HCMC University Of Technology
Faculty of Computer Science & Engineering

Course: Operating Systems
Assignment #2 - Simple Operating System

March 20, 2023

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.

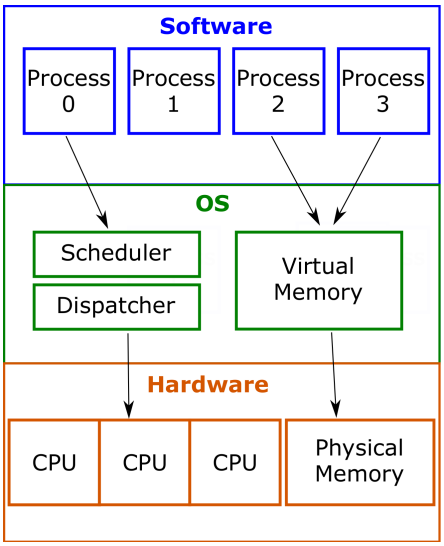

Figure 1: The general view of key modules in this assignment

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.
  - `bitopts.h`: Define operations on bit data.
  - `os-mm.h, mm.h`: Define the structure and basic data for Paging-based Memory Management.
  - `os-cfg.h`: (Optionaal) Define contants use to switch the software configuration.

- 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
  - `mm.c, mm-vm.c, mm-memphy.c`: Implement Paging-based Memory Management
- `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;
#ifdef MLQ_SCHED
     uint32_t prio;
#endif
}
```

The meaning of fields in the struct:

- `PID`: Process's PID
- `priority`: Process priority, the lower value the higher priority the process has. This legacy priority depend on the process's properties and is fixed over execution session.

- `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.
- `prio`: Priority on execution (if supported), and this vale overwrites the default priority when it is existed.

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.
- ***Annotation of Memory region****: A storage area where we allocated for a variable, this term is actually associated with an index of SYMBOL TABLE and usually supports human-readable through variable name and a mapping. Unfortunately, this mapping is out-of-scope of this Operating System course and maybe belong to other couse which explain how to the compiler do it job and map the label to it associated index. For simplicity, we refer here a memory region through its index and it has a limit on the number of variable in each program/process.*
- `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. It needs to remind that this system employs a dual priority mechanism.
The higher priority (with the smaller value), the higher chance that process is picked up by the CPU from the queue (See section 2.1 for more detail). `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.
Dual priority mechanism 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 (this inputt file), it will overwrite and replace the default priority in process description file.

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 create a description file in `input` directory about the hardware and the environment that we will simulate. The description file is defined in the 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 section 1.4).

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.

Again, it's worth to remind that this system equips a **dual priority mechanism**. If you don't have the default priority than we don't have enough the material to resolve the conflict during the scheduling procedure. But, if this value is fixed, it limits the algorithms that the simulation can illustrate the theory. Verify with your real-life environment, there is different priority systems, one is about the system program vs user program while the other also allows you to change the "live" priority.

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

```
./os [configure_file]
```

where `configure_file` is the path to configure file for the environment on which you want to run and it should associated with the name of a description file placed in `input` directory.

2 Implementation

2.1 Scheduler

We first implement the scheduler. Figure 2 shows how the operating system schedule processes. The OS is designed to work on multiple processors. The OS uses multiple-queue called **ready_queue** to determine which process to be executed when a CPU becomes available. Each queue is associated with a fixed priority value. The scheduler is designed based on "multilevel queue" algorithm used in Linux kernel[1].

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). The PCB of the process is pushed to the associated `ready_queue`, the queue with priority equal is `prio` value in OS configuration file define when it load a new process. Then, it waits for the CPU. The CPU runs processes in round-robin style. Each process is allowed to run up to a given period of time. After that, the CPU is forced to enqueue the process back to it associated `priority ready_queue`. The CPU then picks up another process from `ready_queue` and continue running.

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 traversed step of `ready_queue list` is a fixed formulated number based on the priority, i.e. slot= (MAX_PRIO - prio), each queue have only fixed slot to use the CPU and when it is used up, the system must change the resource to the other process in the next queue and left the remaining work for future slot eventhough it needs a completed round of ready_ queue.
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
```

[1]Actually, Linux supports the feedback mechanism which allow to move process among priority queues but we don't the implement feedback mechanism here

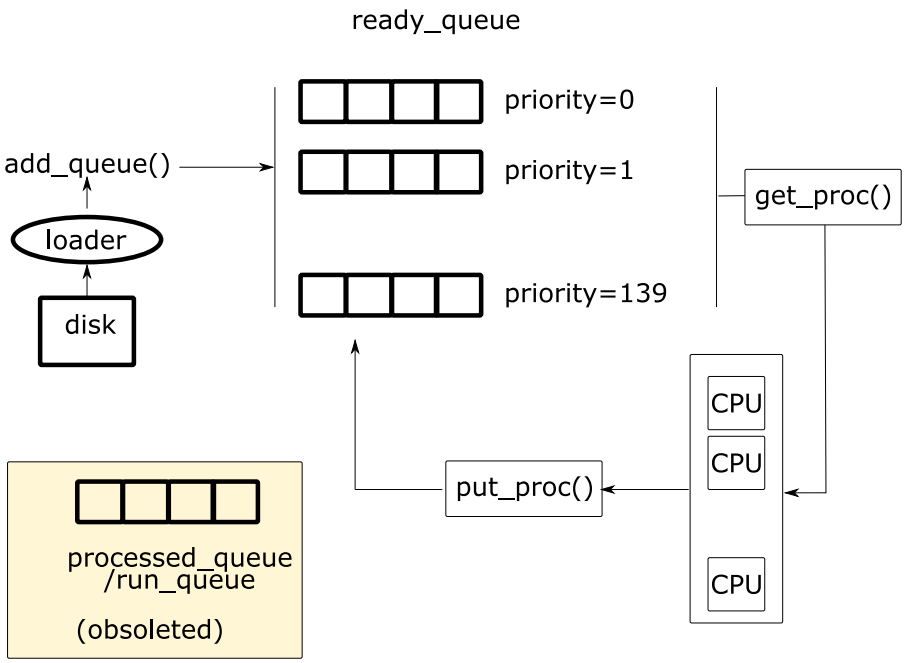

Figure 2: The operation of scheduler in the assignment

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?

2.2 Memory Management

2.2.1 The virtual memory mapping in each process

The virtual memory space is organized as a memory mapping for each process PCB. From the process point of view, the virtual address includes multiple `vm_area` contiguous memory area. In the real world, each area can act as code, stack or heap segmentation. Therefore, the process keeps in its pcb an pointer of multiple contiguous memory areas.

Memory Area Each memory area ranges continuously in `[vm_start,vm_end]`. Although the space spans the whole range, the actual usable area is limited by the top pointing at `sbrk`. In the area between vm_start and sbrk, there are multiple regions captured by `struct vm_rg_struct` and free slots tracking by the list `vm_freerg_list`. Through this design, we make the design to perform the real allocation of physical memory only in the usable area as in Figure 3.

```
/*
 *  Memory region struct
 */
struct vm_rg_struct {
    unsigned long rg_start;
    unsigned long rg_end;
```

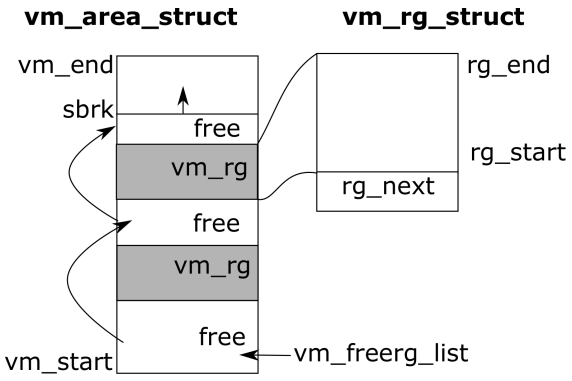

Figure 3: The structure of vm area and region

```
    struct vm_rg_struct *rg_next;
};

/*
 *  Memory area struct
 */
struct vm_area_struct {
    unsigned long vm_id;
    unsigned long vm_start;
    unsigned long vm_end;

    unsigned long sbrk;
/*
 * Derived field
 * unsigned long vm_limit = vm_end - vm_start
 */
    struct mm_struct *vm_mm;
    struct vm_rg_struct *vm_freerg_list;
    struct vm_area_struct *vm_next;
};
```

Memory region As we noted in the previous section 1.3, these regions are actually acted as the variables in human-readable program's source code. Due to current out-of-scope fact, we simply touch in the concept of namespace in term of indexing. We have not equipped enough the principle of the compiler and it is, again, employs the work on such a complex symbol table in this OS course. We temporarily imagine these regions are a set of limit number of regions and are managed using an array of symrgtbl[PAGING_MAX_SYMTBL_SZ]. The size of the array is fixed by a constant denoted the number of variable allowed in each program. To wrap up, we use the `struct vm_rg_struct symrgtbl` to keep the start and the end point of the region and a pointer `rg_next` is reserved for future keep tracking of a set of regions.

```
/*
 * Memory management struct
 */
struct mm_struct {
    uint32_t *pgd;
```

```
    struct vm_area_struct *mmap;

    /* Currently we support a fixed number of symbol */
    struct vm_rg_struct symrgtbl[PAGING_MAX_SYMTBL_SZ];

    struct pgn_t *fifo_pgn;
};
```

Memory mapping is represented by `struct mm_struct` in which keep a tracking of all the mentioned memory instances in a separated contiguous memory area, and there are many areas point out by `struct vm_arera_struct *mmap` in a list structure. The next important field is the `pgd`, which is the page table directory, contains all page table entries to map between page and frame in paging memory management system. We keep the detailed of page-frame mapping to the later section 2.2.3. The other fields are mostly used to keep track for a specific user operation, i.e. caller, fifo page (for referencing), so we left it there and it can be used by your own or just discard it.

CPU addresses the address generated by CPU to access a specific memory location. In paging-based system, it is divided into:

- **Page number (p)**: used as an index into a page table that holds the based address for each page in physical memory.
- **Page offset (d)**: combined with base address to define the physical memory address that is sent to the Memory Management Unit

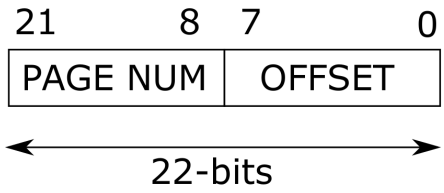

Figure 4: CPU Address

The physical address space of a process can be noncontiguous. We divide physical memory into fixed-sized blocks (the frames) with two sizes 256B or 512B. We proposed various setting combination in Table 1 and end up with the highlighted configuration. This is a referenced settings and can be modified or re-select in other simulations. Based on the setting of 22-bit CPU and 256B page size, we have the CPU address is organized as in Figure 4.

In the VM summary, all structures supporting VM are placed in the module. `mm-vm.c`.
Question: What is the advantage of using the separated memory segmentation design?

CPU bus	PAGE size	PAGE bit	No pg entry	PAGE Entry sz	PAGE TBL	OFFSET bit	PGT mem	MEMPHY	fram bit
20	256B	12	~4000	4byte	16KB	8	2MB	1MB	12
22	256B	14	~16000	4byte	64KB	8	8MB	1MB	12
22	512B	13	~8000	4byte	32KB	9	4MB	1MB	11
22	512B	13	~8000	4byte	32KB	9	4MB	128kB	8
16	512B	8	256	4byte	1kB	9	128K	128kB	4

Table 1: Various CPU address bus configuration value

2.2.2 The system physical memory

Figure 1 shows that the memory hardware is installed in term of the whole system. All processes own their separated memory mappings, but all mappings target a singleton physical devices. There are two kinds of the device which are RAM and SWAP. They both can be implemented by the same physical device as in `mm-memphy.c` with different settings. The supported settings are randomization memory access, sequential/serial memory access, storage capacity.
Despite the possible different settings, the logical usage of these devices can be distinguished. The RAM device which is belonged to the main memory subsystem, can be accessed directly through CPU address bus, or it can be read/write by issuing CPU instruction. Meanwhile, SWAP is just a secondary memory device and all of its stored data manipulation must be performed by moving them to the main memory. Since its lack of direct access to the CPU, the system usually has large SWAP with a small cost and even have more than one instance. In our settings, we support the hardware installed with one RAM device and up to 4 SWAP devices.
The `struct framephy_struct` is mainly used to store the frame number.
The `struct memphy_struct` has basic fields of data of storage and size. The `rdmflg` field defines the memory access is randomly or serially access. The field `free_fp_list` is reserved for retaining the unused memory frame.

```
/*
 * FRAME/MEM PHY struct
 */
struct framephy_struct {
   int fpn;
   struct framephy_struct *fp_next;
};

struct memphy_struct {
   /* Basic field of data and size */
   BYTE *storage;
   int maxsz;

   /* Sequential device fields */
   int rdmflg;
   int cursor;

   /* Management structure */
   struct framephy_struct *free_fp_list;
};
```

Question: What will happen if we divide the address to more than 2-levels in the paging memory management system?

2.2.3 Paging-based address translation scheme

The translation supports both segmentation and segmentation with paging. In this version, we develop a single level paging system which leverages almost one RAM device and one SWAP instance hardware. We are prepared (coded) with the capabilities of multiple memory segments but we still stay with mainly the first and is the only one segment of vm_area with (vmaid = 0). The further versions will take care of the more sufficient paging scheme of many segments or possible overlap/non-overlap among segments.

Page table This structure lets a userspace process find out which physical frame each virtual page is mapped to. It contains one 32-bit value for each virtual page, containing the following data:

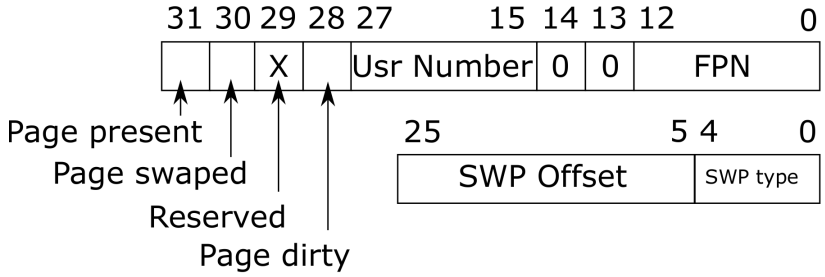

Figure 5: Page Table Entry Format.

```
 * Bits 0-12  page frame number (PFN) if present
 * Bits 13-14 zero if present
 * Bits 15-27 User-defined numbering if present
 * Bits 0-4   swap type if swapped
 * Bits 5-25  swap offset if swapped
 * Bit  28    page is dirty
 * Bits 29    zero
 * Bit  30    page swapped
 * Bit  31    page present
```

The virtual space is isolated to each entity then each `struct pcb_t` has its own table. To work in paging-based memory system, we need to update this struct and the later section will discuss the required modification. In all cases, each process has a completely isolated and unique space, `N` processes in our setting result in `N` page tables and in turn, each page must have all entries for the whole CPU address space. For each entry, the paging number may have an associated frame in MEMRAM or MEMSWP or might have null value, the functionality of each data bit of the page table entry is illustrated in Figure 5. In our chosen highlighted setting in Table 1 we have 16.000-entry table each table cost 64 KB storage space.

In section 2.2.1, the process can access the virtual memory space in a contigous manner of vm area structure. The remained work deals with the mapping between page and frame to provide the contiguous memory space over discreted frame storing mechanism. It falls into the two main approaches of memory swapping and basic memory operations, i.e. alloc/free/read/write, which mostly keep in touch with `pgd` page table structure.

Memory swapping We have been notified that a memory area may not be used up to its limit space then it has the storage space without being mapped. The swapping can help moving the contents of physical frame between the RAM and SWAP. The swapping in mechanism copy content of frame from outside to main memory ram. The swapping out, in reverse direction try to move the content of the frame in RAM to SWAP. In typical context, the swapping help us gain the free frame of RAM since the size of SWAP device is usually large enough.

Basic memory operations in paging-based system

- ALLOC in most case, it fits into available region. If there is no such a suitable space, we need lift up the barrier sbrk and since it have never been touched, it may needs provide some physical frames and then map them using Page Table Entry.
- FREE the storage space associated with the region id. Since we cannot collect back the taken physical frame which might cause memory holes, we just keep the collected storage space in a free list for further alloc request.

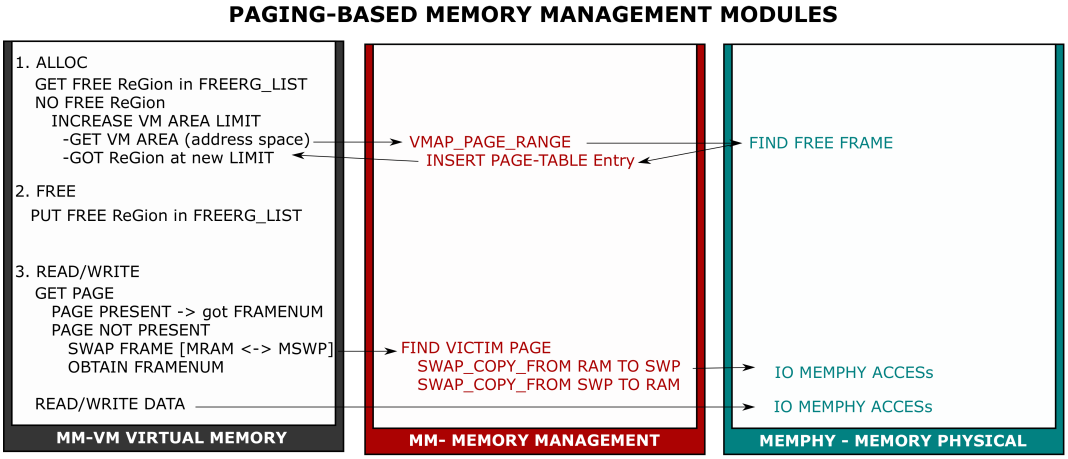

Figure 6: Memory system modules

- READ/WRITE requires to get the page to be presented in the main memory. The most resource consuming step is the page swapping. If the page was in the SWAP device, it needs to bring the page back to RAM device and if it's lack of space, we need to give back some page to SWAP device to make more room.

To perform these operations, it needs a collaboration among the `mm`'s modules as illustrated in Figure 6.
Question What is the advantage and disadvantage of segmentation with paging.

2.2.4 Wrapping-up all paging-oriented implementations

Introduction to the configuration control using constant definition: [2] to make less effort on dealing with the interference among feature-oriented program modules, we apply the same approach with developer community by isolating each feature through a system of configuration. Leveraging this mechanism, we can maintain various subsystems separately existed all in one version of code. We can control the configuration of our simulation program in the `include/os-cfg.h` file

```
// From include/os-cfg.h
#define MLQ_SCHED 1
#define MAX_PRIO 139

#define MM_PAGING
#define MM_FIXED_MEMSZ
```

An example of `MM_PAGING` setting: Therefore, we got a derivation of PCB struct with some additional memory management fields wrapped by a constant definition verification. If we want to use the `MM_PAGING` mode then we enable the associated `#define` config line in `include/os-cfg.h`

```
// From include/common.h
struct pcb_t {
     ...
#ifdef MM_PAGING
     struct mm_struct *mm;
     struct memphy_struct *mram;
```

[2] This section is applied mainly to paging memory management. If you are still working in scheduler section you should keep the default setting and avoid touching too much on these values

```
    struct memphy_struct **mswp;
    struct memphy_struct *active_mswp;
#endif
    ...
};
```

Another example of `MM_FIXED_MEMSZ` setting: Associated with the new verssion of PCB struct, the description file in `input` can keep the old setting with `#define MM_FIXED_MEMSZ` while it still works in paging memory management mode. This mode configuration benefits the backward compatible with old version input file.

New configuration with explicit declaration of memory size (Be careful, new configuration mode meaning that we comment out or delete the constant `#define MM_FIXED_MEMSZ`) If we enable the setting, then the simulation program take 1 more configuration line in the input. This input config is about the system physical memory sizes: MEMRAM and up to 4 MEMSWP. The size value is a non-negative integer value. We can set the size equal 0, but it implies that swap is disabled. To keep this setting valid, we must have a RAM and at least 1 SWAP, these values must be positive integer, the rest values can be set equal 0.

```
[time slice] [N = Number of CPU] [M = Number of Processes to be run]
[MEM_RAM_SZ] [MEM_SWP_SZ_0] [MEM_SWP_SZ_1] [MEM_SWP_SZ_2] [MEM_SWP_SZ_3]
[time 0] [path 0] [priority 0]
[time 1] [path 1] [priority 1]
...
[time M-1] [path M-1] [priority M-1]
```

The highlighted input line is controlled by the constant definition. Double check the input file and the content of `include/os-cfg.h` will help us understand how the simulation program behaves when there may be something strange.

2.3 Put It All Together

Finally, we combine scheduler and Virtual Memory Engine to form a complete OS. Figure 7 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
```

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.

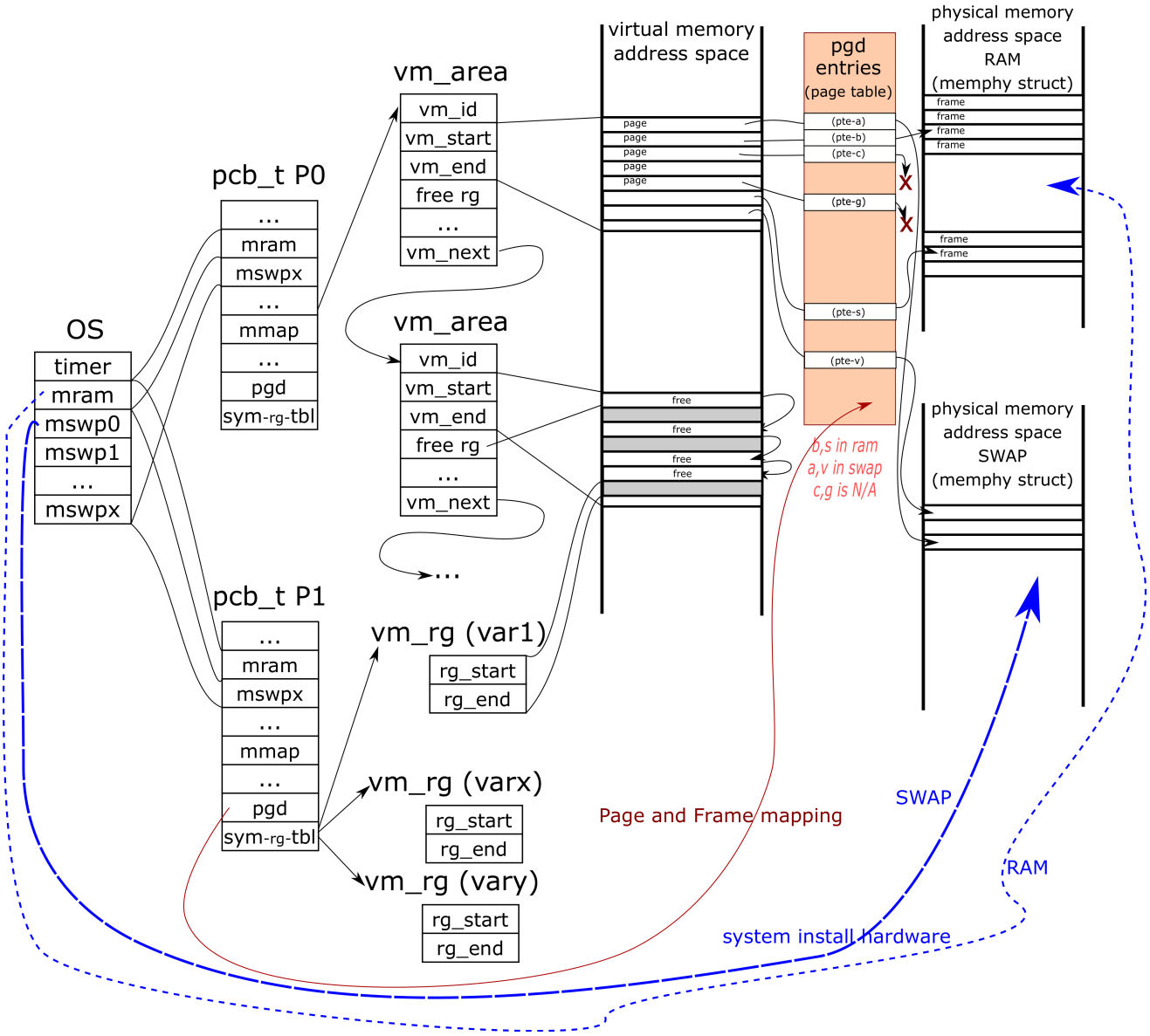

Figure 7: The operation related to virtual memory in the assignment

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 de-allocation 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)