 0 | 100 | 3.90625 |
| 23 | 250 | 5 | 255 | 215 | 39 | 2 | 100 | 3.90625 |
| 24 | 252 | 4 | 256 | 243 | 23 | 0 | 100 | 3.90625 |
| 25 | 253 | 6 | 259 | 98 | 22 | 2 | 100 | 3.51563 |
| 26 | 258 | 30 | 288 | 217 | 119 | 2 | 100 | 3.125 |
| 27 | 261 | 23 | 284 | 142 | 13 | 0 | 100 | 3.90625 |
| 28 | 264 | 22 | 286 | 163 | 28 | 3 | 100 | 4.29688 |
| 29 | 264 | 23 | 287 | 162 | 31 | 0 | 100 | 3.90625 |
| 30 | 288 | 19 | 307 | 507 | 306 | 3 | 100 | 3.125 |
| Average | 202.23 | 26.73 | 228.96 | 214.8 | 66.2 |  | 100 | 3.8151 |

## Appendix N: Comparison of Page Faults and Average Page Fault Servicing Time in ns for FCFS, Priority, and SJF on One CPU

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Process ID:** | **SJF # of Page Faults:** | **SJF Average page fault servicing time (ns):** | **Priority # of Page Faults:** | **Priority Average page fault servicing time (ns):** | **FCFS # of Page Faults:** | **FCFS Average page fault servicing time (ns):** |
| 1 | 8 | 19309.875 | 6 | 7948.333333 | 6 | 72557.66667 |
| 2 | 7 | 18724.42857 | 7 | 10825.28571 | 7 | 65034.42857 |
| 3 | 6 | 9069.666667 | 6 | 13117 | 8 | 69668.75 |
| 4 | 13 | 36908.92308 | 11 | 17075.72727 | 13 | 32272.84615 |
| 5 | 7 | 7397.857143 | 7 | 7565 | 7 | 21483.42857 |
| 6 | 6 | 10971.16667 | 10 | 20714.1 | 8 | 15872.125 |
| 7 | 11 | 24363.45455 | 13 | 20547.38462 | 11 | 24841.90909 |
| 8 | 11 | 30746.63636 | 3 | 18627.33333 | 7 | 21107 |
| 9 | 6 | 62025.16667 | 6 | 10191.16667 | 12 | 47274.75 |
| 10 | 9 | 37221.44444 | 7 | 5433.428571 | 7 | 8860.857143 |
| 11 | 6 | 7314.333333 | 6 | 8143.333333 | 6 | 5802.833333 |
| 12 | 7 | 7941.142857 | 7 | 5182.714286 | 7 | 6812.714286 |
| 13 | 6 | 7314.5 | 6 | 11800.5 | 6 | 11702.83333 |
| 14 | 11 | 45481.63636 | 12 | 28403.66667 | 12 | 24527.16667 |
| 15 | 10 | 12141.8 | 6 | 11654.33333 | 6 | 13897.16667 |
| 16 | 12 | 35962.08333 | 3 | 13848.33333 | 3 | 13360.66667 |
| 17 | 7 | 19894.71429 | 7 | 13750.57143 | 8 | 13458.25 |
| 18 | 6 | 18432 | 6 | 18676 | 6 | 13848.33333 |
| 19 | 7 | 22862.42857 | 16 | 15268.5625 | 7 | 6980 |
| 20 | 6 | 7704.5 | 6 | 9509 | 6 | 14969.83333 |
| 21 | 13 | 39317.07692 | 12 | 17285.91667 | 12 | 36083.91667 |
| 22 | 7 | 5809.714286 | 7 | 6645.571429 | 7 | 12705.57143 |
| 23 | 6 | 14726.33333 | 6 | 13068.16667 | 6 | 16383.83333 |
| 24 | 6 | 9655 | 6 | 11702.83333 | 6 | 10922.5 |
| 25 | 5 | 19894.8 | 7 | 16593 | 5 | 6846.2 |
| 26 | 3 | 74118.33333 | 12 | 22040.25 | 12 | 31646.5 |
| 27 | 8 | 21540.25 | 6 | 16432.83333 | 6 | 10386.16667 |
| 28 | 7 | 7982.857143 | 7 | 10616.28571 | 7 | 3803.142857 |
| 29 | 6 | 2535.833333 | 6 | 6728.833333 | 8 | 14774.875 |
| 30 | 13 | 34028.46154 | 3 | 26819 | 4 | 13750.75 |
|  | **7.866666667** | **22379.88059** | **7.43** | **13873.88059** | **7.533** | **22054.56** |

## Appendix O: Comparison of Running Times for FCFS, Priority, and SJF on One CPU

## Appendix P: Comparison of Waiting Times for FCFS, Priority, SJF on One CPU

## Appendix Q: Comparison of Running Times for FCFS, Priority, SJF on Four CPUs

## Appendix R: Comparison of Waiting Times for FCFS, Priority, SJF on Four CPUs

## Appendix S: Comparison of Completion Times for FCFS, Priority, SJF on Four CPUs

## Appendix T: Comparison of Waiting Times for FCFS and Priority and the Effects of Cache

## Appendix U: Comparison of Running Times for FCFS and Priority and the Effects of Cache

## Appendix V: Comparison of Running Times for FCFS and Priority and the Effects of Cache