VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

CO2017 - OPERATING SYSTEMS

CC04 - GROUP 02 - ASSIGNMENT REPORT

Simple Operating System

Instructor: **PROF. Pham Hoang Anh**

HO CHI MINH CITY, MAY 2023

Member list & Workload

No.	Full name	Student ID	Contribution
1	Nguyen Quang Thien	2152994	50%
2	Nguyen Khanh Nam	2153599	50%

Contents

1 Introduction

In this assignment, three components of a minimized operating system will be simulated: the scheduler, synchronization, and the mechanism of memory allocation from virtual to physical memory. Students will gain hands-on experience and insight into the principles of a simple OS, understand the roles of these key modules, and learn how to apply them in practice using C programming language. The scheduler controls how processes are carried out, synchronization makes ensuring that resources are used safely, and memory allocation facilitates the conversion of virtual addresses to physical memory locations. This assignment provides a foundation for understanding OS principles and components so that it is better prepared for students to tackle more complex systems-level programming tasks in the future.

Figure 1: Operating systems.

The first part is about the Scheduler, which implements a priority queue data structure to manage processes waiting for CPU time and a multi-level queue (MLQ) policy for scheduling processes based on their priority level. (The synchronization issue will be discussed in this part)

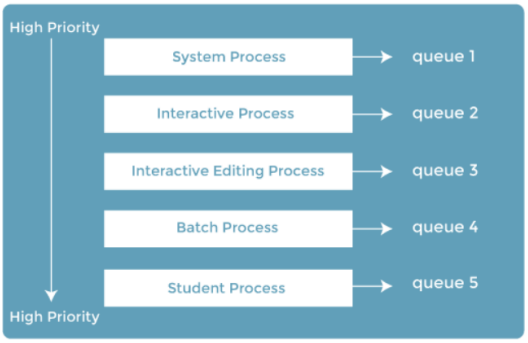

Figure 2: Multi Level Queue scheduling algorithm.

The second part is about Memory Management which encompasses several key components:

- Virtual memory mapping allows each process to have its own isolated memory space.

- The physical memory of the system, such as RAM, is managed to effectively allocate and deallocate memory.

- Paging-based address translation scheme is used to convert virtual addresses to physical addresses.

- Wrapping up paging-oriented implementations involves configuring settings and understanding the behavior of the memory management system.

The operating system will run efficiently and with optimal resource usage attributable to effective memory management.

2 Theoretical Basis

2.1 Multilevel Queue

It is clear that the **Multilevel Queue** Scheduling can lead to the starvation of lower priority processes which can be waiting for infinity. However, when the multilevel queue uses the time slice for each queue that can prevent the above situation which scheduling is referred to as the **Multilevel Feedback Queue**.

A CPU scheduling system entitled the **Multilevel Feedback Queue** (MLFQ) employs numerous queues to manage processes and dynamically modifies the priority of operations based on their activity. It depends on the assumption that a process's priority level may need to alter over time depending on its resource needs and execution characteristics. Processes are stored in a number of queues that each have a distinct priority. The highest priority queue is at the top of the hierarchy, while the lowest priority queue is at the bottom.

The decision of which process to run next is made using a separate scheduling algorithm for each queue. For instance, while the lower priority queues employ Round-Robin (RR) scheduling, the queue with the highest priority may adopt a First-Come-First-Serve (FCFS) method. When a process arrives, it is added to the queue with the highest priority.

When a process is dispatched, it is allowed to run for a maximum time quantum in the current queue (highest priority queue). If the process does not complete within the time quantum, it is preempted and moved to the next queue with a smaller time quantum. (If a process waits in a lower priority queue for too long, its priority will be increased to ensure that it gets a chance to execute or known as the feedback mechanism. This helps prevent starvation)

Benefits of Multilevel Feedback Queue scheduling:

- The dynamic priority adjustment mechanism allows the scheduler to adapt to changing conditions, making it more responsive to changes in the workload.
- It strikes a balance between throughput and reaction time and prevents starvation. While low-priority processes will eventually have their chance to run, high-priority ones will do so right away.
- The MLFQ scheduler can manage a range of workload kinds and execution characteristics through the utilization of multiple queues with various scheduling methods.

Drawbacks of Multilevel Feedback Queue scheduling:

- More complex than other scheduling algorithms due to a large number of parameters and thresholds that must be set, it can be difficult to implement and fine-tune.
- The choice of scheduling methods and parameter settings may have a negative impact on the MLFQ scheduler's performance as the difficulties while choosing the optimal time slices for each queue.
- The feedback mechanism can cause priority inversion and lead to unpredictable behavior.

2.2 Memory Management

2.2.1 The virtual memory mapping in each process

Virtual memory mapping is a technique used by operating systems to regulate how much memory is available to each process. Each process in contemporary operating systems has access to a separate virtual memory space, which is partitioned into several contiguous regions known as virtual memory areas. Each virtual memory area corresponds to a specific segment of the process's program, such as the code, stack, or heap.

The process keeps track of these virtual memory areas through a data structure called a page table, which maps the virtual addresses used by the process to physical addresses in the main memory. This allows each process to have its own isolated memory space, also protects it from interfering with other processes, and allocates memory to processes in a flexible way.

2.2.2 The system physical memory

The physical memory of a computer system refers to the actual hardware components used to store data and programs that are currently being used by the computer. This memory is divided into two primary categories: RAM and SWAP.

- RAM is a type of volatile memory that stores data and program instructions temporarily for the CPU to access quickly. It is typically faster than non-volatile storage and can be accessed randomly, hence the name "random access". RAM is used to hold the operating system and any programs that are currently running, and the amount of RAM a system has can directly affect its performance.
- SWAP is a secondary memory device, on the other hand, a type of non-volatile storage used by the operating system to simulate additional memory space when the RAM is full. When the RAM is full, the operating system will swap out unused data from the RAM and store it in SWAP, freeing up space for new data. SWAP is slower than RAM but allows the system to operate with larger amounts of data and programs than the available physical RAM.

Since RAM has a limited capacity, the system typically equips a larger SWAP device(s) to compensate. The system's memory management subsystem is responsible for managing the allocation of memory resources across both devices.

The physical memory of a system is typically managed by the operating system, which allocates and deallocates memory as needed for various processes and applications. Physical memory is also divided into pages, which are the smallest units of memory that can be allocated or deallocated. The management of physical memory is critical for the overall performance and stability of a computer system.

2.2.3 Paging-based address translation scheme

The translation supports both segmentation and segmentation with paging. The current version uses a single-level paging system that leverages one RAM device and one SWAP instance hardware. Each process has a completely isolated and unique space and its own page table containing information about which physical frame each virtual page is mapped to.

The process can access the virtual memory space in a contiguous manner of vm area structure. The mapping between page and frame provides the contiguous memory space over the discrete frame storing mechanism. There are two main approaches to memory operations in the paging-based system: memory swapping and basic memory operations (alloc/free/read/write).

Memory swapping:

- Swapping helps to move the contents of a physical frame between the MEMRAM and MEMSWAP, and it can be used to gain free frames of RAM since the size of the SWAP device is usually large enough.

Basic memory operations in the paging-based system:

- Allocating available space: in most cases, it fits into the available regions. If there is no such suitable space, need to lift up the barrier and since it has never been touched, it may need to provide some physical frames and then map them using Page Table Entry.
- Freeing storage space: 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 requests.
- Reading/Writing to pages: requires getting the page to be presented in the main memory. The most resource-consuming step is page swapping. If the page was in the MEMSWAP device, it needs to bring that page back to the MEMRAM device (swapping in) and if it lacks space, we need to give back some pages to the MEMSWAP device (swapping out) to make more rooms.

Paging-based address translation is a technique used by modern operating systems to translate virtual addresses used by a process into physical addresses in the system's main memory. In this scheme, the virtual address space of a process is divided into fixed-size pages, typically ranging from 4KB to 2MB in size. The physical memory is also divided into pages of the same size.

When a process references a virtual address, the operating system first uses a page table to translate the virtual page number to a physical page number. The page table is a data structure that maps virtual pages to physical pages, and it is stored in the kernel space of the operating system.

The translation process involves splitting the virtual address into a virtual page number and an offset within the page. The virtual page number is then used to look up the corresponding physical page number in the page table. Once the physical page number is obtained, the offset within the page is added to it to obtain the physical memory address of the data being accessed.

This paging-based address translation scheme allows each process to have its own isolated virtual memory space and protects it from interfering with other processes. It also allows the operating system to allocate physical memory to processes in a flexible way, since pages can be swapped in and out of physical memory as needed. Overall, paging-based address translation is a fundamental mechanism used by modern operating systems to provide memory management and protection.

2.2.4 Wrapping-up all paging-oriented implementations

Clarifies how the approach of configuration control using constant definition is employed to manage the interference among feature-oriented program modules by isolating each feature through a system of configuration.

By utilizing constant definitions and manipulating settings in the **"include/os-cfg.h"** file, different subsystems can be maintained separately within a single version of the code. This approach allows for enabling or disabling specific features in the simulation program, effectively isolating each feature and managing their interactions.

There are two specific configuration settings examples, namely MM PAGING and MM FIXED MEMSZ. The MM PAGING setting involves adding memory management fields to the PCB structure,

which can be enabled by defining the associated configuration line in **"include/os-cfg.h"**. This setting enables the use of the memory paging module.

The MM FIXED MEMSZ setting, on the other hand, enables compatibility with old versions of the input file while operating in the new paging memory management mode. By enabling this setting, backward compatibility is maintained, allowing the program to handle the old setting with the *"# define MM FIXED MEMSZ"* line.

A new configuration mode is further included that requires the explicit specification of memory size. In this mode, the constant definition for MM FIXED MEMSZ needs to be commented out, deleted, or deactivated. An additional line in the input file that specifies the system's physical memory capacity and up to four memory swap sizes must be included.

```
[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]
```

Figure 3: The constant definition control the highlighted input line.

It is important to carefully examine both the input file and the contents of **"include/os-cfg.h"** to understand how the simulation program behaves and to identify any unexpected or unusual aspects related to paging-oriented implementations.

3 Code Implementation

3.1 Scheduler

To begin with, we need to handle both the **enqueue()** function and **dequeue()** function which is included in file **queue.c**. The meaning of those functions is described as follow:

- **enqueue():** put a new process to queue q.
- **dequeue():** return a PCB whose priority is the highest in the queue and remove it from q.

Here is how we define those functions in the file. Starting with the **enqueue()** function:

Before we add a new process to the queue, first we need to check if the process or the queue which are passed into the function is NULL or not. If one of them is a null pointer, the **enqueue()** function does nothing and returns.

Otherwise, it will continue to check if there is enough space for that process. If the queue meets its limit, the process cannot be added to the queue anymore and the function returns without doing any modification.

If the two above conditions have not been met, the process will then continue to be added to the queue, and also the size of that queue is increased by one.

Next, we will continue to implement the **dequeue()** function:

In this assignment, we are expected to implement the multilevel feedback queue based on the process's priority. So, in the **dequeue()** function, based on the information that each queue contains the processes which have equal priority, we just need to pop out the process of the first index in the queue. Make sure to decrease the size of that queue by 1 after taking out the suitable process.

And that is the end of the content in file queue.c which is used to implement the queue concept. Next, we will turn to implementing the sched.c file. There are only one functions that we need to implement in this file:

- **get_mlq_proc():** get a process from PRIORITY [ready_queue].

First is the *get_mlq_proc()* function, here is how we implement it:

The idea is that we will traverse the queue through the loop. Each time, we will check if that queue is empty or not and also check if that queue's CPU time usage does not exceed its limit(calculated by MAX_PRIOR - prior). If the queue's slot is greater than its limit or that queue is empty, the cpu will then find another queue with the lower priority.

If the current queue's priority is equal to MAX_PRIOR, we will reset the slots and continue to go back to the first queue.

The loop will stop when it finds a suitable process. Otherwise, if the loop has traversed twice, which means that there is no process, the function will return a null pointer.

And that is the end of the contents in **Scheduler**. Next, we will then continue to implement the **Memory Management**.

3.2 Memory Management

In this assignment, we are expected to build a simple memory management system by implementing 3 files. They are mm.c, mm-vm.c and mm-memphy.c.

3.2.1 Implement functions in mm-vm.c

To begin with, let's first implement the mm-vm.c file. There are 5 functions that need to be filled in this file.

- **__alloc():** allocate a region memory.
- **__free():** remove a region memory.
- **find_victim_page():** find victim page.
- **pg_getpage():** get the page in ram.
- **validate_overlap_vm_area():** validate the planned memory area is not overlapped.

First is the **__alloc()** function. This function is used to allocate a region memory. What we are expected to implement in this function is that we need to handle the case that if there is no free region that can fit the "size", the function will next continue to increase the limit by using the **inc_vma_limit()** function. After that, we need to handle the free hold and assign that region into the free list.

The second function that we need to handle is the **__free()**. What we need to do is that we need to find the region to free and enlist it into the free region list.

The third one is the **pg_getpage()** which is used to get the page in RAM. We need to handle the case that the page is not presented and what we have to do is to make it online. By finding the victim page and then finding a free frame in the swap area, we will then swap and then update the page table. So that the target page will be at online status. Make sure to bring the page into the fifo list.

The next function is **validate_overlap_vm_area()**. We need to check if the vma is overlapped with other vma or not. By using a pointer to traverse through the vma_list. By checking if the vmastart or vmaend which is in the range of other regions which means that there is an overlap, the function will then return -1.

The final function that we need to implement in this file is the **find_victim_page()** function. This function will be based on the FIFO page replacement algorithm. And the page which is the oldest that has been added into the list can be determined at the tail of the list. By using the basic knowledge of linked list, this function can be handled easily.

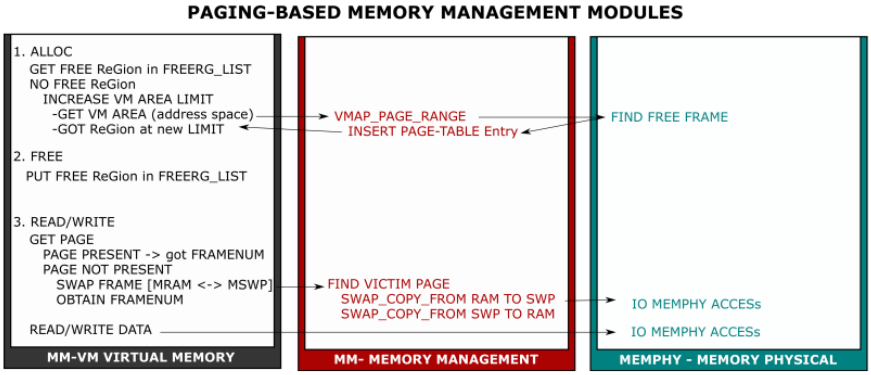

Figure 4: Memory system modules.

3.2.2 Implement functions in mm.c

Next, we will implement two functions in file mm.c, those functions are respectively:

- **vmap_page_range():** map a range of page at aligned address.
- **alloc_pages_range():** allocate req_pgnum of frame in ram.

To begin with, the **vmap_page_range()** function is used to map the virtual address to the new frame to extend the virtual memory area's size.

And for the **alloc_pages_range()** function, this function is used to allocate a number of frame based on the request page number. If there is not enough free frame in RAM, then the page replacement algorithm will be applied to get free frame in the swap area and bring it to the RAM.

3.2.3 Implement function in mm-memphy.c

In this file, there is only one function that we need to handle which is called **MEMPHY_dump()**. The function is used to show the content in the physical memory.

After all, we have finished implementing the content of memory management. Through out this, we achieve more knowledge and be able to understand some fundamental concepts of an simple operating system.

3.3 Output of the program

Here are some output of the program. Note that there are more but we just demonstrate five of them in this report for shortened. The other output can be found in the zip file.

3.3.1 Output of file sched

In order to build this section's Scheduler, we essentially follow the **Multilevel Queue (MLQ) principle**. Each queue has a fixed time slot to use the CPU, and when it is fully filled, the system must switch to the other process in the next queue and leave the remaining work for a future slot even though it requires a finished round of the **ready_queue**. And the traversed step of the **ready_queue** list is a fixed formulated number based on the priority, i.e. Time slot = (MAX PRIO - prio).

```
knam@knam:~$ ./os sched
ld_routine
  Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
Time slot   0
  Loaded a process at input/proc/p2s, PID: 2 PRIO: 20
  CPU 0: Dispatched process  1
Time slot   1
  CPU 1: Dispatched process  2
  Loaded a process at input/proc/p3s, PID: 3 PRIO: 7
Time slot   2
Time slot   3
Time slot   4
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   5
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  3
Time slot   6
Time slot   7
Time slot   8
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   9
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  3
Time slot  10
  CPU 0: Processed  1 has finished
  CPU 0: Dispatched process  2
Time slot  11
Time slot  12
Time slot  13
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  3
Time slot  14
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot  15
Time slot  16
  CPU 1: Processed  3 has finished
  CPU 1 stopped
Time slot  17
Time slot  18
  CPU 0: Processed  2 has finished
  CPU 0 stopped
```

In the **sched** file, the first line "4 2 3" means that the time slot is 4, using 2 CPU(s) run parallel and having 3 processes (p1, p2, p3):

Process ID (PID)	Priority (PRIO)	Arrival Time	Number of instructions
1 (p1)	1	0	10
2 (p2)	20	1	12
3 (p3)	7	2	11

In interval time p1 is 0 and p1 is dispatched to the CPU 0 to execute. When the interval time is 2, the p2 will go to CPU 1 due to CPU 0 executing p1. At the time 4, p1 is put into the queue with the priority increase by 1. After that compare p1 and p3 which reach the interval time at 2, the **multilevel queue** prioritize the queue which is the smallest number, and having the process in. Process p1 has priority after increasing by 1 is 2 and the priority of p3 is 7) so that p1 will continue to go to CPU 0 to be executed. Keep applying this technique, and when the process runs out of instructions, check the queue to see whether there are any processes left running that need to be sent to the CPU to finish. And so on, we will have the **Gantt diagram** below:

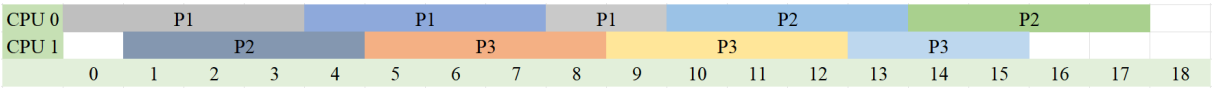

Figure 5: Gantt diagram

3.3.2 Output of file sched_0

```
knam@knam:~$ ./os sched_0
Time slot   0
ld_routine
  Loaded a process at input/proc/s0, PID: 1 PRIO: 12
  CPU 0: Dispatched process  1
Time slot   1
Time slot   2
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   3
Time slot   4
  Loaded a process at input/proc/s1, PID: 2 PRIO: 20
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   5
Time slot   6
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   7
Time slot   8
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   9
Time slot  10
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  11
Time slot  12
  CPU 0: Put process  1 to run queue
```

```
  CPU 0: Dispatched process  1
Time slot   13
Time slot   14
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   15
  CPU 0: Processed  1 has finished
  CPU 0: Dispatched process  2
Time slot   16
Time slot   17
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot   18
Time slot   19
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot   20
Time slot   21
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot   22
  CPU 0: Processed  2 has finished
  CPU 0 stopped
```

In the **sched_0** file, the first line "2 1 2" means that the time slot is 2, using 1 CPU and having 2 processes (s0, s1):

Process ID (PID)	Priority (PRIO)	Arrival Time	Number of instructions
1 (s0)	12	0	15
2 (s1)	20	4	7

Similar to the **sched** file but more basic than cause just employing only one CPU, the s0 process contains 15 instructions and the s1 process has 7 which means the CPU 0 will terminate at time slot 22. And it is shown in the **Gantt diagram** below:

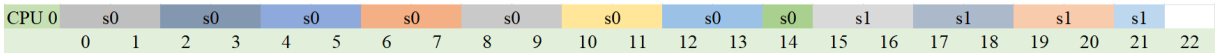

Figure 6: Gantt diagram

3.3.3 Output of file sched_1

```
knam@knam:~$ ./os sched_1
Time slot   0
ld_routine
  Loaded a process at input/proc/s0, PID: 1 PRIO: 12
  CPU 0: Dispatched process  1
Time slot   1
Time slot   2
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   3
Time slot   4
  Loaded a process at input/proc/s1, PID: 2 PRIO: 20
```

```
 CPU 0: Put process  1 to run queue
 CPU 0: Dispatched process  1
Time slot   5
Time slot   6
 Loaded a process at input/proc/s2, PID: 3 PRIO: 20
 CPU 0: Put process  1 to run queue
 CPU 0: Dispatched process  1
Time slot   7
 Loaded a process at input/proc/s3, PID: 4 PRIO: 7
Time slot   8
 CPU 0: Put process  1 to run queue
 CPU 0: Dispatched process  4
Time slot   9
Time slot  10
 CPU 0: Put process  4 to run queue
 CPU 0: Dispatched process  4
Time slot  11
Time slot  12
 CPU 0: Put process  4 to run queue
 CPU 0: Dispatched process  4
Time slot  13
Time slot  14
 CPU 0: Put process  4 to run queue
 CPU 0: Dispatched process  4
Time slot  15
Time slot  16
 CPU 0: Put process  4 to run queue
 CPU 0: Dispatched process  4
Time slot  17
Time slot  18
 CPU 0: Put process  4 to run queue
 CPU 0: Dispatched process  4
Time slot  19
 CPU 0: Processed  4 has finished
 CPU 0: Dispatched process  1
Time slot  20
Time slot  21
 CPU 0: Put process  1 to run queue
 CPU 0: Dispatched process  1
Time slot  22
Time slot  23
 CPU 0: Put process  1 to run queue
 CPU 0: Dispatched process  1
Time slot  24
Time slot  25
 CPU 0: Put process  1 to run queue
 CPU 0: Dispatched process  1
Time slot  26
 CPU 0: Processed  1 has finished
 CPU 0: Dispatched process  2
Time slot  27
Time slot  28
 CPU 0: Put process  2 to run queue
 CPU 0: Dispatched process  3
Time slot  29
Time slot  30
 CPU 0: Put process  3 to run queue
 CPU 0: Dispatched process  2
Time slot  31
Time slot  32
```

```
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
Time slot  33
Time slot  34
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  35
Time slot  36
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
Time slot  37
Time slot  38
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  39
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  3
Time slot  40
Time slot  41
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot  42
Time slot  43
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot  44
Time slot  45
  CPU 0: Processed  3 has finished
  CPU 0 stopped
```

In the **sched_1** file, the first line "2 1 4" means that the time slot is 2, using 1 CPU and having 4 processes (s0, s1, s2, s3):

Process ID (PID)	Priority (PRIO)	Arrival Time	Number of instructions
1 (s0)	12	0	15
2 (s1)	20	4	7
3 (s2)	20	6	12
4 (s3)	7	7	11

This is identical to the **shed_0** file, utilizing one CPU. However, having the case in which 2 processes are the same priority so the CPU will get the process that has loaded sooner. The s0 process contains 15 instructions and the s1, s2, and s3 have 7, 12, and 11 respectively which means the CPU 0 will terminate at time slot 45. It can be seen in the **Gantt diagram** below:

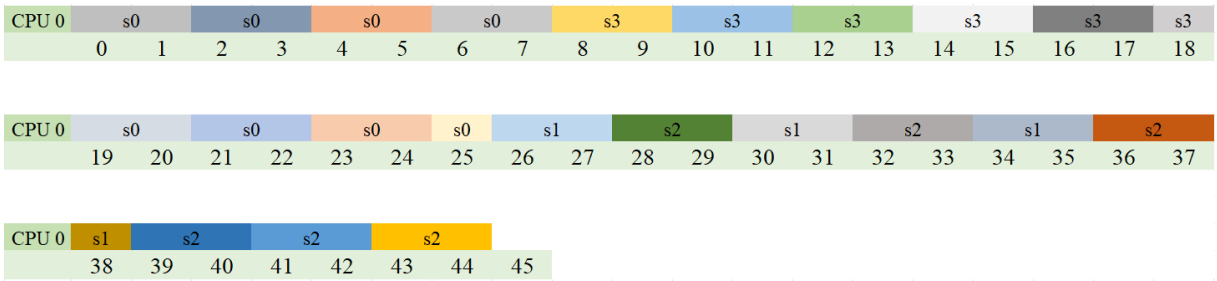

Figure 7: Gantt diagram

It is instantly clear from the running tests above that the **time quantum** specified in the input file will impact the amount of time that each process runs at a particular location in time. Based on the **arrival time** and **priority**, it must determine which process will be serviced before executing in the **time quantum** mentioned. As a result, this procedure will continue until it is accomplished.

▷ **Overall**, as seen by the outputs found in the **Sched**, **Sched_0**, and **Sched_1** files, we can say that they operate remarkably comparable to the **Round Robin** algorithm while with **priority processes**.

3.3.4 Output of file os_0_mlq_paging

```
knam@knam:~$ ./os os_0_mlq_paging
Time slot   0
ld_routine
  Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
  CPU 1: Dispatched process  1
Time slot   1
Time slot   2
  Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
  CPU 0: Dispatched process  2
Time slot   3
  Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
Time slot   4
  Loaded a process at input/proc/p1s, PID: 4 PRIO: 0
Time slot   5
Time slot   6
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
================ Memory content ========================
Time slot   7
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  3
Time slot   8
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  4
Time slot   9
Time slot  10
Time slot  11
Time slot  12
Time slot  13
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  1
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
================ Memory content ========================
Address: 20, Value 100
Time slot  14
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  3
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
```

```
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
================ Memory content ================
Address: 20, Value 100
Time slot   15
read region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
================ Memory content ================
Address: 20, Value 102
Time slot   16
write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
================ Memory content ================
Address: 20, Value 102
Time slot   17
  CPU 1: Processed  1 has finished
  CPU 1: Dispatched process  4
Time slot   18
  CPU 0: Processed  3 has finished
  CPU 0: Dispatched process  2
Time slot   19
Time slot   20
Time slot   21
  CPU 1: Processed  4 has finished
  CPU 1 stopped
Time slot   22
  CPU 0: Processed  2 has finished
  CPU 0 stopped
```

3.3.5 Output of file os_1_mlq_paging_small_1K

```
knam@knam:~$ ./os os_1_mlq_paging_small_1K
Time slot   0
ld_routine
Time slot   1
  Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
Time slot   2
  CPU 1: Dispatched process  1
  Loaded a process at input/proc/s3, PID: 2 PRIO: 39
Time slot   3
  CPU 3: Dispatched process  2
Time slot   4
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
  Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  2
Time slot   5
  CPU 2: Dispatched process  3
```

```
Time slot   6
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
  Loaded a process at input/proc/s2, PID: 4 PRIO: 120
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  2
  CPU 0: Dispatched process  4
Time slot   7
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
================ Memory content ================
  CPU 2: Put process  3 to run queue
  CPU 2: Dispatched process  3
  Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
Time slot   8
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  5
Time slot   9
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  2
  Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
  CPU 2: Put process  3 to run queue
  CPU 2: Dispatched process  6
Time slot  10
  CPU 1: Put process  5 to run queue
  CPU 1: Dispatched process  3
Time slot  11
  CPU 2: Put process  6 to run queue
  CPU 2: Dispatched process  6
  Loaded a process at input/proc/s0, PID: 7 PRIO: 38
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  2
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  5
Time slot  12
  CPU 1: Put process  3 to run queue
  CPU 1: Dispatched process  3
Time slot  13
  CPU 0: Put process  5 to run queue
  CPU 0: Dispatched process  7
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  2
  CPU 2: Put process  6 to run queue
  CPU 2: Dispatched process  6
Time slot  14
  CPU 1: Processed  3 has finished
  CPU 1: Dispatched process  4
  CPU 3: Processed  2 has finished
  CPU 3: Dispatched process  5
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000006
00000004: 80000007
================ Memory content ================
```

```
Address: 20, Value 100
Time slot   15
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000006
00000004: 80000007
================ Memory content ================
Address: 20, Value 100
Address: 64, Value 102
  CPU 2: Put process  6 to run queue
  CPU 2: Dispatched process  6
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot   16
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  4
  CPU 3: Put process  5 to run queue
  CPU 3: Dispatched process  5
write region=0 offset=0 value=0
print_pgtbl: 0 - 512
00000000: 80000006
00000004: 80000007
================ Memory content ================
Address: 20, Value 100
Address: 64, Value 102
Address: 232, Value 1
   Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot   17
  CPU 2: Put process  6 to run queue
  CPU 2: Dispatched process  8
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  6
  CPU 3: Processed  5 has finished
  CPU 3: Dispatched process  7
Time slot   18
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  4
Time slot   19
  CPU 2: Put process  8 to run queue
  CPU 2: Dispatched process  8
  CPU 3: Put process  7 to run queue
  CPU 3: Dispatched process  7
  CPU 0: Processed  6 has finished
  CPU 0: Dispatched process  1
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
================ Memory content ================
Address: 20, Value 100
Address: 64, Value 102
Address: 232, Value 1
Time slot   20
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  4
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000000
```

```
00000004: 80000001
00000008: 80000002
00000012: 80000003
========== Memory content ==========
Address: 20, Value 100
Address: 64, Value 102
Address: 232, Value 1
Time slot   21
  CPU 3: Put process  7 to run queue
  CPU 3: Dispatched process  7
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
read region=2 offset=20 value=102
  CPU 2: Put process  8 to run queue
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
========== Memory content ==========
Address: 20, Value 102
Address: 64, Value 102
Address: 232, Value 1
  CPU 2: Dispatched process  8
Time slot   22
write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000000
00000004: 80000001
00000008: 80000002
00000012: 80000003
========== Memory content ==========
Address: 20, Value 102
Address: 64, Value 102
Address: 232, Value 1
  CPU 1: Processed  4 has finished
  CPU 1 stopped
Time slot   23
  CPU 2: Put process  8 to run queue
  CPU 2: Dispatched process  8
  CPU 3: Put process  7 to run queue
  CPU 3: Dispatched process  7
  CPU 0: Processed  1 has finished
  CPU 0 stopped
Time slot   24
  CPU 2: Processed  8 has finished
  CPU 2 stopped
Time slot   25
  CPU 3: Put process  7 to run queue
  CPU 3: Dispatched process  7
Time slot   26
Time slot   27
  CPU 3: Put process  7 to run queue
  CPU 3: Dispatched process  7
Time slot   28
  CPU 3: Processed  7 has finished
  CPU 3 stopped
```

4 Inquiries and Responses

4.1 Scheduler

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

- When compared to alternative scheduling methods, utilizing a priority queue has the benefit of allowing jobs to be scheduled according to their relative priority. As a result, jobs that are more urgent or important, such as real-time or critical tasks, might take precedence over tasks that are lower on the priority scale.
- Other scheduling algorithms, such as Round Robin (RR) Scheduling and First-Come, First-Served (FCFS) Scheduling, do not consider the relative importance of jobs. RR schedules jobs based on a set time quantum, whereas FCFS schedules tasks in the order they are received. Jobs are prioritized by SJF and SRTF scheduling algorithms based on their anticipated execution times, which may or may not match the jobs' relative priorities.
- By using a priority queue, the scheduler can ensure that high-priority tasks are executed first, which can be critical for certain applications, such as real-time systems or emergency response systems. This can help to improve the overall responsiveness and efficiency of the system.

4.2 Memory Management

4.2.1 The virtual memory mapping in each process

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?

Multiple memory segments or memory sections in an OS implementation provide the following upsides:

- Changeability: The OS can allocate memory to processes according to their memory needs when there are various memory segments. This enables the OS to effectively utilize memory and prevent memory waste.
- Protection: Each memory segment can be assigned its own protection level, allowing the OS to prevent unauthorized access or modification of the memory.
- Memory fragmentation: The use of multiple memory segments reduces memory fragmentation. This is because each segment can be assigned a specific size and type of memory, which allows the OS to manage the memory allocation more efficiently.
- Virtual memory: The implementation of virtual memory requires several memory segments. The amount of physical memory that the OS can utilize as an extension of other memory can be greatly increased by enabling virtual memory.
- Enhanced memory management: The OS may make memory management simpler by using several memory segments. This is due to the fact that it can handle each segment individually, enabling more effective memory allocation and deallocation.

4.2.2 The system physical memory

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

A multi-level paging system would be generated if the address was divided by more than two levels in the paging memory management system. This method may be helpful in addressing the problem of physical memory fragmentation. Nevertheless, there may be certain drawbacks to take into attention.

- First off, using multiple levels of page tables may result in an increase in the memory management system's overhead. The page tables need to be maintained by the operating system, and each additional level of page tables calls for more memory to store data.
- Second, a multi-level paging system may delay access times, as the system needs to traverse multiple levels of page tables to translate a virtual address to a physical address. The system performance could be impacted as an effect of the increased access times to memory.

Overall, even though a multi-level paging system can aid address fragmentation in physical memory, it is crucial to carefully consider the potential trade-offs in terms of system overhead and access times.

4.2.3 Paging-based address translation scheme

What is the advantage and disadvantage of segmentation with paging?

The benefits of both segmentation and paging approaches in memory management are combined in segmented paging, sometimes referred to as segmentation with paging.

Advantages of segmentation with paging:

- Flexible memory allocation: Segmented paging enables dynamic memory segment allocation with varied lengths, allowing for flexibility in fitting the memory needs of various processes or applications. The memory may be efficiently allocated for portions of different sizes.
- Sharing and Protection: Segmented paging offers security against unwanted access to memory segments. It enables effective memory page sharing across many programs, improving system efficiency and utilization of resources.
- Reduced External Fragmentation: Segmented paging eliminates the need to deal with external fragmentation, making memory management more efficient. It ensures that memory blocks are of uniform size, preventing the formation of small, scattered free memory blocks.
- Efficient Virtual Memory Swapping: In segmented paging, the entire segment does not need to be swapped out when swapping data into virtual memory. This simplifies the swapping process and makes it more efficient, as only the necessary pages are swapped.

Disadvantages of segmentation with paging:

- Complexity: A complicated memory management strategy called segmentation with paging needs extra hardware support like segment tables and page tables. The overhead and complexity of the memory management system climb as a consequence.
- Overhead: In order to get the segment table and page table entries while using segmentation with paging, extra memory accesses are required. As a consequence of that, memory access overhead goes up and system performance might be negatively impacted.

- Fragmentation within segments: Although segmented paging reduces external fragmentation, it can still lead to fragmentation within individual segments. If a segment is not effectively managed, it may experience internal fragmentation as smaller portions of the segment remain unused.

In conclusion, segmentation with paging offers flexible and efficient memory management tactics, except that it comes with complications and costs. When picking a memory management strategy for a particular system, both pros and cons should be thoroughly considered.

4.3 Put It All Together

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.

The given result is, in principle, different from what we anticipated when the synchronization is not handled by our system. A "race condition" is a circumstance where many processes simultaneously access and alter the same data and the result of the execution depends on the precise sequence in which the access occurs. In order to prevent race circumstances when there are several CPUs, the critical section conditions: Mutual exclusion, Progress, and Bounded waiting must be present. If the mechanism to avoid the race condition is not ensured, data loss will result from overwriting while accessing the queue and dispatching the process. It goes without saying that we don't want the ensuing modifications to interact with one another. So, process synchronization is necessary.

5 Conclusion

This Simple Operating System assignment is designed to provide us with a comprehensive understanding of the major components of a simple operating system. By completing this assignment, we will gain practical experience with three major modules: scheduler, synchronization, and mechanism of memory allocation from virtual-to-physical memory.

Through this assignment, we will learn how to simulate the fundamental concepts of scheduling, synchronization, and memory management in an operating system. So that we are able to understand the role of key modules in an OS and partially comprehend the principles behind a simple OS.

Overall, the Simple Operating System assignment provides an excellent opportunity for students to enhance their knowledge and skills in operating systems. It is a challenging yet rewarding task that will help them prepare for future careers in computer science and engineering.

The source code and the output have already been attached into the zip file. If you have any problem, please contact via email: nam.nguyenolkmphy@hcmut.edu.vn

References

[1] ABRAHAM SILBERSCHATZ - PETER BAER GALVIN - GREG GAGNE, OPERATING SYSTEM CONCEPTS (TENTH EDITION)

[2] GeeksforGeeks, Multilevel Queue (MLQ) CPU Scheduling, https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/

[3] GeeksforGeeks, Multilevel Feedback Queue Scheduling (MLFQ) CPU Scheduling, https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling-mlfq-cpu-scheduling/

[4] GeeksforGeeks, Introduction of Process Synchronization, https://www.geeksforgeeks.org/introduction-of-process-synchronization/

[5] GeeksforGeeks, Memory Management in Operating System, https://www.geeksforgeeks.org/memory-management-in-operating-system/

[6] GeeksforGeeks, Virtual Memory in Operating System, https://www.geeksforgeeks.org/virtual-memory-in-operating-system/

[7] GeeksforGeeks, Memory Hierarchy Design and its Characteristics, https://www.geeksforgeeks.org/memory-hierarchy-design-and-its-characteristics/

[8] GeeksforGeeks, Paged Segmentation and Segmented Paging, https://www.geeksforgeeks.org/paged-segmentation-and-segmented-paging/