HCMC University Of Technology
Faculty of Computer Science & Engineering

# Course: Operating Systems
# Assignment - Simple Operating System

April 1, 2025

**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

Besides, student will practice the design and implementation of Simple Operating System programming interface via system call.

**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** **3**
- 1.1 An overview . . . 3
- 1.2 Source Code . . . 3
- 1.3 Processes . . . 4
- 1.4 How to Create a Process? . . . 6
- 1.5 How to Run the Simulation . . . 6

**2 Implementation** **7**
- 2.1 Scheduler . . . 7
- 2.2 Extra task for Honored Program - Implementing Completely Fair Scheduler in a Simple OS . 8
- 2.3 Memory Management . . . 9
  - 2.3.1 The virtual memory mapping in each process . . . 9
  - 2.3.2 The system's physical memory . . . 12
  - 2.3.3 Paging-based address translation scheme . . . 13
  - 2.3.4 Wrapping-up all paging-oriented implementations . . . 15
- 2.4 System Call . . . 16
  - 2.4.1 Description . . . 16
  - 2.4.2 Adding a system call to Simple Operating System . . . 17
- 2.5 Put It All Together . . . 19

**3 Submission** **21**
- 3.1 Source code . . . 21
- 3.2 Requirements . . . 21
- 3.3 Report . . . 21
- 3.4 Grading . . . 21
- 3.5 Code of ethics . . . 21

# 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: 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.

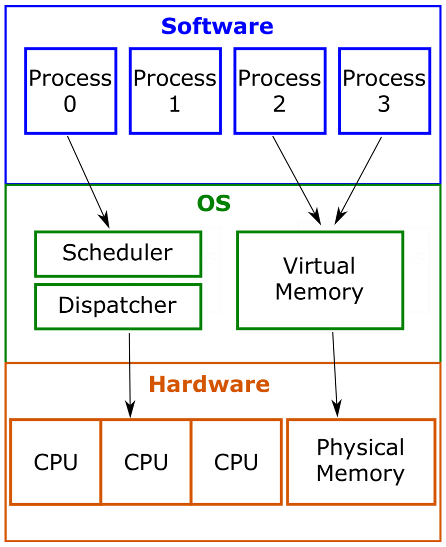

Figure 1: The general view of key modules in this assignment

Through those modules, the OS allows multi-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`: define functions used to implement queue which holds the PCB of processes.
  - `sched.h`: define functions used by the scheduler
  - `mem.h`: define unctions used by Virtual Memory Engine.

  - `loader.h`: (obsoleted) define 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 the constants used 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`: (obsoleted) use to check the functionality of Virtual Memory Engine.
  - `os.c`: contain the main function to start the whole OS system.
  - `loader.c`: implement the loader
  - `sched.c`: implement the scheduler
  - `mem.c`: (obsoleted) implement the previous obsoleted version RAM and Virtual Memory.
  - `mm.c, mm-vm.c, mm-memphy.c`: implement Paging-based Memory Management
- `Makefile`
- `input` the folder contains a set of inputs used for verification
- `output` sample outputs of the system.

## 1.3 Processes

We are going to build a multitasking OS which lets multiple processes run concurrently 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;
     char path[100];
     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.

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 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).

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 even though 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.*

*Note*: 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 we still keep it to avoid bug tracking later.

**Question:** What are the advantages of using the scheduling algorithm described in this assignment compared to other scheduling algorithms you have learned?

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

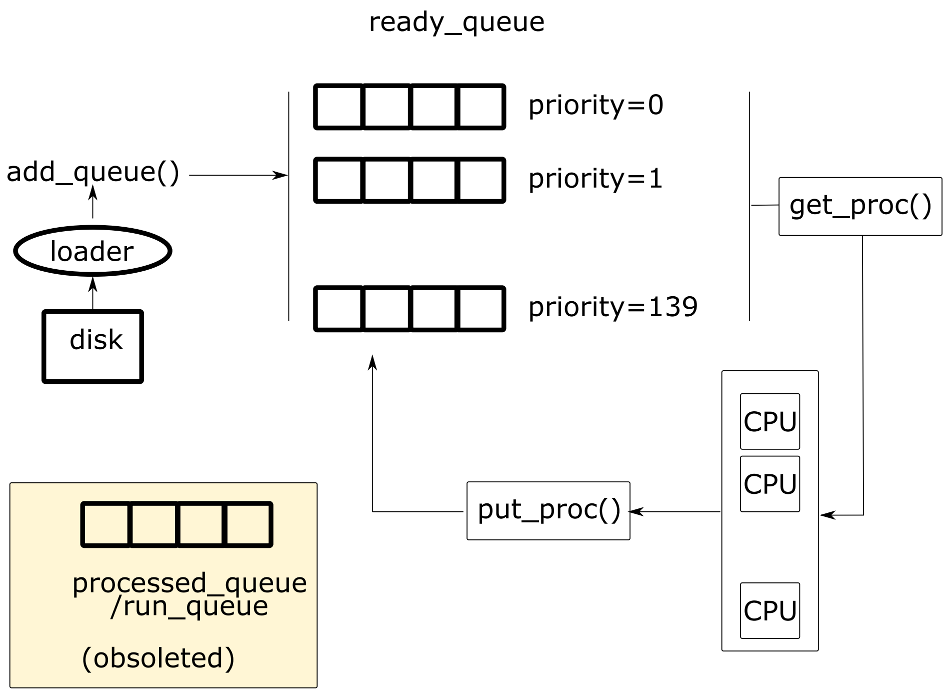

Figure 2: The operation of scheduler in the assignment

## 2.2 Extra task for Honored Program - Implementing Completely Fair Scheduler in a Simple OS

The Completely Fair Scheduler (CFS) is a key component of the Linux kernel's process scheduling mechanism, ensuring fair CPU time allocation among tasks. In this extra task, students will implement CFS as the scheduling algorithm for a simple operating system. CFS operates based on the concept of virtual runtime *vruntime*, which represents the effective CPU time a process has used. The scheduler selects the process with the lowest *vruntime* to run next, ensuring that no process is unfairly starved of CPU resources. When a task is preempted (e.g., a higher-priority task enters), it is reinserted into the tree with its updated *vruntime*. The main features of CFS include:

- Red-Black Tree Scheduling: All runnable tasks are stored in a self-balancing red-black tree, allowing for efficient insertion, deletion, and retrieval.
- Fair CPU Time Distribution: Unlike round-robin schedulers, which allocate fixed time slices, CFS calculates time slices dynamically based on process weight and system load.

Students need to do the following tasks for CFS implementation.

- Study the principles of CFS, including virtual runtime and red-black tree data structures.
- Implement a red-black tree to store processes based on their virtual runtime.
- Develop a function to select the process with the lowest *vruntime* for execution.

$$vruntime + = \frac{\text{actual execution time}}{\text{weight of the task}} \tag{1}$$

where:

- *actual execution time* is the time the task spends running on the CPU.

  - *weight of the task* is derived from its priority (niceness value). Tasks with a lower niceness value (higher priority) get a higher weight, thus accumulating *vruntime* more slowly.
  - Lower *vruntime* means higher priority, so CFS always schedules the task with the smallest *vruntime*.

- Define time slice calculation based on process weights and system load.

$$\text{time slice} = \frac{\text{weight of task}}{\text{total weight of all tasks}} \times \text{target latency} \tag{2}$$

  where:

  - *target latency* is the desired time period in which every runnable task should get CPU time (default is around 20ms in Linux).
  - *total weight of all tasks* is the sum of the weight values of all runnable processes.
  - *time slice* ensures that high-priority tasks (higher weight) get a proportionally larger share of the CPU.

- Define the weight of a task by using a predefined lookup table in the Linux kernel:

$$\text{weight of task} = 1024 \times 2^{\frac{-\text{niceness}}{10}} \tag{3}$$

  where *niceness* ranges from -20 (highest priority) to +19 (lowest priority).

*Question: What are the advantages and disadvantages of CFS compared to MLQ, as described in Section 2.1? Compare the scheduling results with MLQ in terms of average waiting time and total time for CFS management.*

Students can modify the given code to implement the additional option. You should also generate test cases to cover all possible scenarios that may occur during process execution.

In the report, students need to explain the implementation of the CFS scheduling algorithm (including code explanations) and interpret the experimental results using Gantt charts or tables.
A shell script *run.sh*, and a README document are also required to describe how to test the implemented option.

## 2.3 Memory Management

### 2.3.1 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**s (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.

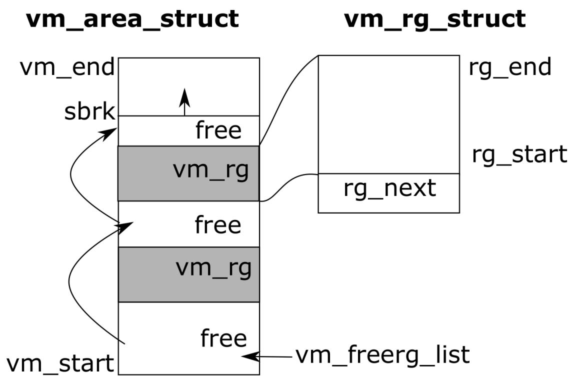

Figure 3: The structure of vm area and region

```
//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.

```c
//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 tracks 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. An important field is the `pgd`, which is the page table directory and contains all page table entries. Each entry maps the page number to the frame number in the paging memory management system. We provide a detailed page-frame mapping in the later section 2.3.3. The `symrgtbl` is a simple implementation of the symbol table. The other fields are mainly used to track specific user operations i.e. caller, fifo page (for referencing). We have included them for your use; you can utilize them as needed (or discard).

**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

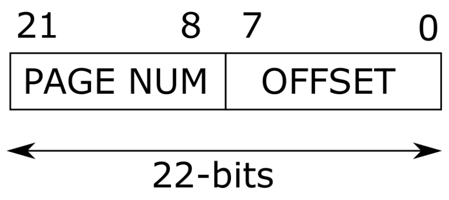

Figure 4: CPU address scheme

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 configurations

### 2.3.2 The system's 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 types of devices, 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 random memory access, sequential/serial memory access, and storage capacity.
Despite the various possible configurations, the logical use of these devices can be distinguished. The RAM device, which belongs to the primary memory subsystem, can be accessed directly from the CPU address bus, allowing it to be read or written using CPU CPU instructions. Meanwhile, SWAP is just a secondary memory device, and all 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 low cost and may have more than one instance. In our settings, we support hardware configured 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 whether the memory access is random or sequential. The fields `free_fp_list` and `used_fp_list` are reserved for retaining unused and used memory frames, respectively.

```c
//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?

### 2.3.3 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 one RAM device and one SWAP instance hardware. We have implemented the capability to handle multiple memory segments, but we mainly focus on the first segment of vm_area (vmaid = 0). The further versions will take into account a sufficient paging scheme for multiple segments and the potential overlap/non-overlap between segments.

**Scheme CPU 32-bit** As in Figure 5

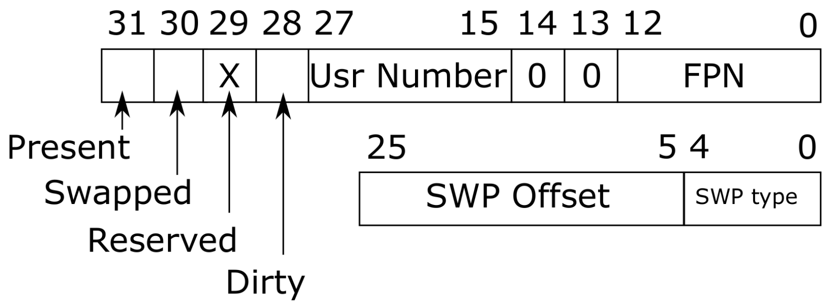

Figure 5: Page Table Entry Format.

This structure allows a userspace process to determine which physical frame each virtual page is mapped to. It contains a 32-bit value for each virtual page, containing the following data:

```
    * Bits 0-12  page frame number (FPN) 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
```

**Page table** The virtual space is isolated for each entity, so 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. Each page must have all entries for the entire 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 **??**. In our chosen highlighted setting in Table 1 we have 16.000-entry table each table cost 64 KB storage space.

In Section 2.3.1, the process can access the virtual memory space in a contiguous manner of the vm area structure. The remaining work deals with the mapping between page and frame to provide the contiguous memory space over the discrete frame storing mechanism. This falls into two main approaches: memory swapping and basic memory operations, i.e. alloc/free/read/write, which mostly keep in touch with **pgd** page table structure.

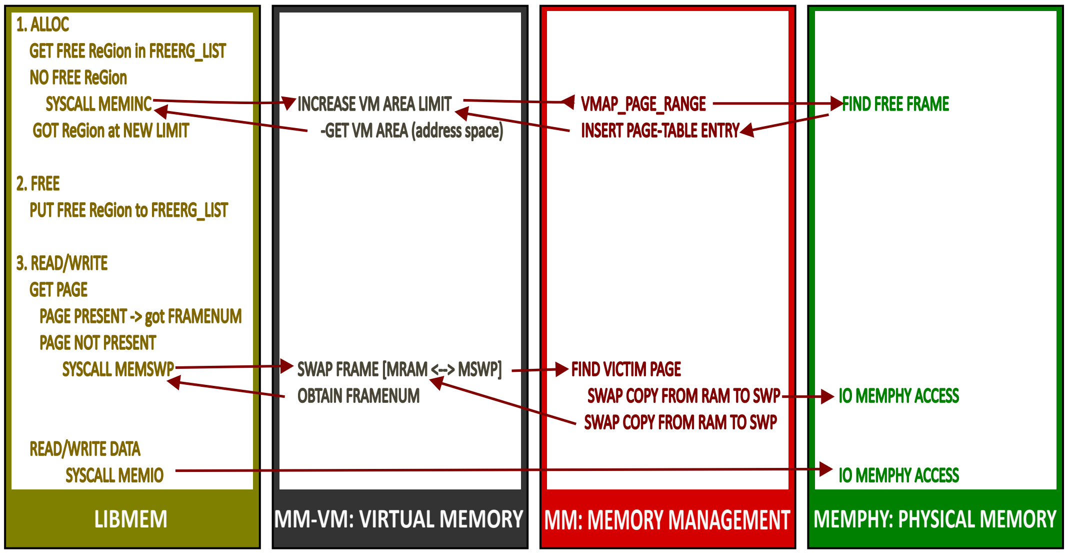

Figure 6: Memory system modules

**Memory swapping** We have been informed that a memory area (/segment) may not be used up to its limit storage space. It means that some storage spaces remain unmapped to MEMRAM. Swapping can help moving the contents of physical frame between the MEMRAM and MEMSWAP. The swapping is a mechanism that copies the frame's content from outside to main memory (RAM). Swapping out, in reverse, attempts to move the content of a frame in MEMRAM to MEMSWAP. In a typical context, swapping helps free up frame of RAM since the size of SWAP device is usually large enough.

**Basic memory operations in paging-based system**

- ALLOC: user call the library functions in libmem and in most cases, it fits into data segment area. If there is no suitable space, we need to expand the memory space by lift up the barrier set by sbrk. Since it have never been touched, it may needs to leverage some MMU systemcalls to obtains physical frames and then map them using Page Table Entry.
- FREE: user call the library functions in libmem to revoke the storage space associated with the given region id. Since we cannot reclaim the taken physical frame which might cause memory holes, we just keep the collected storage space in a free list for future alloc request, all are embedded in libmem library.
- 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 is in the MEMSWAP device, it needs to be brought back to MEMRAM device (swapping in) and if there is a 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 are the advantages and disadvantages of segmentation with paging?

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

```
// 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 compatibility.

**New configuration with explicit declaration of the set of memory sizes** (Be careful, this mode supports a custom memory size, which implies that we comment out or delete or disable the constant `#define MM_FIXED_MEMSZ`). If we are in this mode, 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. The last configuration is the maximum size of virtual memory which is a prerequisite for heap go down setting.

```
[time slice] [N = Number of CPU] [M = Number of Processes to be run]
[RAM_SZ] [SWP_SZ_0] [SWP_SZ_1] [SWP_SZ_2] [SWP_SZ_3]
```

[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 many changes to these values

```
[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.

## 2.4 System Call

### 2.4.1 Description

The system call is the fundamental interface between an application and the Simple Operating System kernel.

**System calls and library wrapper function** System calls are generally not invoked directly, but rather via wrapper function in libstd (or perhaps some other libraries). For details on the direct invocation of a system call, see Figure 7.
The libstd wrapper functions are usually quite thin, doing little work other than copying arguments to the appropriate registers before invoking the system call. However, the wrapper function sometimes does some additional works before invoking the system call.

**System call list** The list of system calls that are available in the Simple Operating System is shown in the following listing:

```
System call        Kernel    Notes
------------------------------------------------------------
listsyscall        3.0
memmap             3.0
killall            3.0       Alpha, was available
                             upto 3.1
```

**System call manual page**

1. **listsyscall** list all system call

```
Name
  listsyscall - list all system call
Sysnopsis
  SYSCALL 0
Description
  listsyscall display the list of all systemcalls.
```

2. **memmap** map memory

```
Name
  memmap - map memory
Sysnopsis
  SYSCALL 17 SYSMEM_OP REG_ARG2 REG_ARG3
Description
  memmap supports various operations on memory mapping. The operations are
       decribed in libmem.h, including :
```

```
 + SYSMEM_MAP_OP with dummy handler
 + SYSMEM_INC_OP with inc_vma_limit() handler
 + SYSMEM_SWP_OP with __mm_swap_page() handler
 + SYSMEM_IO_READ with MEMPHY_read() handler
 + SYSMEM_IO_WRITE with MEMPHY_write() handler
```

3. **killall** kill processes by name

```
Name
  killall - kill processes by name
Sysnopsis
  SYSCALL 101 REGIONID
Description
  killall sends a signals to all processes running the specifed commands.
     The command or program name are specified by memory region
     associated with REGIONID. If no signal name is specified, the
     terminate signal is sent.

  killall is an INCOMPLETED task which acquires the remaining
     implementation by student to terminate the execution of all
     processes whose program name match the string stored in REGIONID.
```

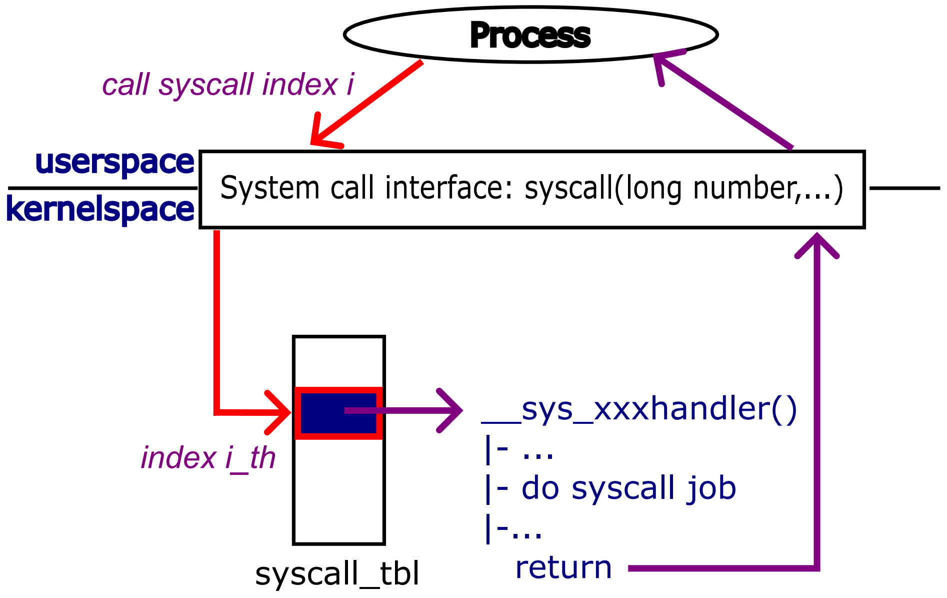

Figure 7: The handling of user application invoking a system call

**Question** What is the mechanism to pass a complex argument to a system call using the limited registers?

### 2.4.2 Adding a system call to Simple Operating System

In this guide, you will learn how to add a simple system call to the Simple Operating System.

**Creation**

1. Create a C file for your system call in `src/sys_xxxhandler.c`

```
// From src/sys_xxxhandler.c
#include "common.h"
#include "syscall.h"
#include "stdio.h"

int __sys_xxxhandler(struct pcb_t *caller, struct sc_regs* reg)
{
    /* TODO: implement syscall job */
    printf("The first system call parameter %d\n", regs->a1);
    return 0;
}
```

2. Create a Makefile entry for your system call with a unique indexing, i.e. 440

```
# From Makefile
SYSCALL_OBJ += $(addprefix $(OBJ)/, sys_xxxhandler.o)
```

3. Add your system call to the kernel's system call table

```
# From src/syscall.tbl
440     xxx         sys_xxxhandler
```

**Installation** In this section, you will install the new kernel and prepare your Simple Operating System to boot into it

1. Compile the kernel source code

```
$make all
```

2. Boot your Simple Operating System

```
$./os <input_file>
```

**Results** In this section, you will write a program to check whether your system call works or not. After that, you will see your system call in action.

1. Create a program **sc** to invoke your system call

```
20 1
syscall 440 1
```

2. A create a configuration file **os_syscall** to boot your Simple Operating System

```
2 1 1
2048 16777216 0 0 0
9 sc  15
```

3. Boot your Simple Operating System

```
$./os os_syscall
```

4. Check the last line of the message output

```
...
The first system call paramter 1
 CPU 0: Processed  1 has finished
 CPU 0 stopped
```

**Congratulations! You have successfully added a system call to the Simple Operating System!**

**Question** What happens if the syscall job implementation takes too long execution time?

## 2.5 Put It All Together

Finally, we combine scheduler and memory management to form a complete OS. Figure 8 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.

Check your work by first compiling the whole source code

```
make all
```

and compare your output with those in `output`. Remember that as we are running in multi-processes environment, there may be more than one correct result. All the outputs are used as samples and are not the restricted results. Your results only need to be explainable and be comparable with **theoretical framework** and does not need to match the output sample.

**Question:** What happens if the synchronization is not handled in your Simple OS? Illustrate the problem of your simple OS (assignment outputs) by example if you have any.

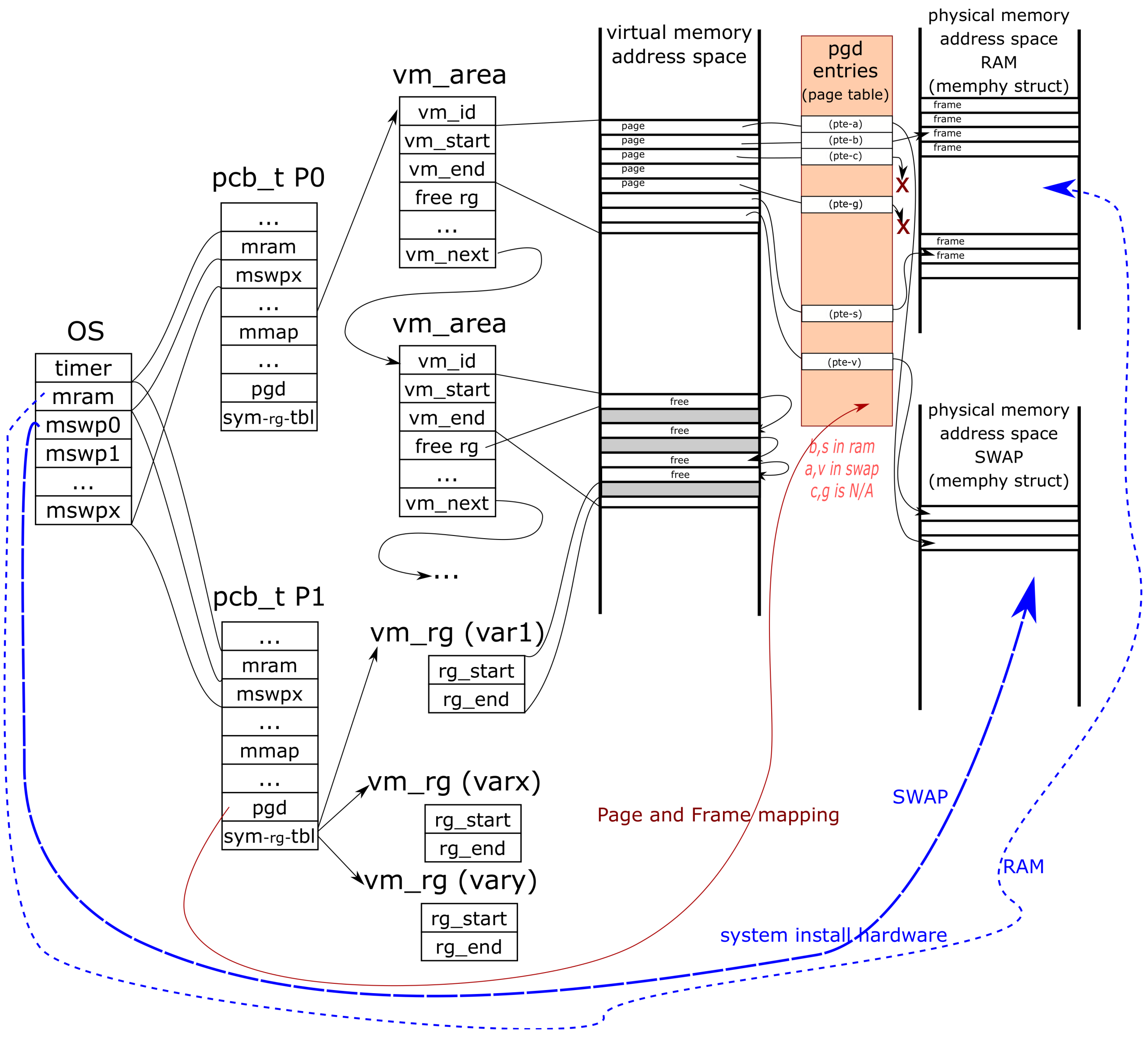

Figure 8: The operation related to virtual memory in the assignment

# 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 Requirements

**Scheduler** implement the scheduler that employs MLQ policy as described in Section 2.1.

**Memory Management** implement the paging subsystem.

**Systemcall** implement the remaining task of the system call killall. The program name fetching has been provided, students need to figure out the matching processes and terminate their execution (as a common killall command usage).

**Questionnaire** student need to answer all questions in the assignment description.

## 3.3 Report

Write a report that answers questions in the implementation section and interprets 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 the memory allocation in heap and data segments.
- SystemCall: show the inter-module interactions among memory storing process name, OS process control, and scheduling queue management.
- Overall: students should find their own way to interpret the results of simulation.

After you finish the assignment, move your report to source code directory and compress the entire directory into a single file named `assignment_STUDENTID.zip` and submit to LMS.

## 3.4 Grading

You must complete this assignment in groups of 4 or 5 students. The overall grade for your group is determined by the following components:

- Demonstration (7 points)
  - Scheduling: 3 points
  - MMU Paging: 2 points
  - Systemcall: 2 points
- Report (3 points)

## 3.5 Code of ethics

Faculty staff members involved in code development reserved all the copyright of the project source code.
**Source Code License Grant**: Author(s) hereby grant(s) to Licensee personal permission to use and modify the Licensed Source Code for the sole purpose of studying while attending the course CO2018 at HCMUT.

## Revision History

| **Revision** | **Date** | **Author(s)** | **Description** |
|---|---|---|---|
| 1.0 | 01.2019 | pdnguyen, Minh Thanh CHUNG, Hai Duc NGUYEN | Created CPU, scheduling, memory |
| 1.1 | 09.2022 | pdnguyen | Add Multilevel Queue (MLQ) CPU Scheduling |
| 2.0 | 03.2023 | pdnguyen | Initialize MM Paging Framework |
| 2.1 | 10.2023 | pdnguyen | Add Page Replacement |
| 2.2 | 03.2024 | pdnguyen | Add CPU Translation Lookaside Buffer (TLB) |
| 2.3 | 10.2024 | pdnguyen | Add heap segment |
| 3.0 | 03.2025 | pdnguyen | Add systemcall |