VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEM

Project: Simple Operating System

Lecturer: Le Thanh Van

Students: Doan Anh Tien - 1852789 (Class CC03)
Ho Hoang Thien Long - 1852161 (Class CC03)
Bui Hoang Phuc - 1952925 (Class CC02)

HO CHI MINH CITY, 30TH MAY 2021

Member list & Workload

No.	Fullname	Student ID	Problems	Percentage of work
1	Doan Anh Tien	1852789	- Scheduling Question - Scheduling Result - Memory Management Question	100%
2	Ho Hoang Thien Long	1852161	- Scheduling Result - Memory Management Result - Synchronization - Put It All Together	100%
3	Bui Hoang Phuc	1952925	- Scheduling Question - Scheduling Result - Memory Management Question	100%

Contents

1 Scheduler

1.1 Implementation question

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

Answer: The Priority Feedback Queue scheduler is not only based on the Multilevel Feedback Queue, but also inherits some properties from other scheduling algorithms. Specifically, the processes run in several different queues like Multilevel Queue algorithm; use the condition like Priority Scheduling algorithm where each process posses its own priority level for execution; and use the equal burst time for each process like Round Robin algorithm.

Multilevel Feedback Queue

In the Multilevel Feedback Queue algorithm, processes are able to moved between queues based on some criteria:

- If a process does not complete its task on time, it will be moved to a lower-priority queue and let the free slot for other process to come in.
- If a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- The process in a lower-priority queue can only be executed when the other higher-priority queues are **empty**.

It can be said that Multilevel Feedback Queue is an extension of Multilevel Queue algorithm.

Priority Feedback Queue

Based on the Multilevel Feedback Queue, the Priority Feedback Queue algorithm uses two queues which are **ready_queue** and **run_queue**:

- The **ready_queue** and **run_queue** are the priority queues, in which the **ready_queue** is the superior compared to **run_queue**. The queues work based on the priorities of processes waiting inside them, the highest priority process will be picked first, if 2 processes have the same priority value, the FIFO mechanism will be used.
- The **ready_queue** stores all arrived processes and feed them into the CPU for execution based on theirs priorities. CPU will always pick processes from **ready_queue** first since it is the higher-priority queue.
- The **run_queue** stores all processes that have not done its time slot and are waiting to be picked up from CPU for the execution again. These processes can only continue its time slot and pushed back into **ready_queue** when the **ready_queue** is empty (aka free).

Advantages of Priority Feedback Queue

From all the properties mentioned above, we can point out some of the advantages of Priority Feedback Queue:

- The quantum time is used just like Round Robin algorithm, ensuring that there is a fairness between processes in terms of time, avoiding the state where CPU is heavily occupied by one process and starvation[1].
- The multiple queues are used, enabling the transfer between them (incompleted process will be paused and transferred to the **run_queue**) and thus speed up the progress for other process instead of let them waiting.
- The lower-priority process that comes up later can still be execute first as some higher-priority ones are waiting in the **run_queue**, and those higher-priority processes might only be 'back to the work' again until the **ready_queue** is empty.
- The higher-priority process can still be selected first if it is in **ready_queue**.

Comparison with other scheduling algorithms

First Come First Serve (FCFS)

- In FCFS, the process with the less burst time has long waiting time, which may results in large average waiting time
- In PFQ, the process has the limited quantum time so it can leave the slot for other processes, which may results in less average waiting time

Shortest Job First (SJF)

- In SJF, the process with the smallest burst time will be executed first, and the other process with higher burst time will need to wait for the chosen one to be completed. The more shorter process coming, the more chance the starvation event occurs.
- In PFQ, although its mechanism may cause a bias on the higher-priority process, all processes might be executed in a certain fairness due to the quantum time.

Round Robin (RR)

- In RR, the throughput of queue heavily depends on the quantum time. If the quantum time large enough, it may behaves the same as FCFS algorithm
- In PFQ, although using the round robin style in CPU, there still have the priority factor to determine the process to be executed next; and the chance for starvation to occurs is not too much.

[1]The phenomenon where one processes use a lot of CPU time and other process that ready to run might wait in an indefinite period of time because of low priority/queueing mechanism

Priority Scheduling (PS)

- In PS, the processing of lower-priority process might be paused for higher-priority processes to be completed (in Preemptive PS); or the lower-priority process might be wait for the newly higher-priority process to be completed. These events might cause starvation when the lower-priority process waits for an indefinite period of time for its turn.
- In PFQ, the combination of two queues will enable the lower-priority process to be executed soon without waiting for too long, while the higher-priority is waiting in on another queue to be pushed back until the first queue is empty.

Multilevel Queue (MLQ)

- In MLQ, the ready queue is partitioned into several smaller queues with their own fixed priority status. It is not until all higher-priority queue finish all of their processes that the lower-priority one can start its execution. This characteristics of MLQ might cause the starvation.
- In PFQ, the queues are classified as priority queues, in which the process that not complete its burst time will be moved from the higher-priority queue to the other one, letting the other process to take their turns. Even though there are some process waiting in the lower-priority queue until the higher one is empty(like MLQ), their average waiting time is still less compared to MLQ, all due to the appropriate quantum time that has been applied for all of the process. Thanks to combining multilevel queue and round-robin style, PFQ gradually prevents the starvation.

1.2 Result

Requirement: Draw Gantt diagram describing how processes are executed by the CPU.

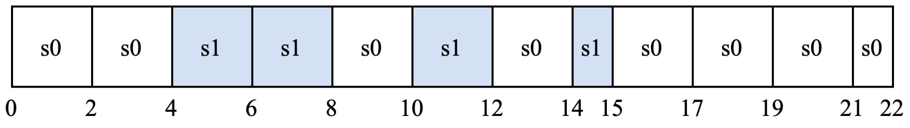
Assumption: Due to the fact the there are many concurrent threads are running, the results may vary, the Gantt chart will follow the theory which maybe different from the real results.

Testcase sched_0 configuration:

- Time slice = 2;
- Number of CPU = 1;
- Number of Processes to be run = 2

	Time start	Priority	Number of instructions
s_0	0	12	15
s_1	4	20	7

Gantt Diagram

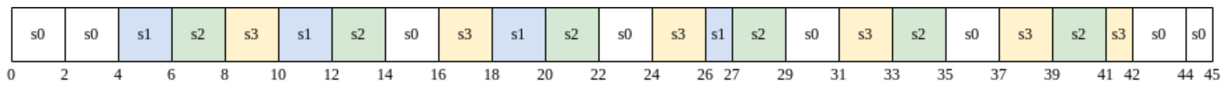


Testcase sched_1 configuration:

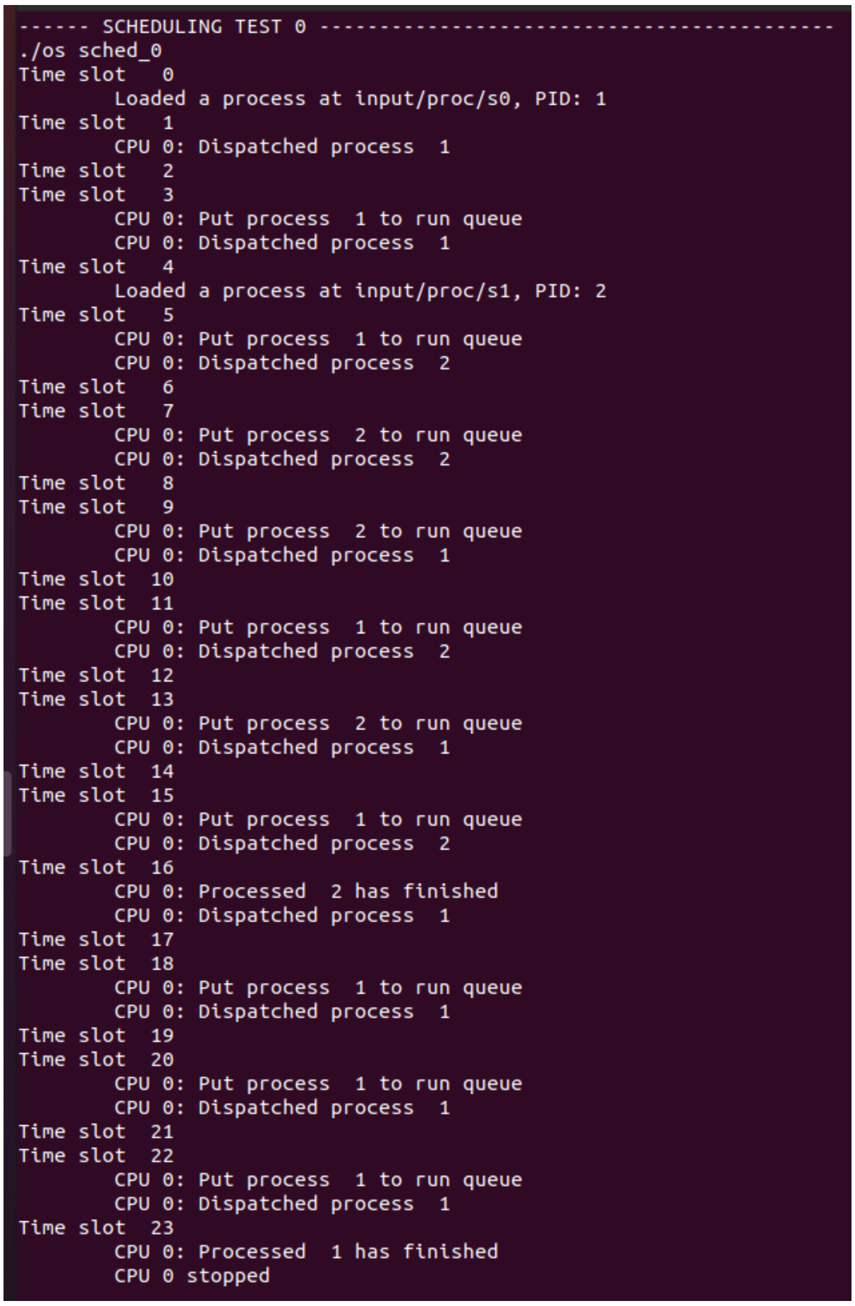
- Time slice = 2;
- Number of CPU = 1;
- Number of Processes to be run = 4

	Time start	Priority	Number of instructions
s_0	0	12	15
s_1	4	20	7
s_2	6	20	12
s_3	7	7	11

Gantt Diagram

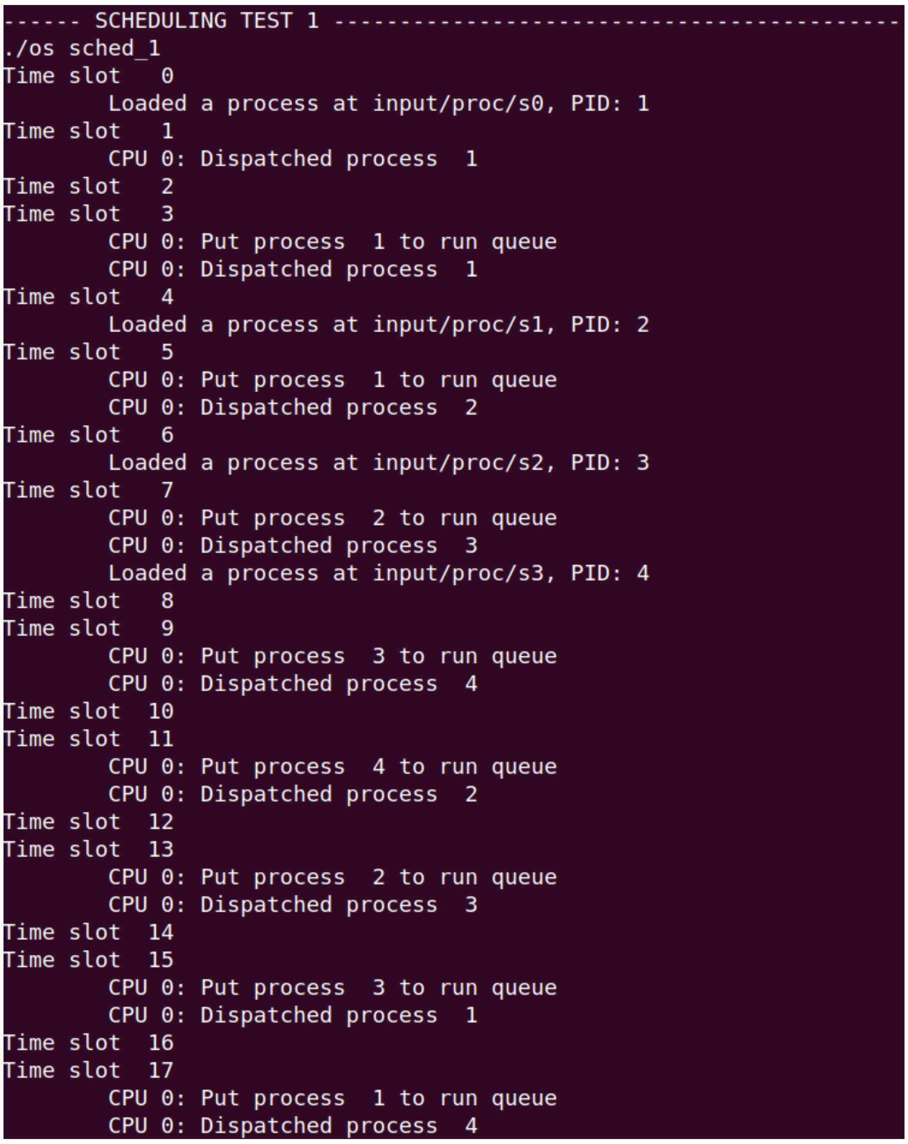


Note: 1 instruction takes 1 time slot to be executed.



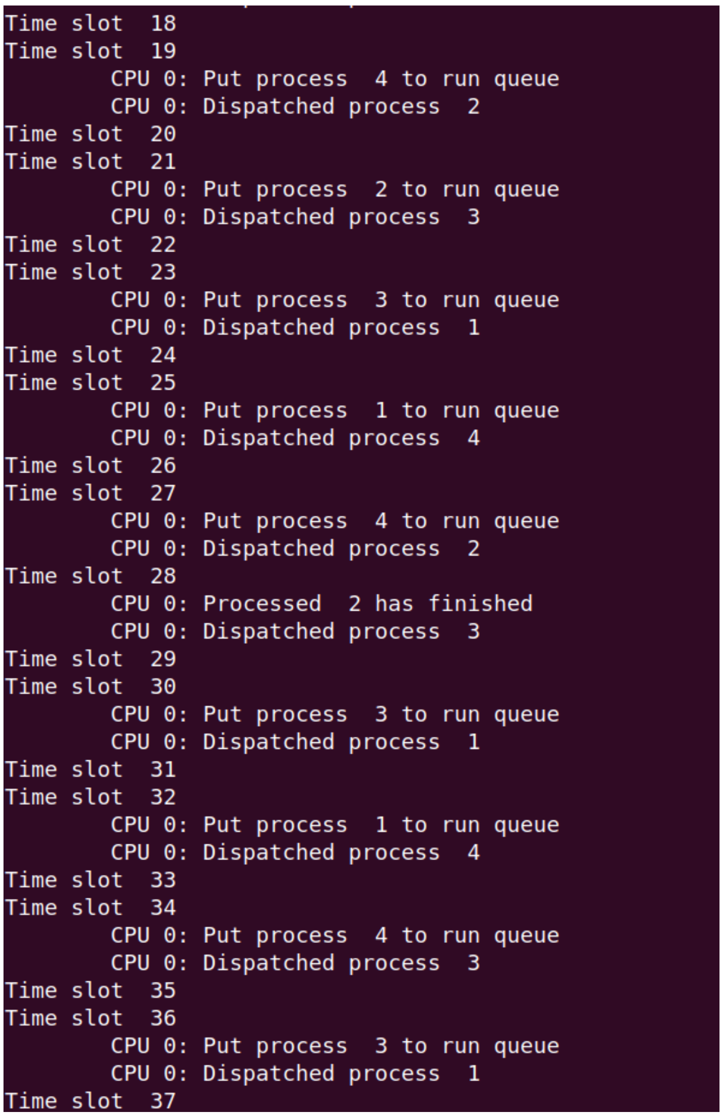
```
------ SCHEDULING TEST 0 -------------------------------------------
./os sched_0
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  10
Time slot  11
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  14
Time slot  15
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot  16
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  17
Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  19
Time slot  20
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  21
Time slot  22
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  23
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Figure 1: The output of sched_0 testcase



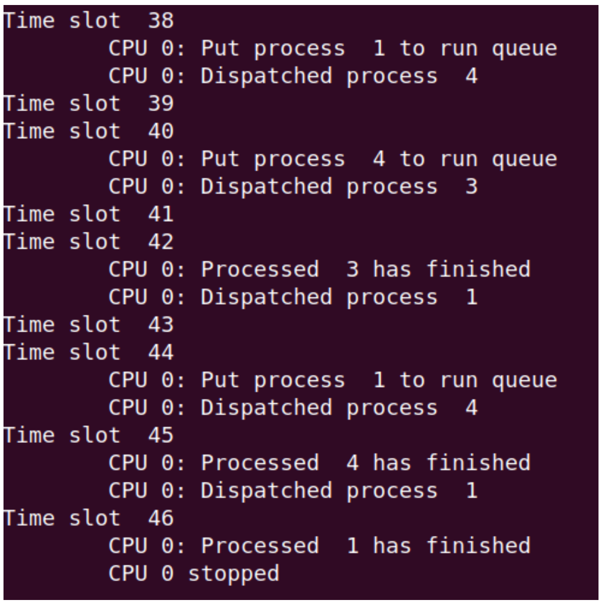
```
------ SCHEDULING TEST 1 -------------------------------------------
./os sched_1
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
        Loaded a process at input/proc/s2, PID: 3
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
        Loaded a process at input/proc/s3, PID: 4
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  10
Time slot  11
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  14
Time slot  15
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  16
Time slot  17
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
```

Figure 2: The output of sched_1 testcase (1)



```
Time slot  18
Time slot  19
        CPU 0: Put process  4 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  3
Time slot  22
Time slot  23
        CPU 0: Put process  3 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  4
Time slot  26
Time slot  27
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  28
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
Time slot  29
Time slot  30
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  31
Time slot  32
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  33
Time slot  34
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  35
Time slot  36
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  37
```

Figure 3: The output of sched_1 testcase (2)



```
Time slot  38
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  39
Time slot  40
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  41
Time slot  42
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  1
Time slot  43
Time slot  44
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  45
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  46
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Figure 4: The output of sched_1 testcase (3)

2 Memory Management

2.1 Implementation question

Question: What is the advantage and disadvantage of segmentation with paging?

Answer: Before going to answer the required question, we manage to interpret the concept and mechanism of paging and segmentation.

Paging

Paging is a memory management method that divides the logical memory into blocks with a fixed size called pages, and at the same time divides the physical memory into blocks called frames with a same size of pages. This scheme demonstrates how the system placing the page into the corresponding physical frame, which works out with the help of address translation and page table.

For each process, there will be a logical address containing two information:

- Page number
- Offset

For each physical address, it also contain two information:

- Frame number
- Offset

The mission of the page table is mapping each page inside it with the corresponding base address (frame number) in the physical memory. This base address will combine with the offset from logical address to form a physical address, which will be finalized as the location that the process will be placed into.

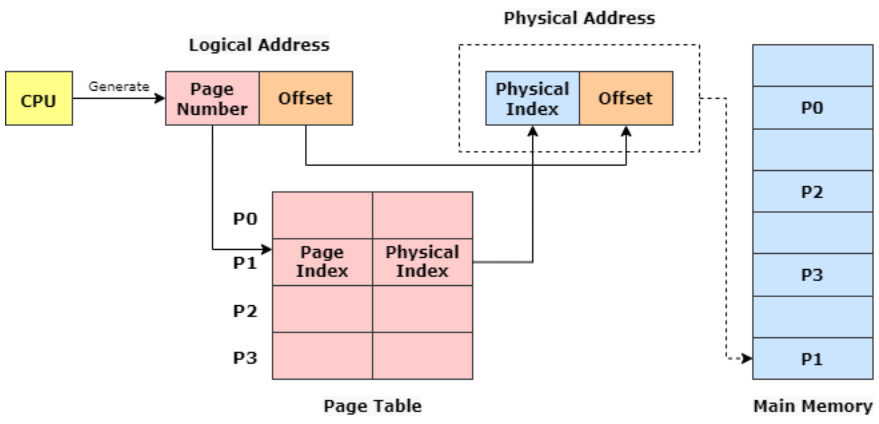


Figure 5: Paging mechanism

Segmentation

In paging memory management technique, it may divide the same function of a process into different pages and those pages may or may not be loaded at the same time into the memory. Therefore, it might decreases the efficiency of the system.

It is better to have segmentation which divides the process into the segments that represent each of functions or parts of the process. Segmentation is a memory management method that divides memory into variable size parts called segments to allocate processes.

For each process, there will be a logical address containing two information:

- Segment number
- Offset

The mission of the segment table is similar to page table, in which it mainly maps each segment inside it with the corresponding base address in the physical memory. This base address will combine with the offset from logical address to form a physical address. However, segment table have a value limit to distinguish it from page table. Each segment contains two information:

- The base address
- The length (limit)

in which the offset value will be compared with limit; and if the offset is smaller than limit, the base address is valid and can be converted to physical address; otherwise, the system will throws an error as the address is invalid.

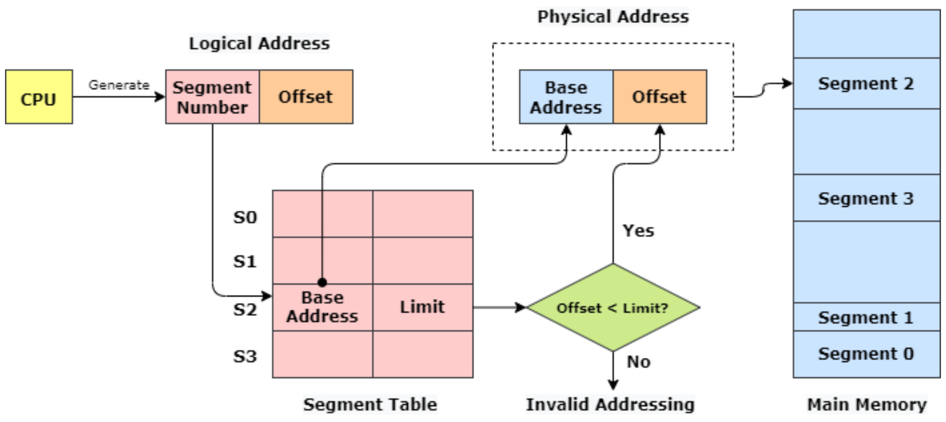


Figure 6: Segmentation mechanism

Segmentation with Paging

Segmented Paging is a technique that combines both Segmentation and Paging features. Memory in Segmented Paging will be divided into variable sized segments and further into fixed sized pages. The logical address of the process will be represented by Segment number (in this project will be called Segment index), Page number (in this project will be called Page index) and Page offset[Le21].

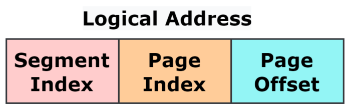


Figure 7: Logical address of a process

Specifically, one process will be fragmented into different segments that will be placed as entries in a segment table:

- Each entry provides a segment index and a pointer to a page table
- The size of page tables inside the segment are limited

From the segment table, page table is located by segment pointer and then it uses the page index to reveal the corresponding page. Also, the acquired page table contains several entries representing the number of pages divided from a segment:

- Each entry provides a page index and a physical index in main memory
- Since a process is divided into segments, one process may have multiple page tables

From the page table, physical index is located and combined with the page offset to form the physical address in main memory (similar to paging mechanism).

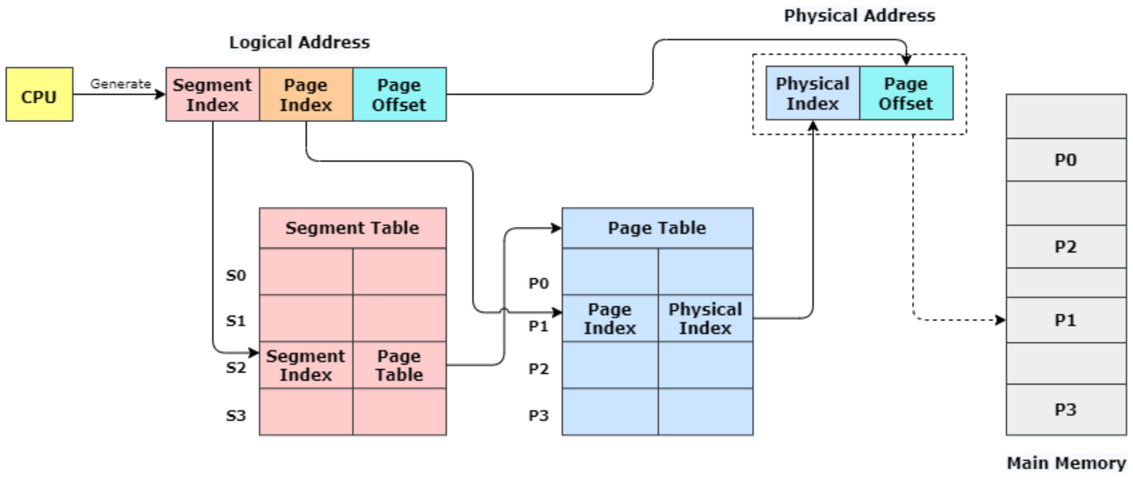


Figure 8: Segmentation with paging mechanism

Note: As we use the fixed-size pages, there is no need for the limit checking like segmentation mechanism.

In this assignment context: the segmentation with paging is used when a process do allocation and deallocation memory regions. Each process have it own segment table of data; each alloc() call will have it own segments, which means the 2 alloc() call cannot used the same segment; the logical address of each inner page is translated to physical memory page.

Advantages of Segmentation with Paging

- The problem of external fragmentation in physical memory is solved because all pages have the fixed size and equal to frames'. There will be no case where the allocating data is larger than unused memory or the amount of unused blocks combined together.
- The segment table contains only one entry corresponding to each segment.
- The memory allocation is simple due to inheriting features from paging mechanism.
- The translation time to physical address is faster than pure paging by accessing segment index first to get the corresponding page index and physical index, rather than iterating all entries of the one huge page table.

Disadvantages

- The internal fragmentation problem still occurs due to paging mechanism.
- The complexity of implementation is much higher as compared to pure paging or segmentation.

2.2 Result

Requirement: Show the status of RAM after each memory allocation and de-allocation call.

We will show 2 tables:

- _mem_stat: show physical memory status.
- logical_mem_stat: show the list of the first byte logical address of certain allocated memory regions.

_mem_stat legends:

- page No.: The global page index of the physical memory.
- PID: The process that allocate that page.
- index: Local page index within the certain alloc command.
- next page No.: The next global page index.

logical_mem_stat legends:

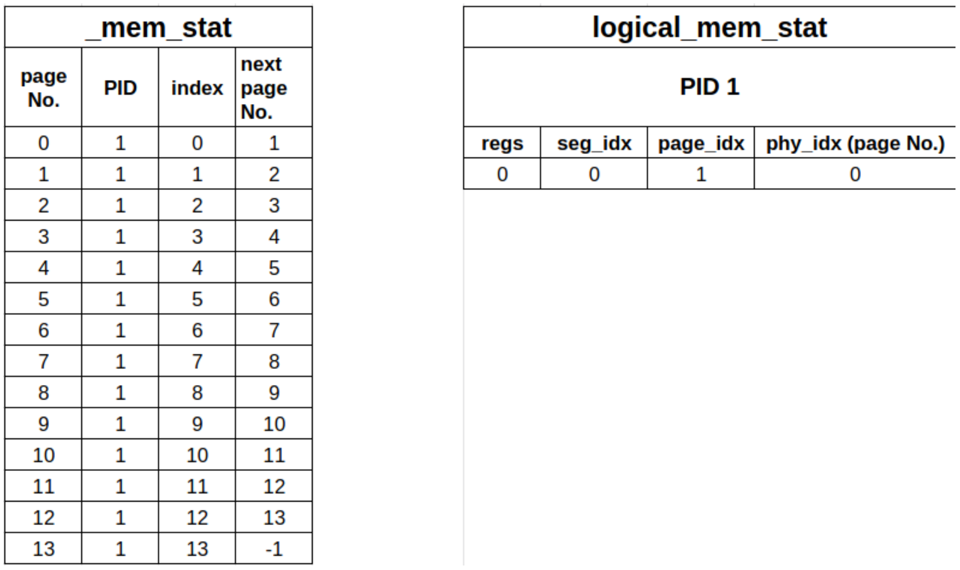
- regs: register that store the first byte logical address.
- seg_idx: segment index of the first byte logical address.
- page_idx: page index of the first byte logical address.
- phy_idx: translated physical page No.

RAM is empty initially.

Note:

- The allocated address starts at 0x400 (one page) after the address 0x0 which is reserved for OS.
- Every alloc() call will get it own segment entries depending on the number of pages.

m0 testcase



_mem_stat			
page No.	PID	index	next page No.
0	1	0	1
1	1	1	2
2	1	2	3
3	1	3	4
4	1	4	5
5	1	5	6
6	1	6	7
7	1	7	8
8	1	8	9
9	1	9	10
10	1	10	11
11	1	11	12
12	1	12	13
13	1	13	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
0	0	1	0

Figure 9: RAM status after **alloc 13535 0**

We need to allocate $ceiling(13535/2^{10}) = 14(pages)$ and store the first byte address to register 0.

_mem_stat			
page No.	PID	index	next page No.
0	1	0	1
1	1	1	2
2	1	2	3
3	1	3	4
4	1	4	5
5	1	5	6
6	1	6	7
7	1	7	8
8	1	8	9
9	1	9	10
10	1	10	11
11	1	11	12
12	1	12	13
13	1	13	-1
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
0	0	1	0
1	1	1	14

Figure 10: RAM status after **alloc 1568 1**

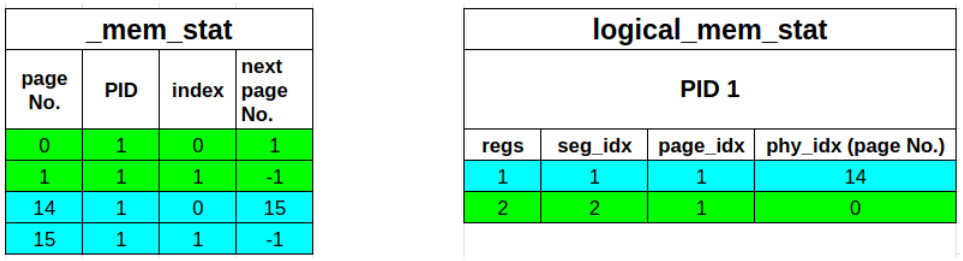
We need to allocate $ceiling(1568/2^{10}) = 2(pages)$ and store the first byte address to register 1.

_mem_stat			
page No.	PID	index	next page No.
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
1	1	1	14

Figure 11: RAM status after **free 0**

Free memory regions starting from the first byte of virtual address stored in register 0.

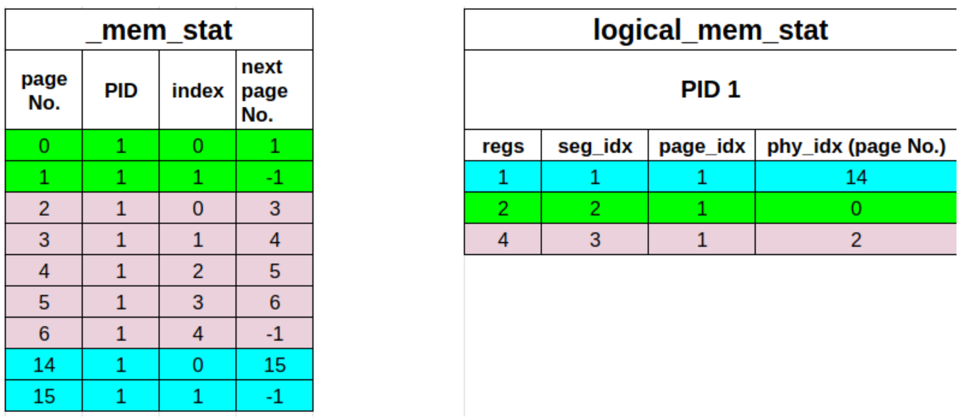


_mem_stat			
page No.	PID	index	next page No.
0	1	0	1
1	1	1	-1
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
1	1	1	14
2	2	1	0

Figure 12: RAM status after **alloc 1386 2**

We need to allocate $ceiling(1386/2^{10}) = 2(pages)$ and store the first byte of virtual address to register 2.



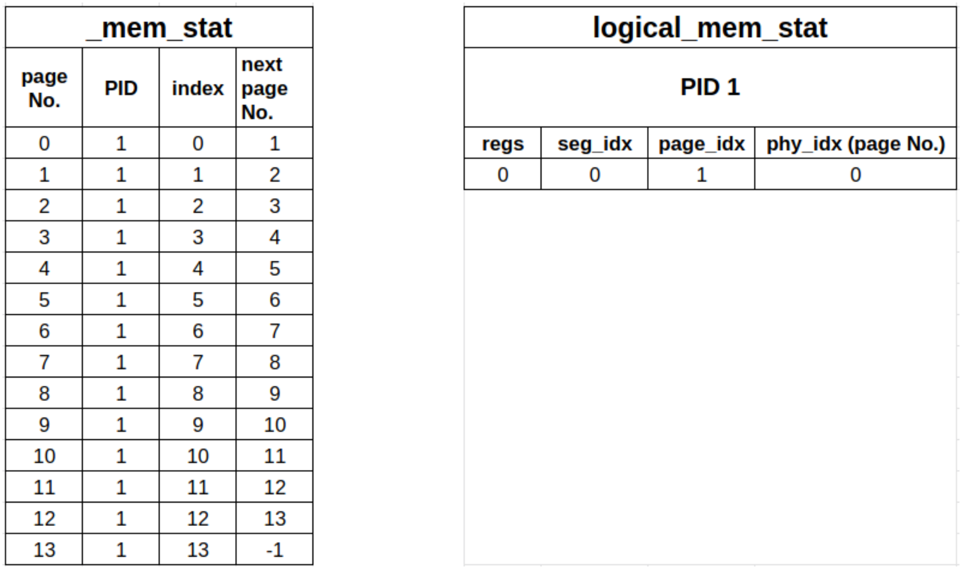
_mem_stat			
page No.	PID	index	next page No.
0	1	0	1
1	1	1	-1
2	1	0	3
3	1	1	4
4	1	2	5
5	1	3	6
6	1	4	-1
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
1	1	1	14
2	2	1	0
4	3	1	2

Figure 13: RAM status after **alloc 4564 4**

We need to allocate $ceiling(4564/2^{10}) = 5(pages)$ and store the first byte of virtual address to register 4.

m1 testcase



_mem_stat			
page No.	PID	index	next page No.
0	1	0	1
1	1	1	2
2	1	2	3
3	1	3	4
4	1	4	5
5	1	5	6
6	1	6	7
7	1	7	8
8	1	8	9
9	1	9	10
10	1	10	11
11	1	11	12
12	1	12	13
13	1	13	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
0	0	1	0

Figure 14: RAM status after **alloc 13535 0**

We need to allocate $ceiling(13535/2^{10}) = 14(pages)$ and store the first byte of virtual address to register 0.

_mem_stat			
page No.	PID	index	next page No.
0	1	0	1
1	1	1	2
2	1	2	3
3	1	3	4
4	1	4	5
5	1	5	6
6	1	6	7
7	1	7	8
8	1	8	9
9	1	9	10
10	1	10	11
11	1	11	12
12	1	12	13
13	1	13	-1
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
0	0	1	0
1	1	1	14

Figure 15: RAM status after **alloc 1568 1**

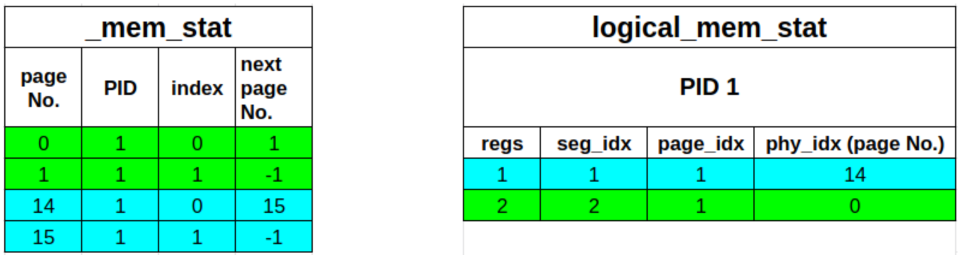
We need to allocate $ceiling(1568/2^{10}) = 2(pages)$ and store the first byte of virtual address to register 1.

_mem_stat			
page No.	PID	index	next page No.
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
1	1	1	14

Figure 16: RAM status after **free 0**

Free memory regions starting from the first byte address stored in register 0.

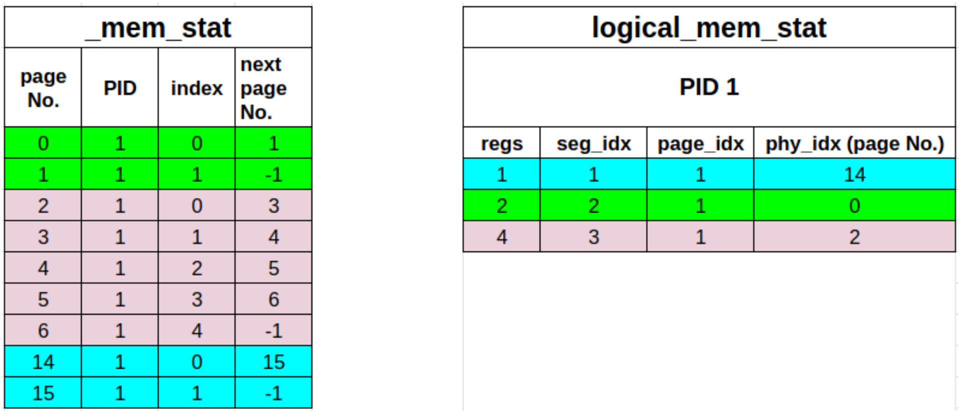


_mem_stat			
page No.	PID	index	next page No.
0	1	0	1
1	1	1	-1
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
1	1	1	14
2	2	1	0

Figure 17: RAM status after **alloc 1386 2**

We need to allocate $ceiling(1386/2^{10}) = 2(pages)$ and store the first byte of virtual address to register 2.

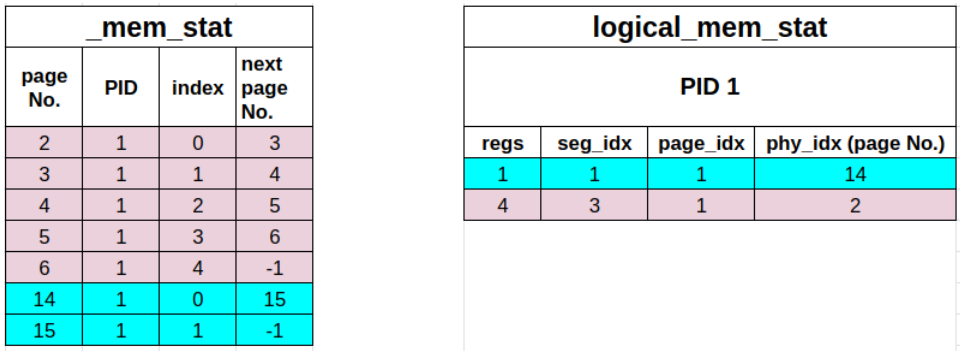


_mem_stat			
page No.	PID	index	next page No.
0	1	0	1
1	1	1	-1
2	1	0	3
3	1	1	4
4	1	2	5
5	1	3	6
6	1	4	-1
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
1	1	1	14
2	2	1	0
4	3	1	2

Figure 18: RAM status after **alloc 4564 4**

We need to allocate $ceiling(4564/2^{10}) = 5(pages)$ and store the first byte of virtual address to register 4.

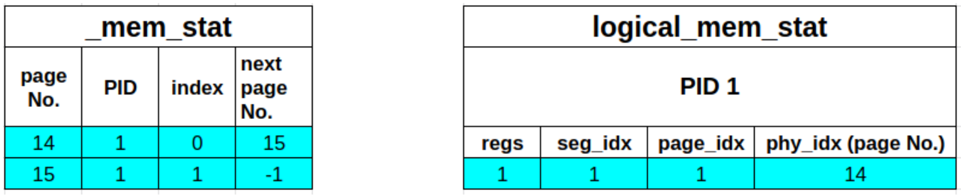


_mem_stat			
page No.	PID	index	next page No.
2	1	0	3
3	1	1	4
4	1	2	5
5	1	3	6
6	1	4	-1
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
1	1	1	14
4	3	1	2

Figure 19: RAM status after **free 2**

Free memory regions starting from the first byte of virtual address stored in register 2.

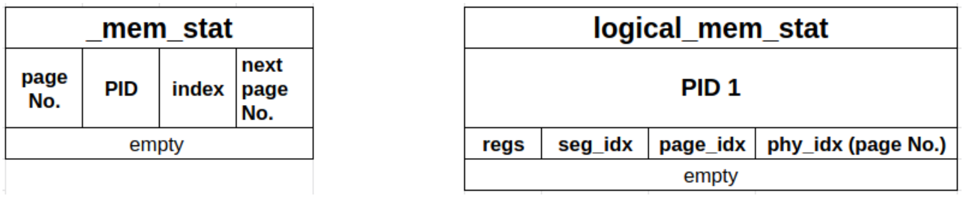


_mem_stat			
page No.	PID	index	next page No.
14	1	0	15
15	1	1	-1

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
1	1	1	14

Figure 20: RAM status after **free 4**

Free memory regions starting from the first byte of virtual address stored in register 4.

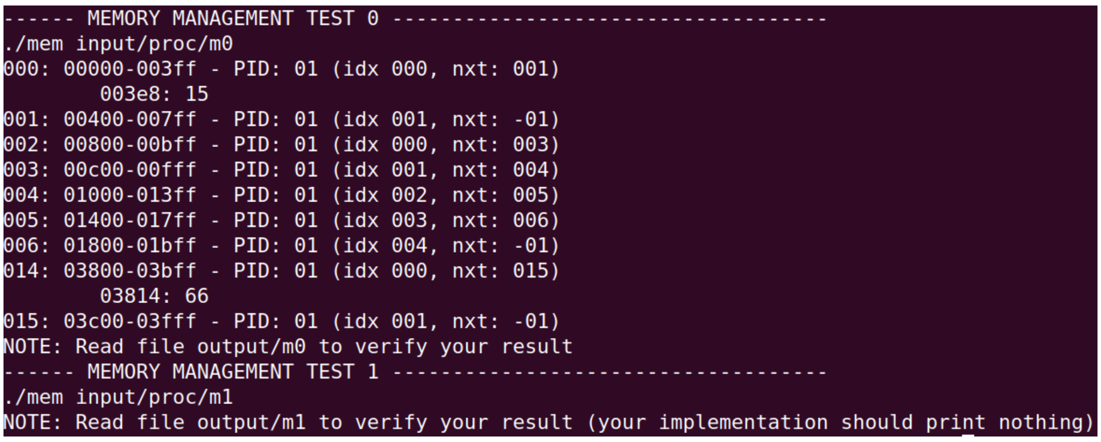


_mem_stat			
page No.	PID	index	next page No.
empty			

logical_mem_stat			
PID 1			
regs	seg_idx	page_idx	phy_idx (page No.)
empty			

Figure 21: RAM status after **free 1**

Free memory regions starting from the first byte of virtual address stored in register 1.



```
------ MEMORY MANAGEMENT TEST 0 -----------------------------------
./mem input/proc/m0
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
        003e8: 15
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
        03814: 66
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
NOTE: Read file output/m0 to verify your result
------ MEMORY MANAGEMENT TEST 1 -----------------------------------
./mem input/proc/m1
NOTE: Read file output/m1 to verify your result (your implementation should print nothing)
```

Figure 22: RAM status after finishing the 2 testcases

3 Synchronization

3.1 Implementation

Using 2 mutex locks to control the access of multiple CPUs to shared memory regions

- **queue_lock** - protect the queue of processes: Whenever the OS calls enqueue() or dequeue() methods when doing scheduling.
- **mem_lock** - protect the physical as well as virtual memory: Whenever the processes call alloc(), free(), read(), write().

4 Put It All Together

Requirement: Student find their own way to interpret the result of simulation.

4.1 Overall description

Assumptions: Due to the fact the the processes run concurrently, there will be ordered differences among different executions. The result below is the one that has smallest differences with the sample result.

Overall description: Firstly, the OS will load the config file specifying time slice, number of CPUs and processes. Secondly, the OS will load the processes to priority feedback queues handling scheduler. For each process, there is a number of instructions that need to be executed. The OS will load each process to the CPU, CPU then process one instruction for each time slot. There are some instructions need memory allocation, so the memory management mechanism is needed, which is Segmentation with Paging mechanism.

Result overall description: The results show 2 parts, first one for scheduling log for each time slot, the second one show the final physical memory status after the processes are finished.

4.2 os_0 testcase

Testcase os_0 configuration:

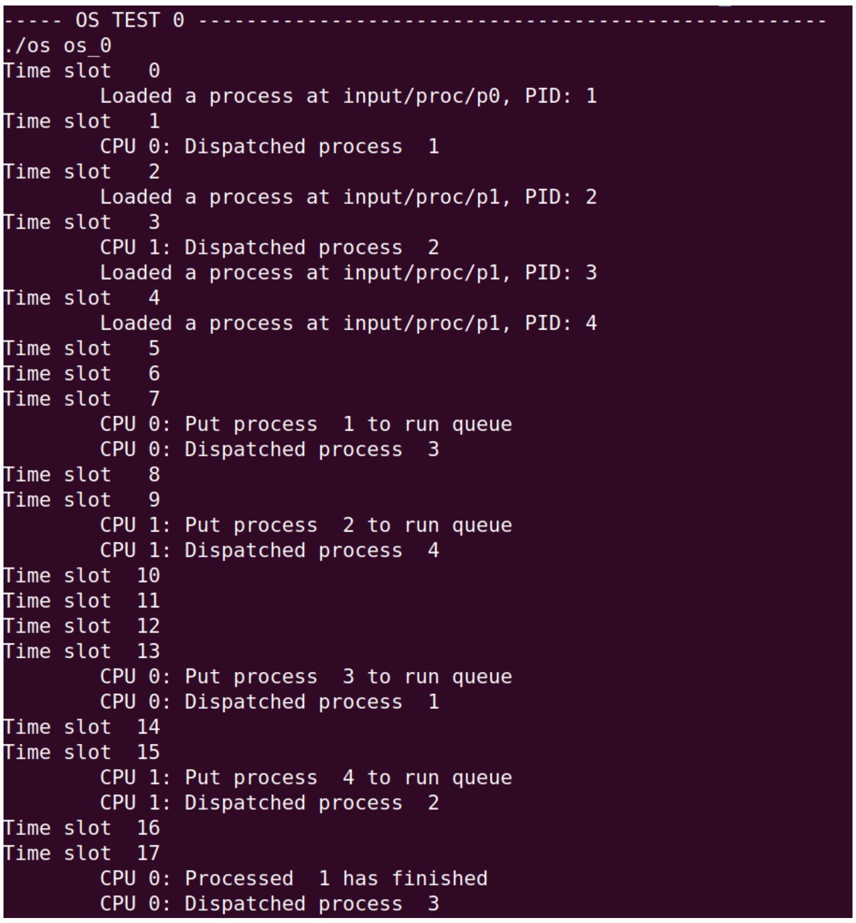
- Time slice = 2
- Number of CPU = 1
- Number of Processes to be run = 4

	Time start	Priority	Number of instructions
s_0	0	12	15
s_1	4	20	7

4.2.1 Result

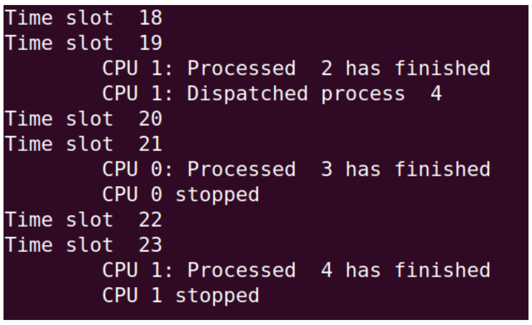
There are some main points in the result

- There are only 2 processes but the configurations file declares that there are 4 processes needed to be run. The OS load process *p0* 1 time as PID 1, load *p1* 3 times as PID 2, 3, 4; as we can see from time slot 0 to 4 in **Figure 23**.
- In the final memory contents, specifically the write() call that write the value 0a also show that there are 3 *p1* processes are executed in **Figure 25**



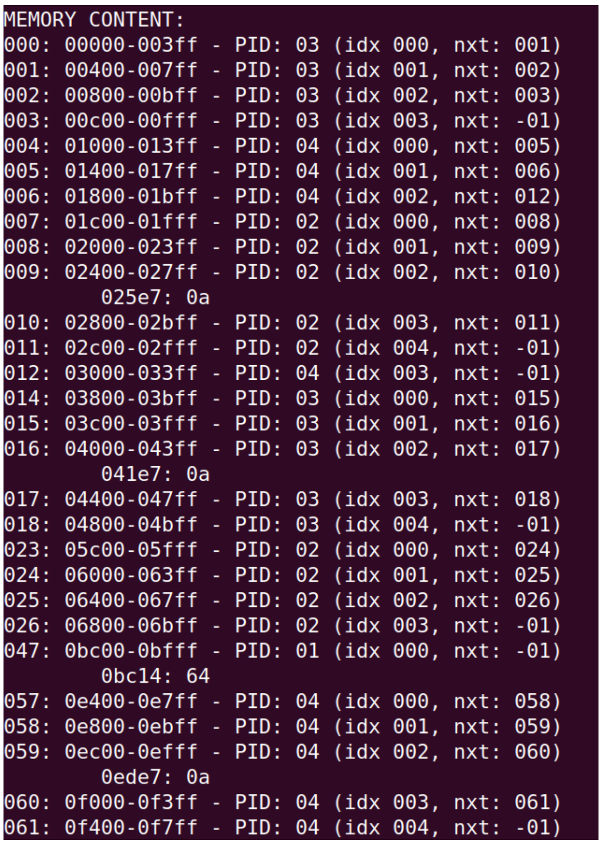
```
----- OS TEST 0 -----------------------------------------------------
./os os_0
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/p1, PID: 2
Time slot   3
        CPU 1: Dispatched process  2
        Loaded a process at input/proc/p1, PID: 3
Time slot   4
        Loaded a process at input/proc/p1, PID: 4
Time slot   5
Time slot   6
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot   8
Time slot   9
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  4
Time slot  10
Time slot  11
Time slot  12
Time slot  13
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  14
Time slot  15
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  2
Time slot  16
Time slot  17
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  3
```

Figure 23: sched result of os_0 (1)



```
Time slot  18
Time slot  19
        CPU 1: Processed  2 has finished
        CPU 1: Dispatched process  4
Time slot  20
Time slot  21
        CPU 0: Processed  3 has finished
        CPU 0 stopped
Time slot  22
Time slot  23
        CPU 1: Processed  4 has finished
        CPU 1 stopped
```

Figure 24: sched result of os_0 (2)



```
MEMORY CONTENT:
000: 00000-003ff - PID: 03 (idx 000, nxt: 001)
001: 00400-007ff - PID: 03 (idx 001, nxt: 002)
002: 00800-00bff - PID: 03 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 03 (idx 003, nxt: -01)
004: 01000-013ff - PID: 04 (idx 000, nxt: 005)
005: 01400-017ff - PID: 04 (idx 001, nxt: 006)
006: 01800-01bff - PID: 04 (idx 002, nxt: 012)
007: 01c00-01fff - PID: 02 (idx 000, nxt: 008)
008: 02000-023ff - PID: 02 (idx 001, nxt: 009)
009: 02400-027ff - PID: 02 (idx 002, nxt: 010)
        025e7: 0a
010: 02800-02bff - PID: 02 (idx 003, nxt: 011)
011: 02c00-02fff - PID: 02 (idx 004, nxt: -01)
012: 03000-033ff - PID: 04 (idx 003, nxt: -01)
014: 03800-03bff - PID: 03 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 03 (idx 001, nxt: 016)
016: 04000-043ff - PID: 03 (idx 002, nxt: 017)
        041e7: 0a
017: 04400-047ff - PID: 03 (idx 003, nxt: 018)
018: 04800-04bff - PID: 03 (idx 004, nxt: -01)
023: 05c00-05fff - PID: 02 (idx 000, nxt: 024)
024: 06000-063ff - PID: 02 (idx 001, nxt: 025)
025: 06400-067ff - PID: 02 (idx 002, nxt: 026)
026: 06800-06bff - PID: 02 (idx 003, nxt: -01)
047: 0bc00-0bfff - PID: 01 (idx 000, nxt: -01)
        0bc14: 64
057: 0e400-0e7ff - PID: 04 (idx 000, nxt: 058)
058: 0e800-0ebff - PID: 04 (idx 001, nxt: 059)
059: 0ec00-0efff - PID: 04 (idx 002, nxt: 060)
        0ede7: 0a
060: 0f000-0f3ff - PID: 04 (idx 003, nxt: 061)
061: 0f400-0f7ff - PID: 04 (idx 004, nxt: -01)
```

Figure 25: memory status of os_0

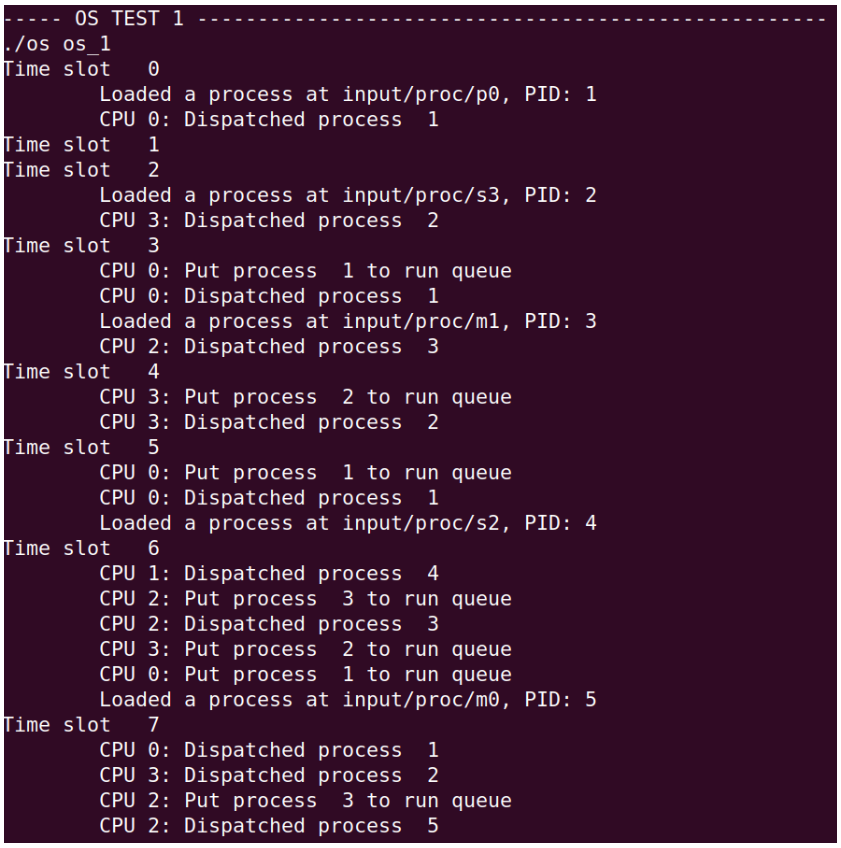
4.3 os_1 testcase

Testcase os_1 configuration:

- Time slice = 2
- Number of CPU = 4
- Number of Processes to be run = 8

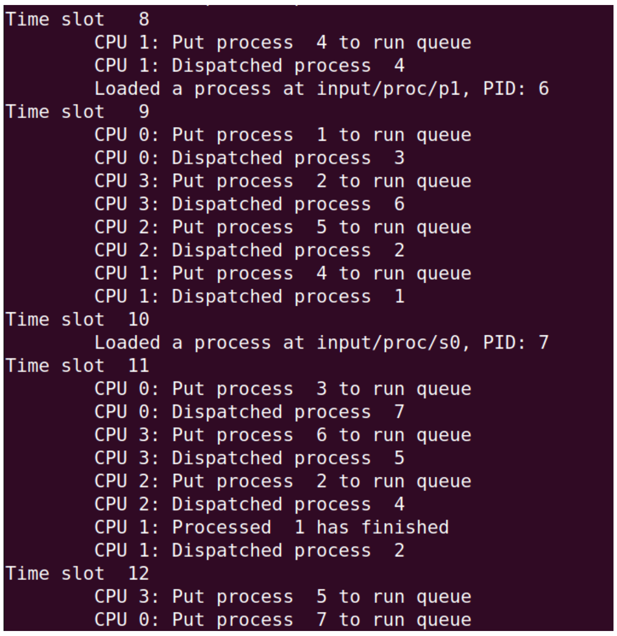
	Time start	Priority	Number of instructions
p_0	1	1	10
s_3	2	7	11
m_1	4	1	8
s_2	6	20	12
m_0	7	1	7
p_1	9	1	10
s_0	11	12	15
s_1	16	20	7

4.3.1 Result



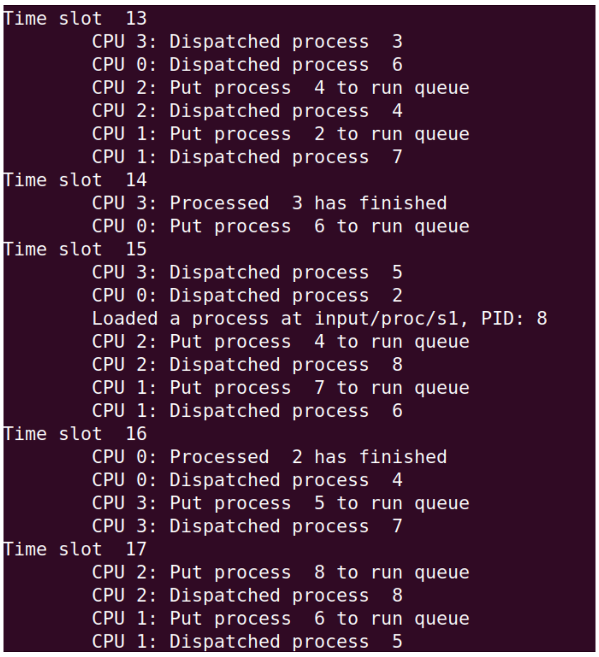
```
----- OS TEST 1 ---------------------------------------------------
./os os_1
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
        CPU 0: Dispatched process  1
Time slot   1
Time slot   2
        Loaded a process at input/proc/s3, PID: 2
        CPU 3: Dispatched process  2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/m1, PID: 3
        CPU 2: Dispatched process  3
Time slot   4
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s2, PID: 4
Time slot   6
        CPU 1: Dispatched process  4
        CPU 2: Put process  3 to run queue
        CPU 2: Dispatched process  3
        CPU 3: Put process  2 to run queue
        CPU 0: Put process  1 to run queue
        Loaded a process at input/proc/m0, PID: 5
Time slot   7
        CPU 0: Dispatched process  1
        CPU 3: Dispatched process  2
        CPU 2: Put process  3 to run queue
        CPU 2: Dispatched process  5
```

Figure 26: sched result of os_1 (1)



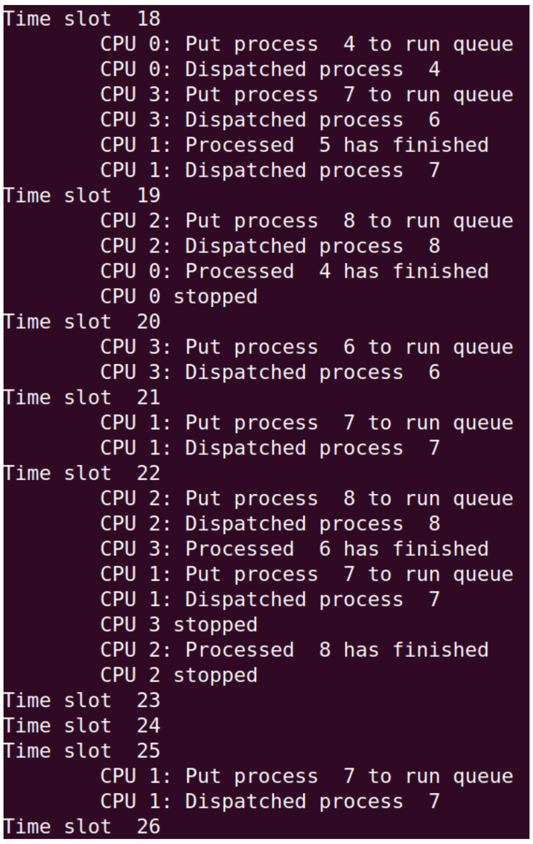
```
Time slot   8
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
        Loaded a process at input/proc/p1, PID: 6
Time slot   9
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  6
        CPU 2: Put process  5 to run queue
        CPU 2: Dispatched process  2
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  1
Time slot  10
        Loaded a process at input/proc/s0, PID: 7
Time slot  11
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  7
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  5
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  4
        CPU 1: Processed  1 has finished
        CPU 1: Dispatched process  2
Time slot  12
        CPU 3: Put process  5 to run queue
        CPU 0: Put process  7 to run queue
```

Figure 27: sched result of os_1 (2)



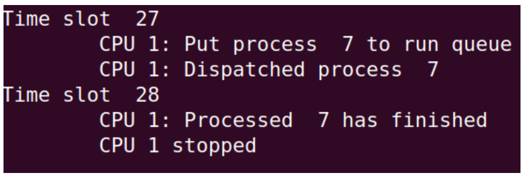
```
Time slot  13
        CPU 3: Dispatched process  3
        CPU 0: Dispatched process  6
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  7
Time slot  14
        CPU 3: Processed  3 has finished
        CPU 0: Put process  6 to run queue
Time slot  15
        CPU 3: Dispatched process  5
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/s1, PID: 8
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  8
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  6
Time slot  16
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
        CPU 3: Put process  5 to run queue
        CPU 3: Dispatched process  7
Time slot  17
        CPU 2: Put process  8 to run queue
        CPU 2: Dispatched process  8
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  5
```

Figure 28: sched result of os_1 (3)



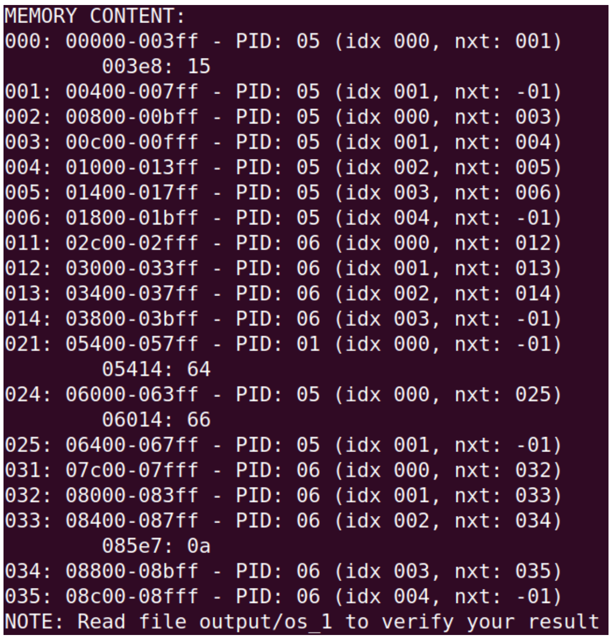
```
Time slot  18
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  6
        CPU 1: Processed  5 has finished
        CPU 1: Dispatched process  7
Time slot  19
        CPU 2: Put process  8 to run queue
        CPU 2: Dispatched process  8
        CPU 0: Processed  4 has finished
        CPU 0 stopped
Time slot  20
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
Time slot  21
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  22
        CPU 2: Put process  8 to run queue
        CPU 2: Dispatched process  8
        CPU 3: Processed  6 has finished
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 3 stopped
        CPU 2: Processed  8 has finished
        CPU 2 stopped
Time slot  23
Time slot  24
Time slot  25
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  26
```

Figure 29: sched result of os_1 (4)



```
Time slot  27
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  28
        CPU 1: Processed  7 has finished
        CPU 1 stopped
```

Figure 30: sched result of os_1 (5)



```
MEMORY CONTENT:
000: 00000-003ff - PID: 05 (idx 000, nxt: 001)
        003e8: 15
001: 00400-007ff - PID: 05 (idx 001, nxt: -01)
002: 00800-00bff - PID: 05 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 05 (idx 001, nxt: 004)
004: 01000-013ff - PID: 05 (idx 002, nxt: 005)
005: 01400-017ff - PID: 05 (idx 003, nxt: 006)
006: 01800-01bff - PID: 05 (idx 004, nxt: -01)
011: 02c00-02fff - PID: 06 (idx 000, nxt: 012)
012: 03000-033ff - PID: 06 (idx 001, nxt: 013)
013: 03400-037ff - PID: 06 (idx 002, nxt: 014)
014: 03800-03bff - PID: 06 (idx 003, nxt: -01)
021: 05400-057ff - PID: 01 (idx 000, nxt: -01)
        05414: 64
024: 06000-063ff - PID: 05 (idx 000, nxt: 025)
        06014: 66
025: 06400-067ff - PID: 05 (idx 001, nxt: -01)
031: 07c00-07fff - PID: 06 (idx 000, nxt: 032)
032: 08000-083ff - PID: 06 (idx 001, nxt: 033)
033: 08400-087ff - PID: 06 (idx 002, nxt: 034)
        085e7: 0a
034: 08800-08bff - PID: 06 (idx 003, nxt: 035)
035: 08c00-08fff - PID: 06 (idx 004, nxt: -01)
NOTE: Read file output/os_1 to verify your result
```

Figure 31: memory status of os_1

5 Evaluation and conclusion

5.1 Evaluation

5.1.1 Scheduling Component

The scheduling algorithm in this operating system project has shown its efficient characteristics in terms of reducing average waiting time and starvation's occurrence. It can be seen from the *sched* and *os* testcases that every process will be dispatched into CPUs for a approximately equal time interval, and follow the Gantt chart that we have constructed theoretically.

Nevertheless, there is a trade off between it efficiency and high complexity, high resource. Our implementation spending resources to store the 2 queues; the action dequeue() take $O(n^2)$ (where n is the queue size) time complexity + feedback complexity between 2 queues.

Finally, we have successfully built this scheduling model.

5.1.2 Memory Management

The segmentation with paging mechanism has reduced the searching time for a physical frame by dividing the logical address into 3 parts, as well as separated each alloc() memory regions clearly by giving them their own segments and increase the security of each process's memory regions.

However, by giving each memory region the whole segment, there are a number of free logical address values cannot be used by other regions.

Freeing memory regions takes less time by just iterating the segment table then deleting the whole segment according to that region, if not, we have to iterate the segment table, then iterating each page table and delete every page entry according to that region.

The break pointer has not been well handling, we just decrease the break pointer value when the region that needs to be deleted account for the last segment, other regions deletions do not effect the break pointer value. Ideally speaking, the break pointer should decrease every time a certain region is freed.

Finally, we have successfully built the physical memory allocation and the logical one. However, the break pointer has not been handled appropriately.

5.1.3 Synchronization

We uses 2 mutex locks to protect the queues and logical, physical memory space to allow only 1 CPU accessing to these regions at a time.

5.2 Conclusion

The scope of this assignment is to build 3 simply basic mechanism of the OS, CPU scheduling, memory management and synchronization.

For CPU scheduling part, we have understand the mechanism behind and recognized the benefit of the mechanism compared to other scheduling mechanism that we have learnt so far. We have implemented successfully the Priority feedback queue to schedule processes executed by the CPU.

For memory management part, we have understand the mechanism behind and recognized the advantages and disadvantages of Segmentation with Paging mechanism. We have implemented partly successfully the Segmentation with Paging mechanism to allocate data memory regions requested by processes, mapping logical to physical address. However, when allocate memory regions, the break pointer have not been appropriately handled in terms of its meaning, but it do not effect the results of the testcases.

For synchronization part, We uses 2 mutex locks to protect the queues and logical, physical memory space to allow only 1 CPU accessing to these regions at a time.

We will try to understand the break pointer manipulation in the future if has the chance.

References

[Le21] T.Van Le. *Simple Operating System*. 2021.