HCMC University Of Technology
Faculty of Computer Science & Engineering

Course: Operating Systems
Assignment - Simple Operating System

April 8, 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 understand and draw the role of OS key modules.

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 which 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.

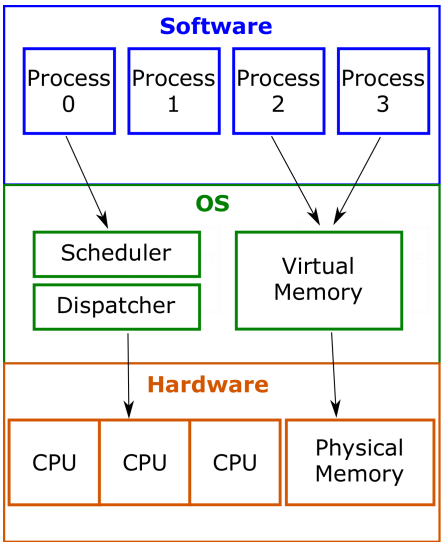

Figure 1: The general view of key modules in this assignment

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`: (Optional) 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[10];
     uint32_t pc;
#ifdef MLQ_SCHED
     uint32_t prio;
#endif
     struct page_table_t * page_table;
     uint32_t bp;
}
```

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.
- `page_table`: The translation from virtual addresses to physical addresses (obsoleted do not use).
- `bp`: Break pointer, use to manage the heap segment.
- `prio`: Priority on execution (if supported), and this value 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 allocate the storage space 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 mechanism. Unfortunately, this mapping is out-of-scope of this Operating System course. It might belong another couse which explains how to the compiler do its job and map the label to its associated index. For simplicity, we refer here a memory region through its index and it has a limit on the number of variables 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 registers holding 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 process has, the process has higher chance to be 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 tackling 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 amount of time (in seconds) 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).

From the second line onward, each line represents the process arrival time , the path to the file holding the content of the program to be loaded and its priority. 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 schedules 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` having the same priority with the value *prio* of this process. Then, it waits for the CPU. The CPU runs processes in round-robin style. Each process is allowed to run in time slice. 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 levels into subsets, 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 and then dispatch 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.

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.

Notice: the `run_queue` is something not compatible with the theory and has been obsoleted for a while. We don't need it in both theory paradigm and code implementation, it is such a legacy/outdated code but

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

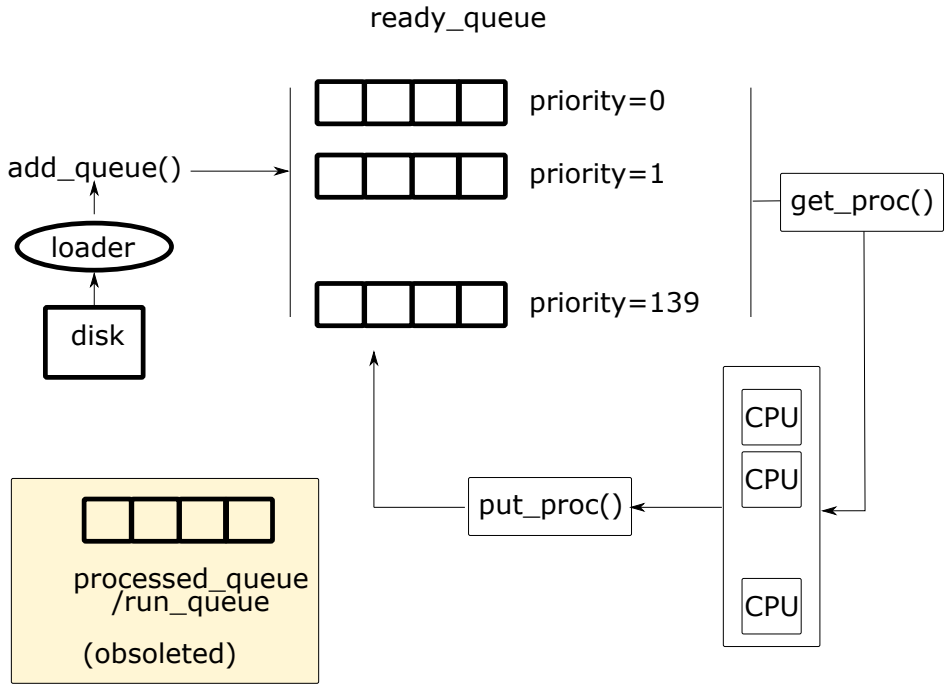

Figure 2: The operation of scheduler in the assignment

we still keep it to avoid bug tracking later.

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_areas` (contiguously). In the real world, each area can act as code, stack or heap segment. Therefore, the process keeps in its pcb a 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 actual allocation of physical memory only in the usable area, as in Figure 3.

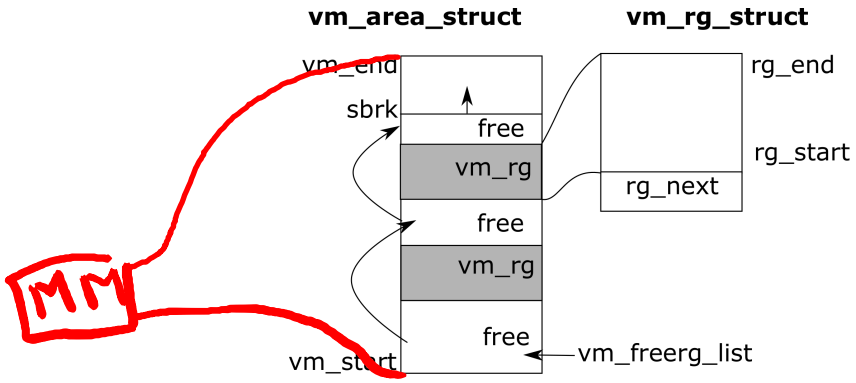

Figure 3: The structure of vm area and region

```c
//From include/os-mm.h
/*
 *  Memory region struct
 */
struct vm_rg_struct {
   unsigned long rg_start;
   unsigned long rg_end;

   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 the human-readable program's source code. Due to the current out-of-scope fact, we simply touch in the concept of namespace in term of indexing. We have not been equipped enough the principle of the compiler. It is, again, overwhelmed to employs such a complex symbol table in this OS course. We temporarily imagine these regions as a set of limit number of region. We manage them by using an array of symrgtbl[PAGING_MAX_SYMTBL_SZ]. The array size is fixed by a constant, PAGING_MAX_SYMTBL_SZ, 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 the pointer `rg_next` is reserved for future

set tracking.

```
//From include/os-mm.h
/*
 * Memory mapping 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`, which keeps tracking of all the mentioned memory regions in a separated contiguous memory area. In each memory mapping struct, many memory areas are pointed out by `struct vm_area_struct *mmap` list. The following important field is the `pgd`, which is the page table directory, contains all page table entries. Each entry is a map between the page number and the frame number in the paging memory management system. We keep detailed page-frame mapping to the later section 2.2.3. The `symrgtbl` is a simple implementation of the symbol table. The other fields are mainly used to keep track of a specific user operation i.e. caller, fifo page (for referencing), so we left it there, and it can be used on your own or just discarded 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

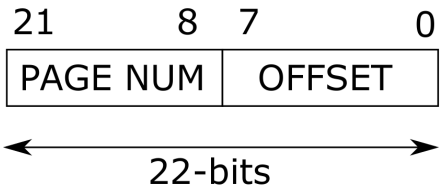

Figure 4: CPU Address

The physical address space of a process can be non-contiguous. We divide physical memory into fixed-sized blocks (the frames) with two sizes 256B or 512B. We proposed various setting combinations in Table 1 and ended up with the highlighted configuration. This is a referenced setting and can be modified or re-selected in other simulations. Based on the configuration of 22-bit CPU and 256B page size, 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: In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments?

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 terms of the whole system. All processes own their separated memory mappings, but all mappings target a singleton physical device. There are two kinds of devices 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, and storage capacity.
In spite of the various possible configurations, the logical use of these devices can be distinguished. The RAM device, belonging to the primary memory subsystem, can be accessed directly from the CPU address bus, i.e., can be read/written with CPU instructions. 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 it lacks direct access from the CPU, the system usually equips a large SWAP at a small cost and even has 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 storage and size. The `rdmflg` field defines the memory access is randomly or serially access. The fields `free_fp_list` and `used_fp_list` are reserved for retaining the unused and the used memory frames, respectively.

```
//From include/os-mm.h
/*
 * 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;
   struct framephy_struct *used_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 that 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 into account the sufficient paging scheme of multiple segments or possible overlap/non-overlap between segments.

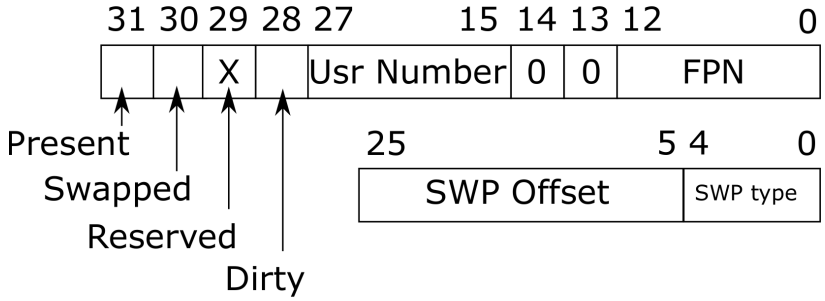

Figure 5: Page Table Entry Format.

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:

```
 * 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    dirty
 * Bits 29    reserved
 * Bit  30    swapped
 * Bit  31    presented
```

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 o We have been informed that a memory area (/segment) may not be used up to its limit storage space. It means that there are the storage spaces which aren't mapped to MEMRAM. The

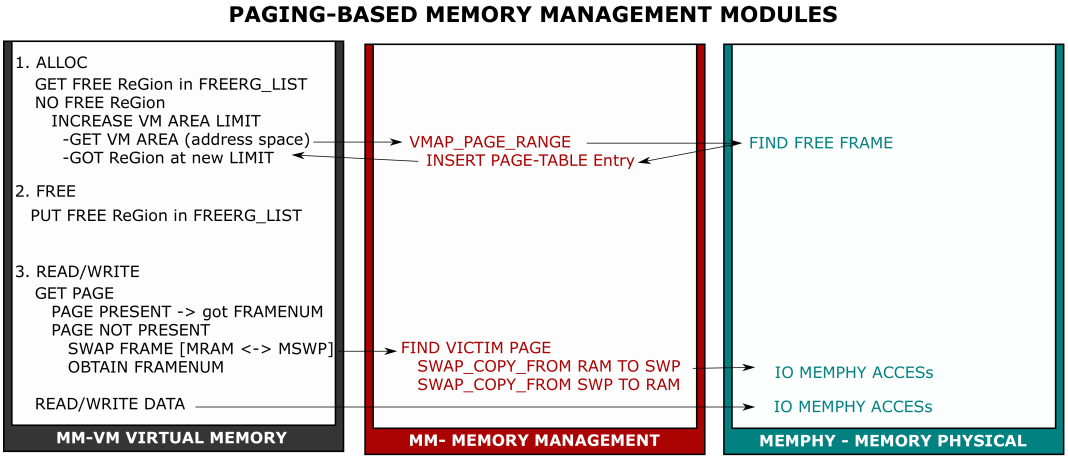

Figure 6: Memory system modules

swapping can help moving the contents of physical frame between the MEMRAM and MEMSWAP. The swapping is the mechanism performs the copying the frame's content from outside to main memory ram. The swapping out, in reverse, tries to move the content of the frame in MEMRAM to MEMSWAP. 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.
- 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 MEMSWAP device, it needs to bring that page back to MEMRAM device (swapping in) and if it's lack of space, we need to give back some pages to MEMSWAP device (swapping out) to make more rooms.

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 in the developer community by isolating each feature through a system of configuration. Leveraging this mechanism, we can maintain various subsystems separately all existed in a single version of code. We can control the configuration used in our simulation program in the `include/os-cfg.h` file

[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

```
// From include/os-cfg.h
#define MLQ_SCHED 1
#define MAX_PRIO 140

#define MM_PAGING
#define MM_FIXED_MEMSZ
```

An example of `MM_PAGING` setting: With this new modules of memory paging, we got a derivation of PCB struct added some additional memory management fields and they are wrapped by a constant definition. If we want to use the `MM_PAGING` module 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;
     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 the new paging memory management mode. This mode configuration benefits the backward compatible with old version input file. Enabling this setting allows the backward compatible.

New configuration with explicit declaration of memory size (Be careful, this mode supports a custom memory size implies that we comment out or delete or disable the constant `#define MM_FIXED_MEMSZ`) If we are in this mode, then the simulation program takes one additional line from the input file. This input line contains the system physical memory sizes: a MEMRAM and up to 4 MEMSWP. The size value requires a non-negative integer value. We can set the size equal 0, but that means the swap is deactivated. To keep a valid parameter, we must have a MEMRAM and at least 1 MEMSWAP, those values must be positive integers, the remaining values can be set to 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 contents 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 memory management to form a complete OS. Figure 7 shows the complete organization of the OS memory management. 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.

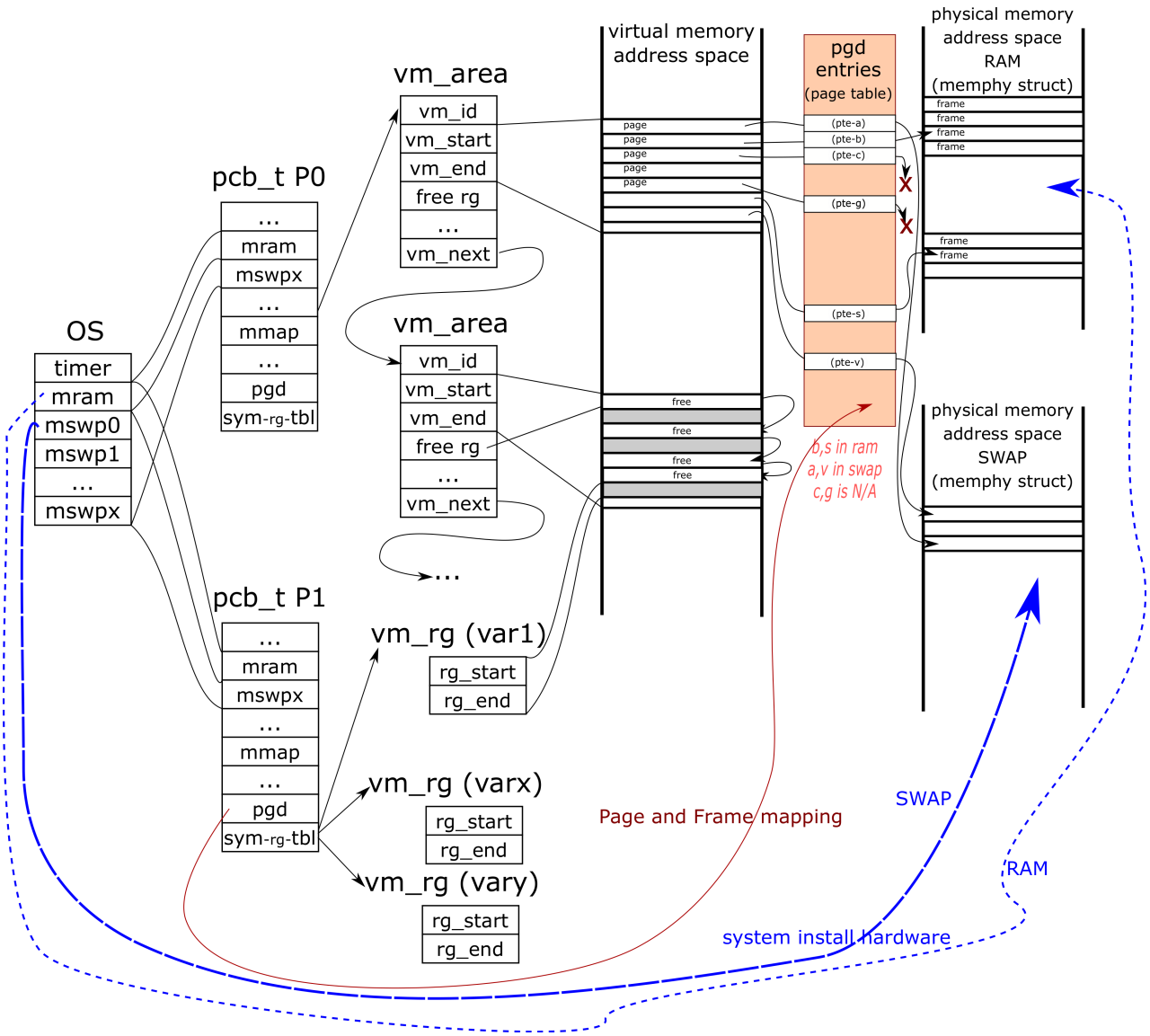

Figure 7: The operation related to virtual memory in the assignment

Check your work by fist compiling the whole source code

```
make 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. All the outputs are used as samples and is not the restricted reults. Your results need to be explained and be compared with theorical framework.

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.

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 `assignment_MSSV.zip` and submit to BKEL.

3.3 Grading

You must carry out this assignment by groups of 2 or 3 students. The overall grade your group is a combination of two parts:

- Demonstration (6 points)
- Report (4 points)