Yeezus Operating System

Project Report

Version 2.0

Mark Zeagler, Jessica Brummel,

Nick Curtin, Pablo Avecilla, Selena Guillen

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 28

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 Single-threaded FCFS and Priority Comparison 31

7.2 Percentage of RAM and Cache Used 31

7.3 FCFS and Priority Comparison I/O, completion time, waiting time analysis 31

7.4 Jobs Assigned to each CPU 31

7.5 Comparison of Performance for 1-CPU vs N-CPU Runs 31

8. Conclusions 32

9. Supporting Information 33

9.1 Table of Figures 33

9.2 Appendix A: Instruction Format 34

9.3 Appendix B: Instruction Set 36

9.4 Appendix C – Process Data Table (FCFS Scheduling, 1 Thread) 37

9.5 Appendix D – Process Data Table (Priority Scheduling, 1 Thread) 38

9.6 Appendix E – Process Data Table (FCFS Scheduling, 4 Threads) 39

9.7 Appendix F – Process Data Table (Priority Scheduling, 4 Threads) 40

# Abstract

The following document will feature the explanation of the project’s modules, the design of the system architecture, the data collected by Part 1 and Part 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 Phase 2, Part 1 and 2. The goal for Phase 1 was to develop a working operating system emulation that read in a text file containing hex instructions and data, processed that data, and would output the data after all processes had been completed. The system for Phase 1 was to emulate a 1-core processor in Part 1, which was developed into a 4-core processor for Part 2, and emulated the registers, cache, RAM, and disk.

Phase 2 is meant to further develop this system to further simulate how a modern computer operates. The goal for Part 1 was to add a third scheduling algorithm, Shortest Job First. This scheduling algorithm chooses the process with the fewest instructions for execution. The goal for Part 2 was to introduce paging into the memory system and I/O blocking into the process execution.

# 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.

Throughout the course of this document, any instructions specified by Dr. Bobbie will be displayed in this font, to help differentiate from our group’s work.

# Overall Description

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.

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.

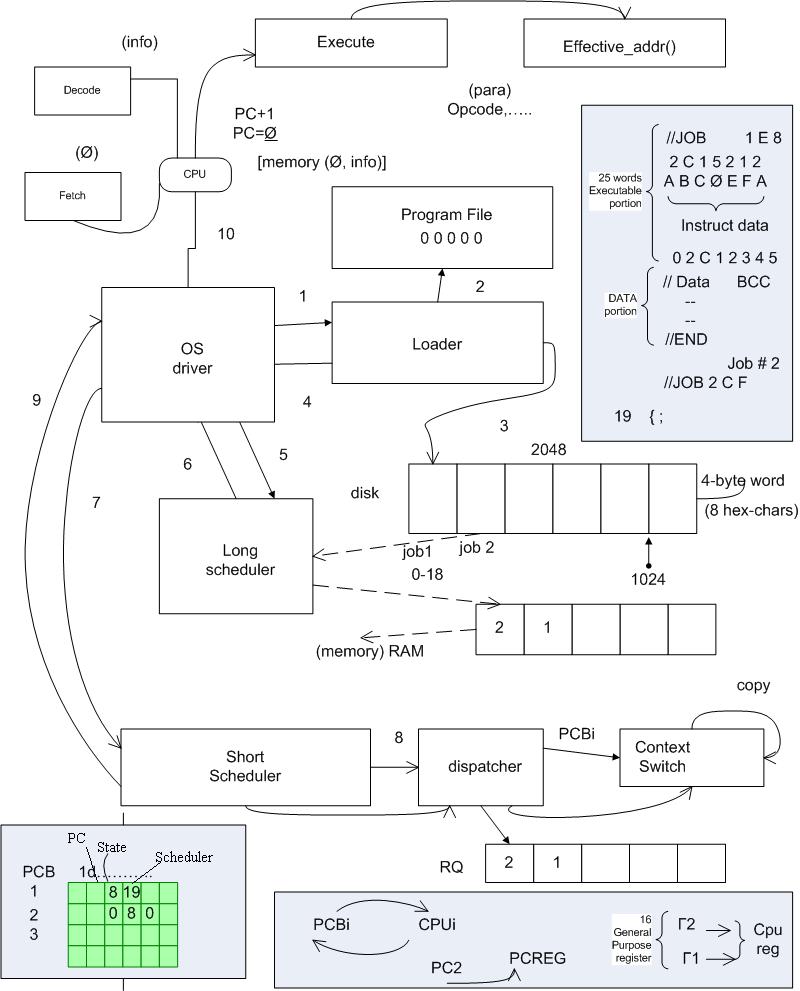


Figure 1 Overview of the Operating System components (provided by Dr. Bobbie).

# 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.

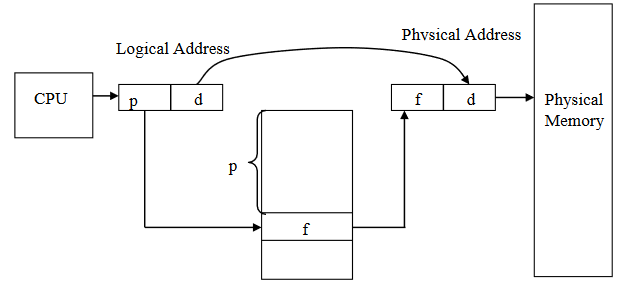


Figure 7 Diagram of a Page Table (provided by Dr. Bobbie).

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.

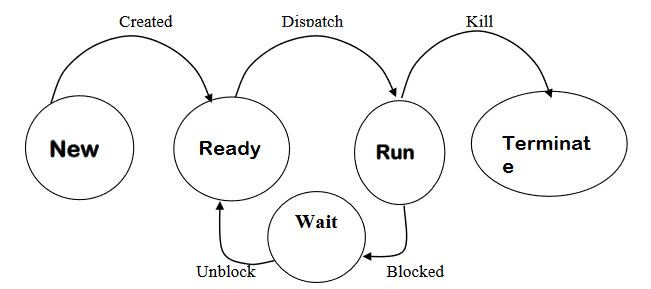


Figure 8 Diagram of process states (provided by Dr. Bobbie).

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

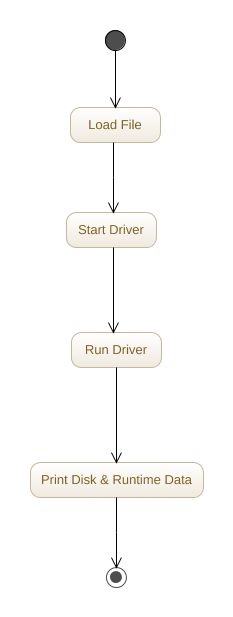


Figure 9 System overview diagram.

The 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.

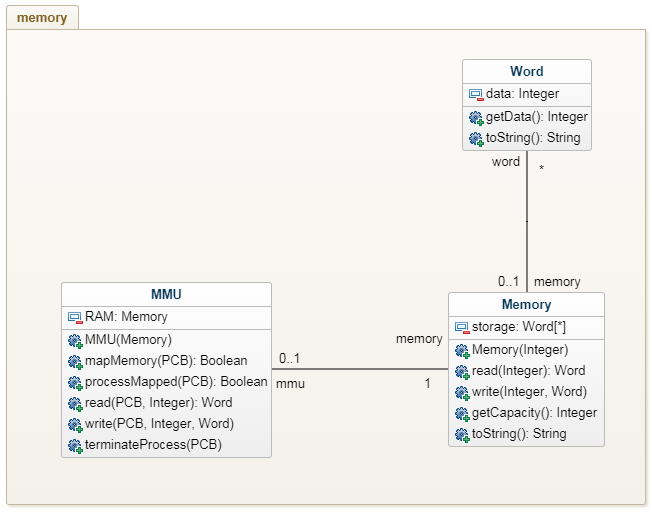


Figure 10 UML class diagram of the memory system.

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.

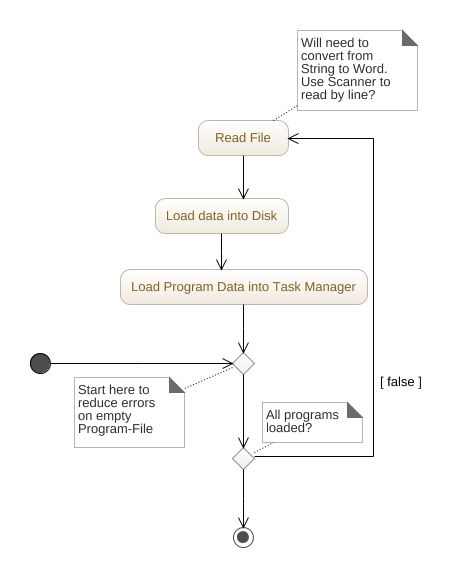


Figure 11 Activity diagram of the loader.

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

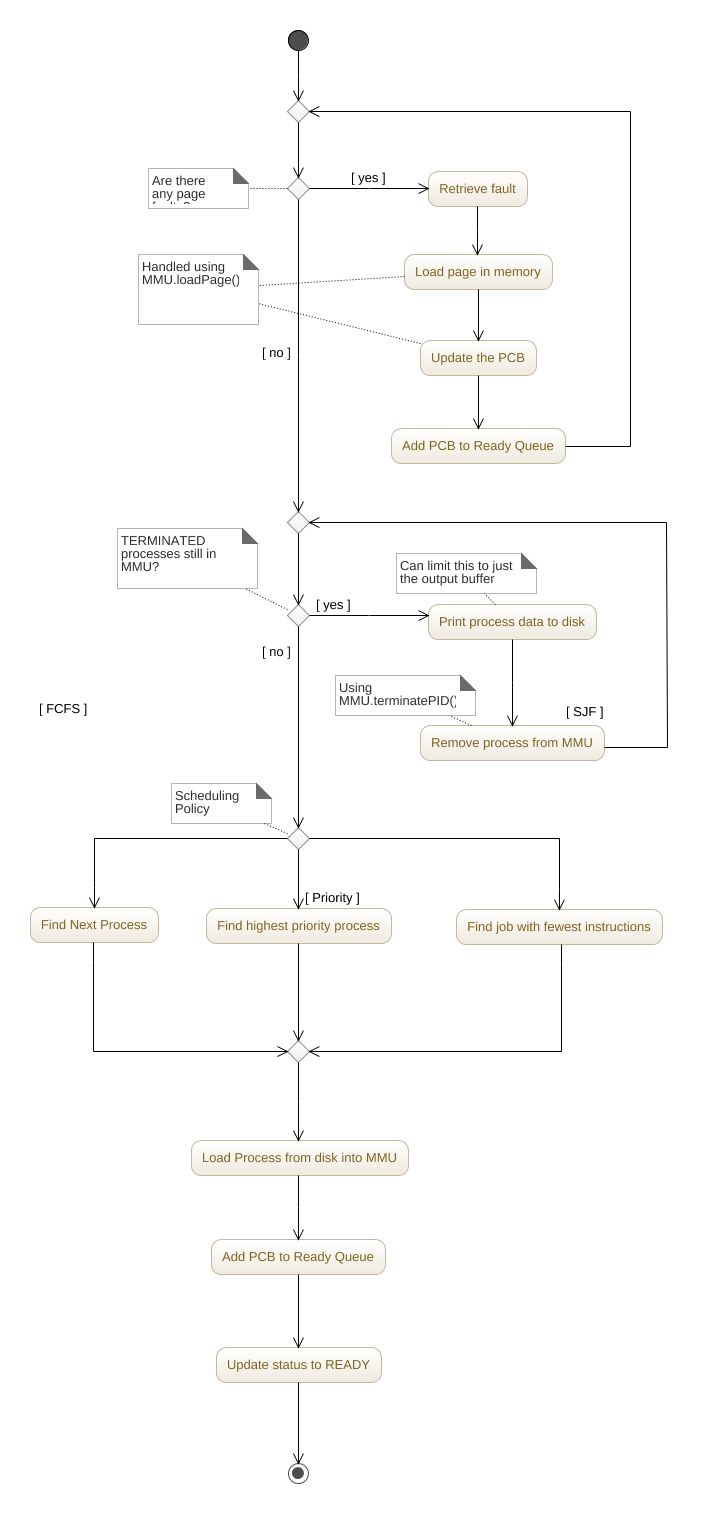


Figure 12 Activity diagram of the scheduler.

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

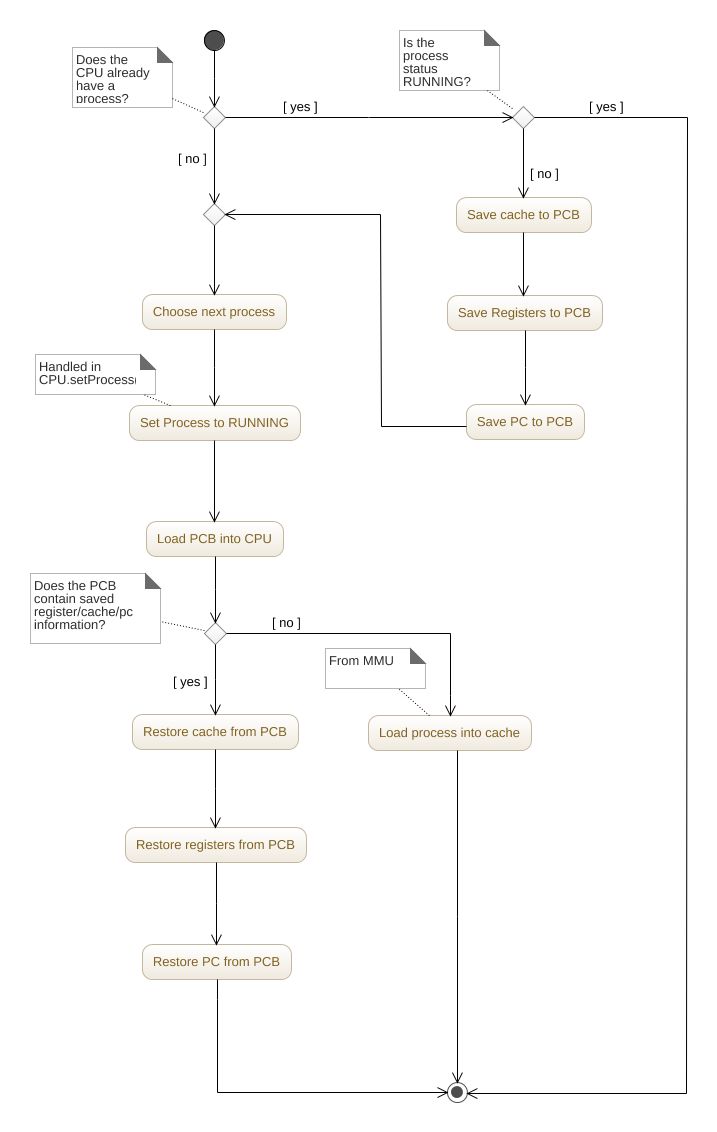


Figure 13 Activity diagram of the dispatcher.

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.

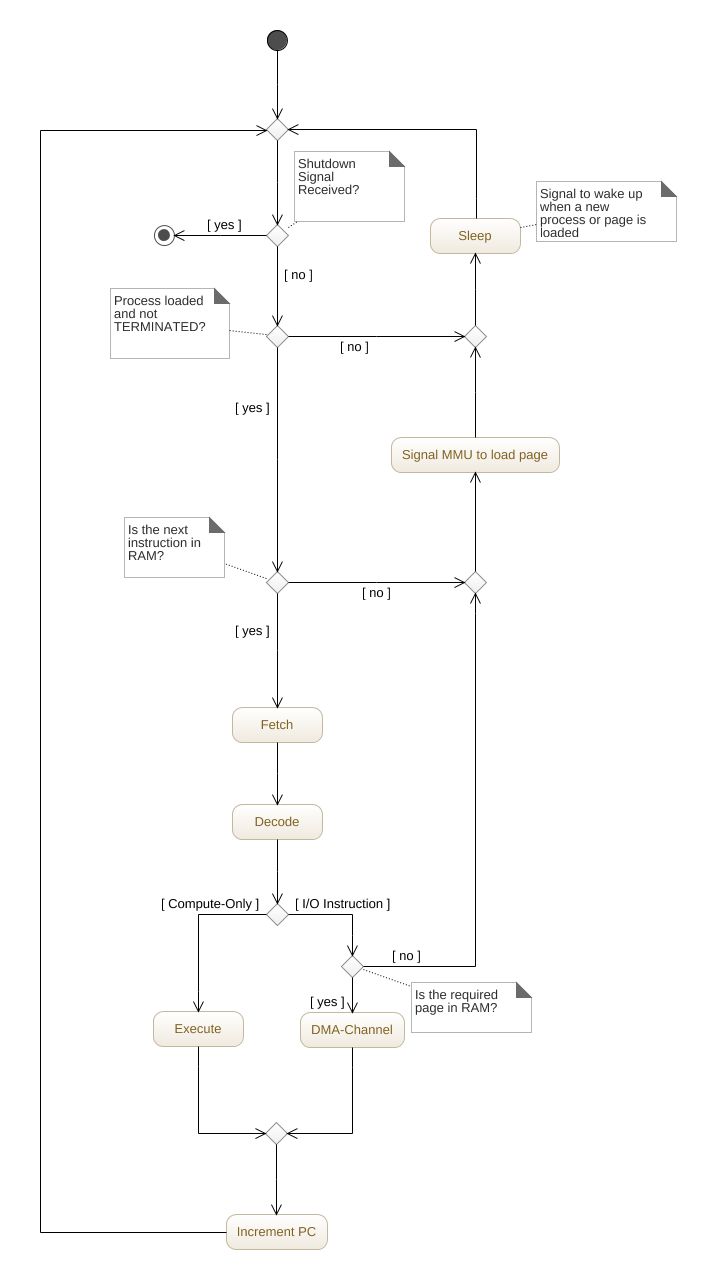


Figure 14 Activity diagram of the CPU.

## 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

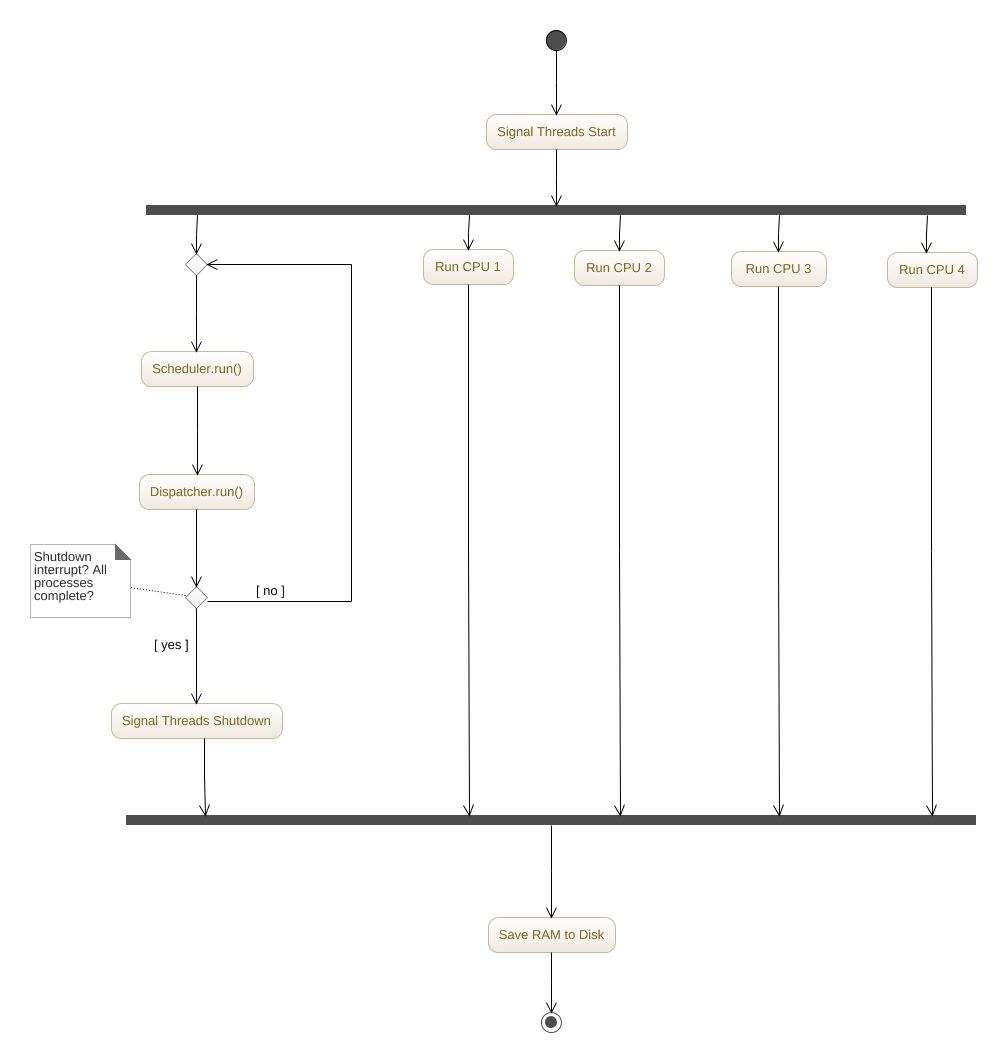
The fetch operation was designed to be a set of lines within the CPU’s execution loop instead of a separate component due to its simplicity. It was originally designed to read from the MMU but was later adapted to utilize the cache. Even with the further addition of the paged cache, nothing had to be 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

### DMA-Channel

## Multiprocessor Architecture



### 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

### Paging

### I/O Blocking

# Simulation Discussion

# Data Analysis

## Single-threaded FCFS and Priority Comparison

Priority scheduling and first come first serve scheduling differed wildly under wait and completion times. For FCFS, the average wait time was roughly 83 ms, and the average completion time was roughly 85 ms. This is a lot larger than the average wait time under priority scheduling, which was almost 12 ms, and the average completion time of roughly 13 ms. It is concluded that the priority scheduling algorithm resulted in the most favorable wait and completion times. The two scheduling algorithms also had the same number of I/O operations for each process.

## 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%.

## FCFS and Priority Comparison I/O, completion time, waiting time analysis

The two scheduling algorithms for the 4 CPU model had varied results. The FCFS scheduled OS has the following results: average waiting time was 33.13 ms, average completion time was: 42.44 ms, average cache usage: 67.56%, and average RAM usage: 6.59%. The Priority scheduled OS had the following data: average waiting time: 22.7 ms, average completion time: 25.7 ms, average cache used: 67.56%, average RAM used: 6.59%. From this data, it’s concluded that by using the priority scheduling algorithm, the operation system execution the jobs faster, as the waiting time was almost 8 ms less. The two algorithms however, did not see a difference in the number of I/O operations, because I/O operations is independent from the scheduling algorithm.

## Jobs Assigned to each CPU

|  |  |  |
| --- | --- | --- |
| CPU Number | Number of Jobs Assigned FCFS | Number of Jobs Assigned Priority |
| 0 | 8 | 10 |
| 1 | 7 | 8 |
| 2 | 7 | 6 |
| 3 | 8 | 6 |

Table 2 Jobs Assigned to each CPU.

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

The results from the single processor version of the OS, and the multiprocessor version of the OS concluded that having N-CPUs results in faster completion times, and lessens the average wait times. According to the data, the average completion time for priority scheduling on a 4 CPU OS was 25.7 ms, which is less than half of the single CPU priority scheduling time of 57.76 ms. Furthermore, the multiprocessors FCFS average completion time was 42.33 ms, while as the single processor had an average completion time of 467.43 ms. The multiprocessor had only 10% of the waiting time that the single processor had. The two different systems did not have a difference in percent of cache and percent ram used per each process, as the processes themselves did not change.

# Conclusions

# Supporting Information

## Table of Figures

[Figure 1 Overview of the Operating System components (provided by Dr. Bobbie). 6](#_Toc509786195)

[Figure 3 Overview of the Driver's flow (provided by Dr. Bobbie). 8](#_Toc509786196)

[Figure 4 Outline of the loader’s logic (provided by Dr. Bobbie). 9](#_Toc509786197)

[Figure 5 Heart of the DMA-channel module/thread (provided by Dr. Bobbie). 13](#_Toc509786198)

[Figure 6 Overview of the CPU’s execution (provided by Dr. Bobbie). 14](#_Toc509786199)

[Figure 7 The suggested structure of the PCB (provided by Dr. Bobbie). 14](#_Toc509786200)

## Appendix A: 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 Scheduling, 1 Thread)



## Appendix D – Process Data Table (Priority Scheduling, 1 Thread)



## Appendix E – Process Data Table (FCFS Scheduling, 4 Threads)



## Appendix F – Process Data Table (Priority Scheduling, 4 Threads)

