HCMC University Of Technology
Faculty of Computer Science & Engineering

Course: Operating Systems
Assignment - Simple Operating System

October 24, 2022

Goal: The objective of this assignment is the simulation of major components in a simple operating system, for example, scheduler, synchronization, related operations of physical memory and virtual memory.

Content: In detail, student will practice with three major modules: scheduler, synchronization, mechanism of memory allocation from virtual-to-physical memory.

- scheduler
- synchronization
- the operations of mem-allocation from virtual-to-physical

Result: After this assignment, student can understand partly the principle of a simple OS. They can draw the role and meaning of key modules in the OS as well as how it works.

Contents

1 Introduction

1.1 An overview

The assignment is about simulating a simple operating system to help student understand the fundamental concepts of scheduling, synchronization and memory management. Figure 1 shows the overall architecture of the "operating system" we are going to implement. Generally, the OS has to manage two "virtual" resources: CPU(s) and RAM using two core components:

- Scheduler (and Dispatcher): determines with process is allowed to run on which CPU.
- Virtual memory engine (VME): isolates the memory space of each process from other. The physical RAM is shared by multiple processes but each process do not know the existence of other. This is done by letting each process has its own virtual memory space and the Virtual memory engine will map and translate the virtual addresses provided by processes to corresponding physical addresses.

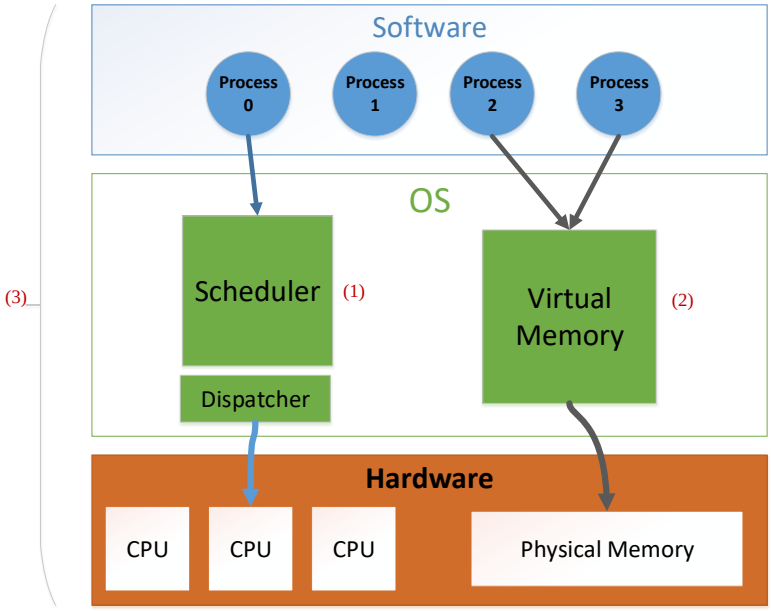

Figure 1: The general view of key modules in this assignment

Through those modules, The OS allows mutliple processes created by users to share and use the "virtual" computing resources. Therefore, in this assignment, we focus on implementing scheduler/dispatcher and virtual memory engine.

1.2 Source Code

After downloading the source code of the assignment in the "Resource" section on the portal platform and extracting it, you will see the source code organized as follows.

- Header files
  - `timer.h`: Define the timer for the whole system.
  - `cpu.h`: Define functions used to implement the virtual CPU

  - `queue.h`: Functions used to implement queue which holds the PCB of processes
  - `sched.h`: Define functions used by the scheduler
  - `mem.h`: Functions used by Virtual Memory Engine.
  - `loader.h`: Functions used by the loader which load the program from disk to memory
  - `common.h`: Define structs and functions used everywhere in the OS.
- Source files
  - `timer.c`: Implement the timer.
  - `cpu.c`: Implement the virtual CPU.
  - `queue.c`: Implement operations on (priority) queues.
  - `paging.c`: Use to check the functionality of Virtual Memory Engine.
  - `os.c`: The whole OS starts running from this file.
  - `loader.c`: Implement the loader
  - `sched.c`: Implement the scheduler
  - `mem.c`: Implement RAM and Virtual Memory Engine
- `Makefile`
- `input` Samples input used for verification
- `output` Samples output of the operating system.

1.3 Processes

We are going to build a multitasking OS which lets multiple processes run simultaneously so it is worth to spend some space explaining the organization of processes. The OS manages processes through their PCB described as follows:

```
// From include/common.h
struct pcb_t {
     uint32_t pid;
     uint32_t priority;
     uint32_t code_seg_t * code;
     addr_t regs;
     uint32_t pc;
     struct seg_table_t * seg_table;
     uint32_t bp;
}
```

The meaning of fields in the struct:

- `PID`: Process's PID
- `priority`: Process priority, Scheduler will let processes with higher priority run before the one with lower priority.
- `code`: Text segment of the process (To simplify the simulation, we do not put the text segment in RAM).
- `regs`: Registers, each process could use up to 10 registers numbered from 0 to 9.

- `pc`: The current position of program counter.
- `seg_table`: Page table used to translate virtual addresses to physical addresses.
- `bp`: Break pointer, use to manage the heap segment.

Similar to the real process, each process in this simulation is just a list of instructions executed by the CPU one by one from the beginning to the end (we do not implement jump instructions here). There are five instructions a process could perform:

- `CALC`: do some calculation using the CPU. This instruction does not have argument.
- `ALLOC`: Allocate some chunk of bytes on the main memory (RAM). Instruction's syntax:

```
alloc [size] [reg]
```

  where `size` is the number of bytes the process want to allocate from RAM and `reg` is the number of register which will save the address of the first byte of the allocated memory region. For example, the instruction `alloc 124 7` will allocate 124 bytes from the OS and the address of the first of those 124 bytes with be stored at register #7.

- `FREE` Free allocated memory. Syntax:

```
free [reg]
```

  where `reg` is the number of the register which holds the address of the first byte of the memory region to be deallocated.

- `READ` Read a byte from memory. Syntax:

```
read [source] [offset] [destination]
```

  The instruction reads one byte memory at the address which equal to the value of register `source` + `offset` and saves it to `destination`. For example, assume that the value of register #1 is `0x123` then the instruction `read 1 20 2` will read one byte memory at the address of `0x123 + 14` (14 is 20 in hexadecimal) and save it to register #2.

- `WRITE` Write a value register to memory. Syntax:

```
write [data] [destination] [offset]
```

  The instruction writes `data` to the address which equal to the value of register `destination` + `offset`. For example, assume that the value of register #1 is `0x123` then the instruction `write 10 1 20` will write 10 to the memory at the address of `0x123 + 14` (14 is 20 in hexadecimal).

1.4 How to Create a Process?

The content of each process is actually a copy of a program stored on disk. Thus to create a process, we must first generate the program which describes its content. A program is defined by a single file with the following format:

```
[priority] [N = number of instructions]
instruction 0
instruction 1
...
instruction N-1
```

where `priority` is the **default** priority of the process created from this program. The higher priority, the higher chance that process is picked up by the CPU from the queue (See section 2.1 for more detail). Please remember that this default value can be overwrite by the "live" priority during process execution calling. For resolving the conflict, when it has priority in process loading, it will overwrite (aka. is the chosen) and replace the default priority in process description file.
`N` is the number of instructions and each of the next `N` lines(s) are instructions represented in the format mentioned in the previous section. You could open files in `input/proc` directory to see some sample programs.

1.5 How to Run the Simulation

What we are going to do in this assignment is to implement a simple OS and simulate it over virtual hardware. To start the simulation process, we must first describe the configuration of the hardware and the environment that we will simulate. This is done by creating a configure file in following format

```
[time slice] [N = Number of CPU] [M = Number of Processes to be run]
[time 0] [path 0] [priority 0]
[time 1] [path 1] [priority 1]
...
[time M-1] [path M-1] [priority M-1]
```

where `time slice` is the period of time for which a process is allowed to run. `N` is the number of CPUs available and `M` is the number of processes to be run. The last parameter `priority` is the "live" priority when the process is invoked and this will overwrite the default priority in process desription file (refers section1.4). In each one of the next `M` lines are the time at which a process is started and the path to the file holding the content of the program to be loaded. You could find configure files at `input` directory.

To start the simulation, you must compile the source code first by using `Make all` command. After that, run the command

```
./os [configure path]
```

where `configure path` is the path to configure file for the environment on which you want to run.

2 Implementation

2.1 Scheduler

We first implement the scheduler. Figure 2 shows how the operating system schedule processes. Although the OS is designed to work on multiple processors, in this part of the assignment, we assume the system has many processors. The OS uses **ready_queue system** to determine which process to be executed when a CPU becomes available. The scheduler design is based on "multilevel feedback queue" algorithm used in Linux kernel.

According to figure 2, the scheduler works as follows. For each new program, the loader will create a new process and assign a new PCB to it. The loader then reads and copies the content of the program to the text segment of the new process (pointed by `code` pointer in the PCB of the process - section 1.3). Finally, the PCB of the process is pushed to `ready_queue` and waits for the CPU. The CPU runs processes in round-robin style. Each process is allowed to run upto a given period of time. After that, the CPU is forced to enqueue the process to the appropriate `priority ready_queue`. The CPU then picks up another process from `ready_queue` and continue running. The rest of the flow is belong to CPU management code to execute it in `ready_queue`.

In this system, we implement the Multi-level Queue (MLQ) policy. The system contains `MAX_PRIO` priority levels. Although the real system, i.e. Linux kernel, may group these level into subset, we keep the design where each priority is held by one `ready_queue` for simplicity. We simplify the add_queue and `put_proc` as putting the proc to appropriated ready queue by priority matching. The main design is belong to the MLQ policy deployed by `get_proc` to fetch a proc to deliver to a CPU.

The description of **MLQ policy**: the traverse step of `ready_queue list` is a fixed formulated number based on the priority, i.e. slot= (MAX_PRIO - prio)
An example in Linux MAX_PRIO=140, prio=0..(MAX_PRIO - 1)

```
prio = 0        |         1          | ....    | MAX_PRIO - 1
slot = MAX_PRIO |   MAX_PRIO - 1     | ....    |      1
```

MLQ policy only goes through the fixed step to traverse all the queue in the priority `ready_queue` list. Your job in this part is to implement this algorithm by completing the following functions

- `enqueue() and dequeue()` (in `queue.c`): We have defined a struct (`queue_t`) for a priority queue at `queue.h`. Your task is to implement those functions to help put a new PCB to the queue and get the next 'in turn' PCB out of the queue.
- `get_proc()` (in `sched.c`): gets PCB of a process waiting from the `ready_queue` system. The selected `ready_queue` 'in turn' has been described in the above policy.

To check your work, compile the source code using Makefile

```
make sched
```

and then run the scheduler using sample configures with

```
make test_sched
```

You could compare your result with model answers in `output` directory. Note that because the loader and the scheduler run concurrently, there may be more than one correct answer for each test.

Question: What is the advantage of using priority queue in comparison with other scheduling algorithms you have learned?

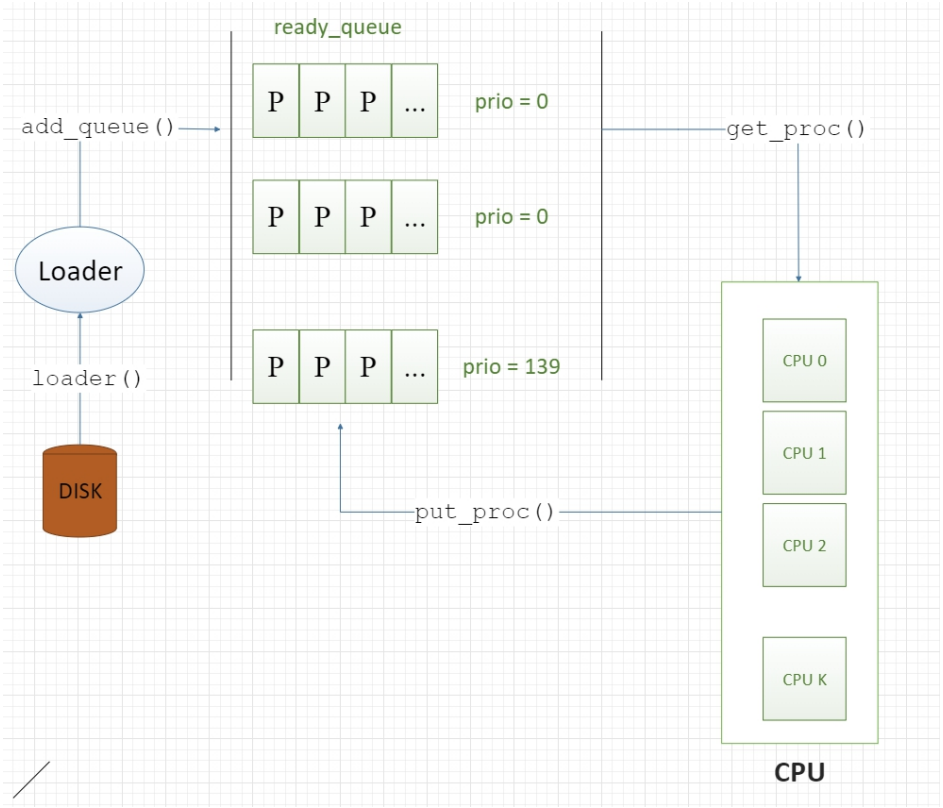

Figure 2: The operation of scheduler in the assignment

2.2 Memory Management

The virtual memory engine uses Segmentation with Paging mechanism to manage the memory. By default, the size of virtual RAM is 1 MB so we must use 20 bit to represent the address of each of its byte. With the segmentation with paging mechanism, we use the first 5 bits for segment index, the next 5 bits for page index and the last 10 bits for offset. You may want to review lecture note chapter 8 (page #38 - #39) to understand how this mechanism work. After that, you could implement the process of translating virtual address of a process to physical one by completing the following functions:

- `get_page_table()` (in `mem.c`): Find the page table given a segment index of a process.
- `translate()` (in `mem.c`): uses `get_page_table()` function to translate a virtual address to physical address.

Note that functions mentioned above are applied for processes which have already had segment table and page table for us to do our job. The main problem is how to construct those tables. Obviously, we cannot map the whole virtual address space of a process to RAM's address space because RAM is shared by multiple processes and memory regions used by one process should not be used by other (ensure isolation). Therefore, we should start the process with empty segment table and keep updating it every time the process allocate or deallocate memory.

To simplify the implementation of memory allocation and deallocation, the OS maintains a special structure called `_mem_stat` which tracks the status of physical pages. Particularly, RAM is split into multiple pages according to Segmentation with Paging mechanism and the OS allocates memory to processes by pages. That is, if the process requires an amount of memory that less than the page size, it sill receives a whole new

page. The responsibility of _mem_stat is to maintain the status (used/free) of those pages. Particularly, _mem_stat is a table in which each row is a struct defined as follows

```
struct {
    uint32_t proc;
    uint32_t index;
    uint32_t next;
}
```

where proc is the PID of the process which is currently use this page. If proc = 0, the page is free and the OS could allocated it to any process. Process may require a memory region whose size is much larger than the size of a single page. In this case, the OS must allocate multiple pages to this process. Since the process uses virtual address to access the content of RAM, we must let the virtual address of allocated pages contiguous but their physical addresses are not need to be. For example, support a process requires 2 pages and we have exactly 2 free pages in RAM but those page are not contiguous: the address of their first byte are 0x00000 and 0x01000, respectively. In the virtual address space, we create two contiguous pages whose first byte are 0x00000 and 0x00020 (offset lasts 5 bits) and map page 0x00000 to physical page 0x00000, page 0x00020 to physical page 0x01000. Although process feels that the memory regions it has allocated is contiguous, they are actually not. Since we use this method for memory allocation, a row in _mem_stat has index field to know let us know its location in the list of allocated page. The next field is used to indicate the index in _mem_stat of next page in the list of allocated pages. If the page is the last page, this field is set to -1. In the previous example, the page 0x00000 has index = 0 and that of the page 0x01000 would be 1 while the value of their next are 0x80 and -1 respectively.
Figure 3 shows how we allocate new memory regions and create new entry in the segment and page tables inside a process. Particularly, for each new page we allocated, we must add new entry to page tables according to this page's segment number and page number.
Your main task is to implement functions alloc_mem() and free_mem both are in mem.c based on the algorithm described above. To verify your implementation, you must first compile the code using Makefile:

```
make mem
```

and then run the test on sample configuration

```
make test_mem
```

You could compare your results with model answers in output directory. Your result must be identical to the answer.
Question What is the advantage and disadvantage of segmentation with paging.

2.3 Put It All Together

Finally, we combine scheduler and Virtual Memory Engine to form a complete OS. Figure 4 shows the complete organization of the OS. The last task to do is synchronization. Since the OS runs on multiple processors, it is possible that share resources could be concurrently accessed by more than one process at a time. Your job in this section is to find share resource and use lock mechanism to protect them.
Check your work by fist compiling the whole source code

```
make all
```

and then check your results with model answers by running the following command

```
make test_all
```

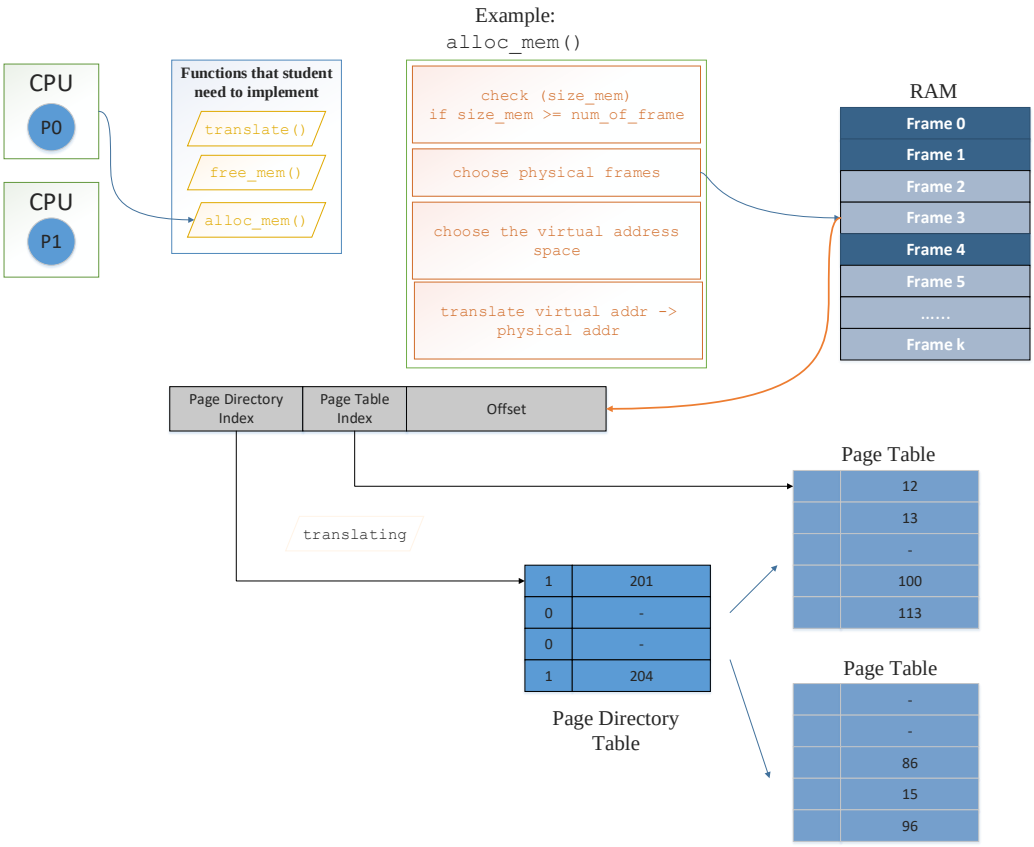

Figure 3: The operation related to virtual memory in the assignment

and compare your output with those in `output`. Remember that as we are running multiple processes, there may be more than one correct result.

Question: What will happen if the synchronization is not handled in your simple OS? Illustrate by example the problem of your simple OS if you have any.

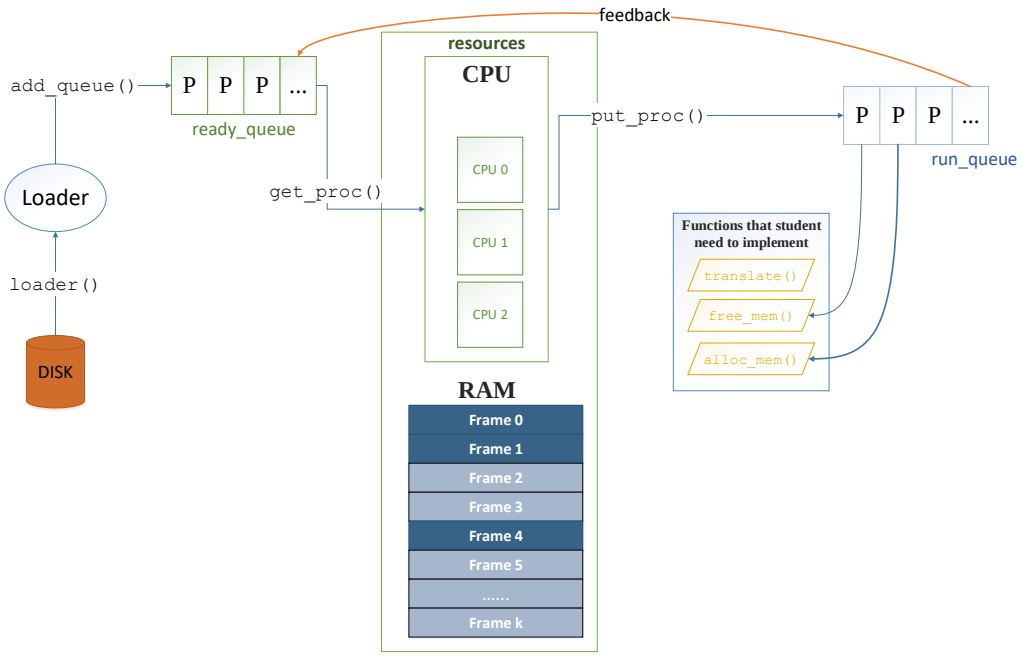

Figure 4: The model of assignment 2

3 Submission

3.1 Source code

Requirement: you have to code the system call followed by the coding style. Reference: https://www.gnu.org/prep/standards/html_node/Writing-C.html.

3.2 Report

Write a short report that answer questions in implementation section and interpret the results of running tests in each section:

- Scheduling: draw Gantt diagram describing how processes are executed by the CPU.
- Memory: Show the status of RAM after each memory allocation and deallocation function call.
- Overall: student find their own way to interpret the results of simulation.

After you finish the assignment, moving your report to source code directory and compress the whole directory into a single file name `assignment1_MSSV.zip` and submit to Sakai.

3.3 Grading

You must carry out this assignment by groups of three or four. The overall grade for each student is a combination of two parts

- Group: how well your group carry out the assignment (6 points)
  - Scheduling (1.5 points)
  - Memory (1.5 points)
  - Synchronization (1.5 points)
  - Report (1.5 points)
- Individual: how much you contribute to the group and how well you understand the assignment (4 points)