Semester Project - Specification
1-CPU is Part 1,
4-CPUs is Part 2
CS 3502 – Operating Systems

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

The simulator will run on a 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 to complete the OS simulator.

The following sections describe the components of the simulator and its virtual machine environments.

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.

The Driver:
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;
        }
```

The 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
}
```

The 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.

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.

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!).

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;
```

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.

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.

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.

I. The CPU (Part 1)

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
}
```

ComputeOnly

This method 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
```

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
}
```

Queues and other code

Need – queues (new, ready, block/waiting, suspend)
Need – conversion routines or use built-in calls between hex, decimal, binary types (in Java or other).

II. Multiprocessor Architecture (Part 2)

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 program cache of the CPU by the short-term loader. 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* 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.

WHAT TO DO:

Part 1

In Part 1, the following tasks are to be performed:

1. Design and implement all the modules, and any others you find necessary, of the Simulator.
2. Execute the set of given processes (30) and measure the following:
   a. Waiting and completion times for each job; and the averages of both
   b. Number of I/O operations each process made
   c. Percentage of RAM space used and cache used
   d. Measure and compare the above performance metrics for FIFO and Priority (non-preemptive) scheduling policies

WHAT TO DO:

Part 2

In Part 2, the following tasks are to be performed:

1. Add a program cache to the CPU module and compile it. And when the scheduler assigns a process to the CPU, the whole process must be loaded into the program cache of the CPU from the RAM. Run this single CPU as a single thread (a uniprocessor system).
2. Construct four (4) clones of the CPU and run them concurrently as threads. The threads **share** certain resources (*memory* and ready queue) which are maintained by the system. Each clone has its own/private cache. Any available thread or CPU can be used to run any processes in the queue.
3. Modify the scheduler and the dispatcher by extending them respectively into m-scheduler and m-dispatcher (for the multiprocessor system). Use asymmetric algorithm for scheduling – you need to rethink your design of the scheduling and dispatching logic for the multiprocessing case.
4. Execute the set of given processes (30): [Note: Each scheduled process must be run from start to finish on the designated CPU – no blocking or interruption in Phase 1]

WHAT TO MEASURE:

In Part 2, the following metrics are to be discussed:

a. Waiting and completion times for jobs assigned to each CPU; and the averages of both
b. Number of I/O each process made
c. Which jobs and percentages of jobs assigned to each CPU; RAM space used and cache used per CPU
d. Measure, compare, and discuss the performance metrics for 1-CPU and N-CPU runs

Turn in a summary report, along with data tables and graphs (if any), the code (on a pen-drive). The tracing, or verification, of the correctness of your Simulator will be done or presented at the demo time. Again, the report should describe your observations, findings, and conclusions.