CS3502/01 Phase 1 Report

Prepared by:

Ryan Wheeler

Chad Forbes

Scott Seo

Alex Rusch

JD Knobloch

Matthew Rowe

0. Abstract and Introduction

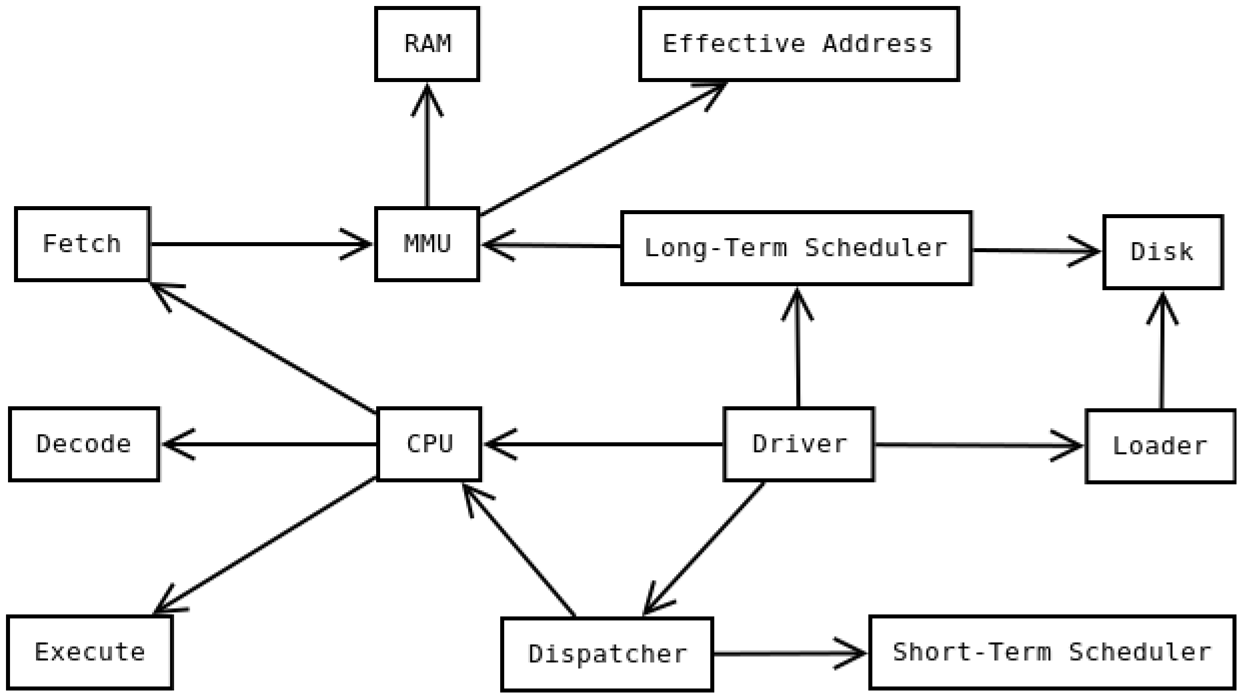
This document describes the design and implementation of a virtual machine solutions for phase 1 of the project. Also included is a discussion on simulated usages of this system (both single-processor and multi-processor versions). No metric data is included due to us not finishing the system on-time.

1. Design and Architecture

1.1 Part 1

The main components of the system are the Driver, CPU, MMU, RAM, Short-Term Scheduler, Long-Term Scheduler, Loader, Disk, and Dispatcher. The Driver serves as the entry point into the system, constructing all of the other components and operating them to provide the virtual machine's functionality.

Figure 1.1.A -- Single-processor virtual machine block diagram.



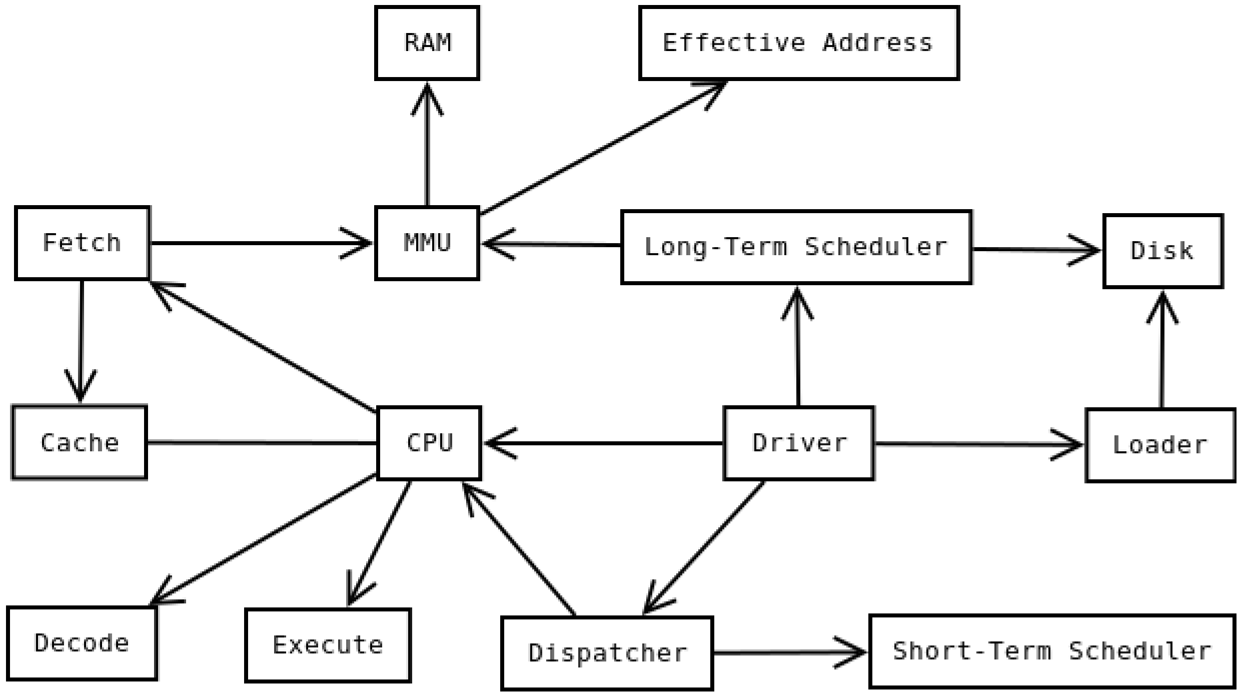
The order of execution inside of the Driver begins with it invoking the Loader, which loads the program files into the machine's disk and creates a PCB for each program. Then the Long Term Scheduler loads jobs from the disk into RAM via the MMU as it is able to. Next, the Dispatcher is invoked, which will place a process that has been loaded into memory onto the CPU if it is idle. Then the CPU performs its functions via the Compute Only components: those of Fetch, Decode, and Execute. Fetch acquires the next instruction via the MMU and passes it to the

Decoder, which determines the type of instruction and extracts register data from it, and then the Executor performs the operation. After this, the Driver repeats the process, beginning with the Loader, until all jobs have been finished, at which point, the virtual machine exits.

1.2 Part 2

In updating the architecture to support multi-core processing, three main features need to be added: the Dispatcher needs to act on all processors in an asymmetric manner, caches need to be added to the processors, and a mutex semaphore must be used in order to access memory.

Figure 1.2.A -- Multiprocessor virtual machine block diagram.



2. Implementation Modules

Following in this section is an enumeration of the major modules implemented in the system, organized by general purpose. Each module includes a brief description of its purpose and function.

2.1 Driver

This module drives the simulation's different modules: it sets up the virtual environment by loading jobs into the virtual machine's disk, loads those jobs into RAM when space is available, schedules ready jobs to be executed, and then sequentially executes each instruction of the current job.

2.2 CPU

2.2.1 Fetch
This module fetches (via the MMU) the next instruction to be processed according to the CPU's Program Counter.

2.2.2 Decode
This module translates a fetched hexadecimal instruction into binary and sets relevant register data on the CPU.

2.2.3 Execute
This module performs the actions associated with the decoded instruction and increments the Program Counter.

2.2.4 Computer Only
Sequentially invokes the Fetch, Decode, and Execute modules to emulate one clock tick on the CPU.

2.3 MMU

2.3.1 Memory
The MMU wraps access to the virtual machine's RAM and serves as a go-between for all operations that require reading or writing to or from RAM. It takes in read and write requests to virtual addresses and performs the reads and writes to the appropriate physical addresses.

2.3.2 Effective Address
This module translates a virtual address into a physical address and returns it to the invoking component.

2.4 RAM
This module can only be interacted with via the MMU wrapper. It manages all words stored in the virtual machine's RAM and allows for reading and writing to/from physical addresses.

2.5 Disk
This module is the simulated hard disk of the virtual machine. It stores and provides means of reading and writing words on the disk.

2.6 Short Term Scheduler
This module manages the ready queue and provides the next ready job to the dispatcher when it is requested. The short term scheduler provides both FIFO and non-preemptive priority ordering of jobs.

2.7 Long Term Scheduler
This module loads jobs from the virtual machine's disk into its RAM, allocating RAM space for each process as it becomes available.

2.8 Loader
At the beginning of the simulation, this module loads the programs to be executed into the disk of the virtual machine and parses information for each job into a new PCB that will eventually be loaded into memory by the long term scheduler.

2.9 Dispatcher
This module moves the current job off of a CPU and places the next job on. In doing so, it offloads the CPU's registers into the PCB of the job it is taking off, and then loads the registers and other data of the PCB of the new job onto the CPU.

2.10 PCB
This module stores all pertinent information about a job in the virtual machine, including its addresses in memory, its registers, program counter, state, and priority.

2.11 DMA
This module handles I/O operations in-place of a CPU, so all CPU's can spend more time handling computation operations, which are performed much faster than reads and writes.

3. Simulation
This section details the flow the control through an execution of the virtual machine. Part 1 discusses the single-processor version of the system, and Part 2 discusses the additions made to make it into a multiprocessor system.

3.1 Part 1
For part 1, the simulation begins in the driver module. It constructs all of the objects required for the system to operate: the Disk, RAM, schedulers, CPU, Loader, Dispatcher, etc.

Figure 3.1.A -- Code snippet for the Driver module.

The Driver then loads all of the processes that will be run into the Disk via the Loader module. While doing so, a new PCB is created for each job to store information about that job, including its addresses, priority, and size.

Figure 3.1.B -- Code snippet for the Loader module.

The Driver then begins to loop, consecutively loading jobs into RAM, placing jobs on the CPU, and executing the current job. This process continues until all thirty jobs in the system have been run.

The Long Term Scheduler checks the space required for each of the jobs that have been loaded into the system, but have not been loaded into RAM or run. When space is available for a job, it is loaded into RAM and its PCB is placed in the ready queue.

Figure 3.1.C -- Code snippet for the Long Term Scheduler.

If the CPU is idle, the Dispatcher requests the next process to be run from the ready queue, as determined by the scheduling algorithm used by the Short Term Scheduler. The Dispatcher then loads the PCB of the next job onto the CPU.

Figure 3.1.D -- Code snippet for the Dispatcher.

```
public static void loadJobs(){
    if (Driver.CPU.isIdle() && Driver.ShortTermScheduler.readyQueue.size() > 0){
      PCB pcb =
Driver.ShortTermScheduler.readyQueue.remove(Driver.ShortTermScheduler.readyQueue.size());
      if (Driver.CPU.shouldUnload()){
        System.out.println("Job: " + Driver.CPU.currentJobNumber());
        EndJobs();
```

The Compute Only module is then invoked, which performs instruction fetching, decoding, and execution for one cycle of the CPU and increments the Program Counter.

Figure 3.1.E -- Code snippet for the Computer Only module.

Figure 3.1.F -- Code snippet for the Fetch module.

```
public String fetch(int progCounter)
        {
                String instruction = cache[progCounter];
                return instruction;
        }
```

Figure 3.1.G -- Code snippet for the Decode module.

```
                instructionType = Conversions.binaryStringToLiteralInteger(tempInString.substring(0, 2));
                opCode = Conversions.binaryStringToLiteralInteger(tempInString.substring(2, 8));

                switch(instructionType)
                {
                        //cases of 00, 01, 10, 11, and a base
                }
```

Figure 3.1.H -- Code snippet for the Execute module. One case included

```
int opCode = instruction;

                switch(opCode)
                {
                // read
                case 0:
                {
                        if (reg2 > 0)
                        {
                                registerSpace[reg1] =
Conversions.hexToDecimal(cache[registerSpace[reg2]/4].substring(2));
                        }
                        else
                        {
                                registerSpace[reg1] =
Conversions.hexToDecimal(cache[tempAddress/4].substring(2));
                        }
                        break;
                }
```

In order to fetch an instruction from memory, the CPU accesses invokes the Memory module and provides a virtual address.

Figure 3.1.I -- Code snippet for the Memory module.

```
public MMU() {
                ram = new RAM();
                frameTable = new Hashtable<Integer, List<Frame>>(NUM_PAGES);
                freeFrames = new LinkedList<>();
                for(int lcv = 0; lcv < NUM_PAGES; lcv++)
                        freeFrames.add(lcv);
        }
```

The Memory module invokes the Effective Address module to translate the virtual address given to it into a physical address.

Figure 3.1.J -- Code snippet for the Effective Address module.

```
        private int calcEffectiveAddress(int virtualAddress, PCB pcb) {
                // search through all frames at the hashed page index for the one
                // with the correct process id:
                List<Frame> frames = frameTable.get(calcPageNumber(virtualAddress));
                if(frames == null) // if no frames at that index
                        return -1;

                int physAddr = -1;
                for(Frame f : frames)
                        if(f.processID == pcb.getJobID()) {
                                physAddr = f.frameNumber * PAGE_SIZE;
                                break;
                        }

                // if no frame found for the process in the given virtual address:
                if(physAddr == -1)
                        return -1;

                physAddr += virtualAddress % PAGE_SIZE;
                return physAddr;
        }
```

The Memory module then uses the physical address to read or write to RAM in the appropriate location.

Figure 3.1.K -- Code snippet for the RAM module.

```
public String read(int physicalAddress) {
                return contents[physicalAddress];
        }

        public void write(int physicalAddress, String word) {
                contents[physicalAddress] = word;
}
```

3.2 Part 2
For part 2, the simulation runs largely the same as in part 1, however there are some differences.

First, the virtual machine has multiple processors available. To facilitate this, the Dispatcher is upgraded to asymmetrically distribute ready processes to idle processors. This Dispatcher, when invoked checks all processors to see if they are idle and assigns different ready processes to each idle processor.

Additionally, each processor is assigned a portion of the RAM space for its own use. Any processor may only use the virtual space assigned to it, and attempts to draw from other processors' spaces will force the job on that processor to terminate. Further, in order to access memory without causing read/write issues, a processor must acquire a lock on memory, meaning that only one processor may access RAM at a time.

Lastly, each processor possess a local cache, which is filled when a job is moved onto the processor with all of the data and instructions of the job.

4. Data

4.1 Part 1
We did not finish the project in time to collect all of our data due to things not being fully integrated just yet.

4.1.1 Data for FIFO Scheduling
We did not finish the project in time to collect all of our data due to things not being fully integrated just yet.

4.1.2 Data for Non-Preemptive Priority Scheduling
We did not finish the project in time to collect all of our data due to things not being fully integrated just yet.

4.2 Part 2
We did not finish the project in time to collect all of our data due to things not being fully integrated just yet.

4.2.1 Data for FIFO Scheduling

We did not finish the project in time to collect all of our data due to things not being fully integrated just yet.

4.2.2 Data for Non-Preemptive Priority Scheduling

We did not finish the project in time to collect all of our data due to things not being fully integrated just yet.

5. Conclusions

As we were not able to finish integrating everything in time, our data is inconclusive, so we do not have any concrete findings. Most of the pieces are coming together, so the system should be able to cooperate with each other soon enough.