

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



OPERATING SYSTEM (CO2018)

Assignment

Simple Operating System

Advisor: Lê Thanh Vân

Students:	Nguyễn Hữu Trùng Dương	2052929 - CC01
	Nguyễn Hoàng Anh Thư	2053478 - CC01
	Nguyễn Phước Nguyên Phúc	2053342 - CC01

HO CHI MINH CITY, DECEMBER 2022

Member list & Workload

No.	Fullname	Student ID	Task
1	Nguyễn Hữu Trùng Dương	2052929	- Scheduler
2	Nguyễn Hoàng Anh Thư	2053478	- Memory Management
3	Nguyễn Phước Nguyên Phúc	2053342	- Memory Management

Contents

1 Scheduler	3
1.1 Answer the question	3
1.2 Implementation	4
1.3 Result	5
1.3.1 Testcase sched_0:	5
1.3.2 Testcase sched_1:	6
2 Memory Management	10
2.1 Answer the question	10
2.2 Implementation	11
2.3 Result	15
2.4 Explanantion	15
3 Put It All Together	17
3.1 Result:	19
3.1.1 Os_mlq_0:	19
3.1.2 Os_mlq_1	20
3.1.3 Os_mlq_2:	24
4 Conclusion	27

1 Scheduler

1.1 Answer the question

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

Answer:

Priority Queue (PQ) is a process scheduling algorithm based on priority where the scheduler selects tasks according to priority. Thus, processes with higher priority execute first followed by processes with lower priorities.

If two jobs have the same priorities then the process that should execute first is chosen on the basis of **round-robin** or **FCFS**. Which process should have what priority depends on a process' memory requirements, time requirements, the ratio of I/O burst to CPU burst, etc.

Advantages of Priority Queue:

- Easy to use.
- Processes with higher priority execute first which saves time.
- The importance of each process is precisely defined.
- A good algorithm for applications with fluctuating time and resource requirements.

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
- If the CPU is preoccupied with one lengthy order process, all other orders are left sitting idle, and this can cause a significant backup. Therefore, FCFS lower the device utilization.
- In PQ, the Average waiting time is smaller than FCFS in preemptive type

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 the other hand, for PQ, the major problem is the starvation or indefinite blocking. The system keeps executing the high priority processes and the low priority processes never get executed.
- However, the problem of starvation in PQ can be solved through aging which means to gradually increase the priority of a process after a fixed interval of time by a fixed number.

Round Robin (RR):

- In RR, the throughput of queue heavily depends on the quantum time. Very large time quantum makes RR same as the FCFS while a very small time quantum will lead to the overhead as context switch will happen again and again after very small intervals.

Multilevel Queue (MLQ)

- In MLQ Some processes may face starvation as higher priority queues are never becoming empty.
- However, This problem can be solved by Aging technique like PQ scheduling.

Multilevel Feedback Queue (MLFQ)

- MLFQ is good but It is the most complex algorithm.

1.2 Implementation

♣ Get_mlq_proc() function:

First, this function check the ready queue from prio = 0 to Max_prio to know if the queue is empty or not. If it is, the function moves to next queue to check.

When it reaches non-empty queue, it will get and return the first element in that queue by function dequeue

```
1 struct pcb_t * get_mlq_proc(void) {
2     struct pcb_t * proc = NULL;
3     /*TODO: get a process from PRIORITY [ready_queue].
4      * Remember to use lock to protect the queue.
5      */
6     pthread_mutex_lock(&queue_lock);
7     for (int priority = 0 ; priority < MAX_Prio; ++priority)
8     {
9         if ( check_empty(&mlq_ready_queue[priority]) != 0) // queue not empty -> traverse
10        {
11            proc = dequeue(&mlq_ready_queue[priority]);
12            break;
13        }
14    }
15    pthread_mutex_unlock(&queue_lock);
16    return proc;
17 }
```

♣ Enqueue() function:

This function will put a new PCB into the given ready queue that has the same prio value with process. The new PCB will be placed after the last element in this queue.

```
1 void enqueue(struct queue_t * q, struct pcb_t * proc) {
2     /* TODO: put a new process to queue [q] */
3     if (q->size < MAX_QUEUE_SIZE)
4     {
5         q->proc[q->size] = proc;
6         q->size += 1;
7     }
8 }
9
10 }
```

♣ Dequeue() function:

This function return the suitable PCB - first element of the given ready queue. Move_up function is used to move almost all elemenst (from second elements until the last element) to left of the array by one.

```
1 struct pcb_t * dequeue(struct queue_t * q) {
2     /* TODO: return a pcb whose prioprity is the highest
3      * in the queue [q] and remember to remove it from q
4      */
5     if (q->size == 0) {
6         return NULL;
7     }
8     struct pcb_t * temp ;
9     temp = q->proc[0];
10    move_up(q);
11
12    q->size -= 1;
13    return temp;
14 }
15
16 void move_up(struct queue_t * q)
17 {
18     for (int i = 0 ; i <= q->size -2 ; ++i)
19     {
20         q->proc[i] = q->proc[i+1];
21     }
22 }
23 }
```

1.3 Result

1.3.1 Testcase sched_0:

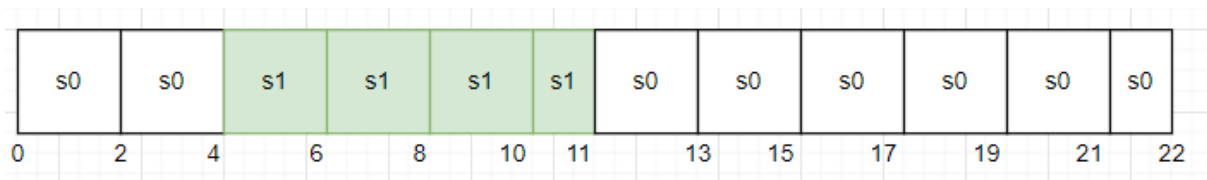
Configuration:

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

	Time start	Priority	Number of instruction
S0	0	2	15
S1	4	0	7

Figure 1: sched 0 config

Gantt Diagram



Real result

```
trungduong@trungduong-VirtualBox:~/ass/ossim_source_code/ossim_source_code$ make test_sched
----- SCHEDULING TEST 0 -----
./os sched_0
Time slot 0
  Loaded a process at input/proc/s0, PID: 1 PRIO: 2
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 PRIO: 0
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 2
Time slot 10
Time slot 11
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 2
Time slot 12
  CPU 0: Processed 2 has finished
  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
Time slot 16
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 17
Time slot 18
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
```

Figure 2: Result of sched0 (1)

```

CPU 0: Dispatched process 1
Time slot 15
Time slot 16
    CPU 0: Put process 1 to run queue
    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
NOTE: Read file output/sched_0 to verify your result

```

Figure 3: Result of sched0 (2)

1.3.2 Testcase sched_1:

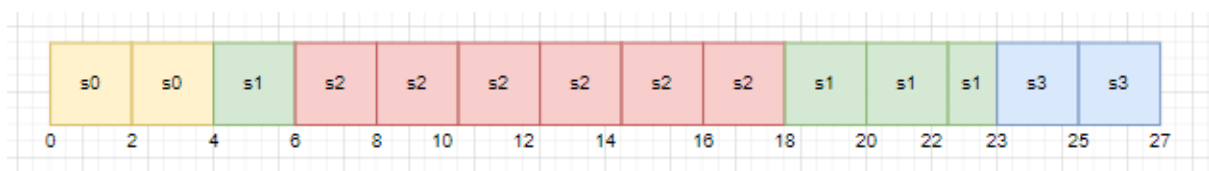
Configuration:

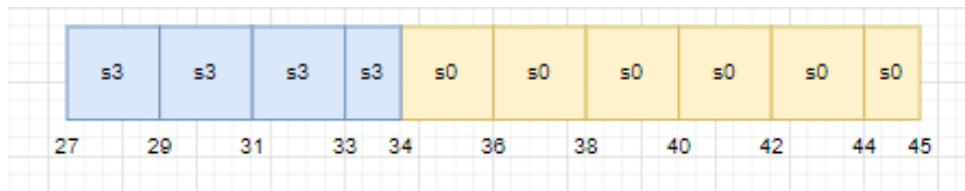
- Time slice = 2
- Number of CPU = 1
- Number of process to be run = 4

	Time start	Priority	Number of instruction
S0	0	3	15
S1	4	1	7
S2	6	0	12
S3	7	2	11

Figure 4: sched 1 config

Gantt Diagram





Real result

```

----- SCHEDULING TEST 1 -----
./os sched_1
Time slot 0
    Loaded a process at input/proc/s0, PID: 1 PRI0: 3
Time slot 1
    CPU 0: Dispatched process 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 PRI0: 1
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 2
Time slot 5
Time slot 6
    Loaded a process at input/proc/s2, PID: 3 PRI0: 0
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 3
Time slot 7
    Loaded a process at input/proc/s3, PID: 4 PRI0: 2
Time slot 8
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 3
Time slot 9
Time slot 10
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 3
Time slot 11
Time slot 12
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 3
Time slot 13
Time slot 14
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 3
Time slot 15
Time slot 16
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 3
Time slot 17
Time slot 18
    CPU 0: Processed 3 has finished

```

Figure 5: Result of sched1 (1)


```
Time slot 17
Time slot 18
    CPU 0: Processed 3 has finished
Time slot 19
    CPU 0: Dispatched process 2
Time slot 20
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 2
Time slot 21
Time slot 22
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 2
Time slot 23
    CPU 0: Processed 2 has finished
    CPU 0: Dispatched process 4
Time slot 24
Time slot 25
    CPU 0: Put process 4 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 4
Time slot 28
Time slot 29
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 4
Time slot 30
Time slot 31
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 4
Time slot 32
Time slot 33
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 4
Time slot 34
    CPU 0: Processed 4 has finished
    CPU 0: Dispatched process 1
Time slot 35
Time slot 36
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
```

Figure 6: Result of sched1 (2)

```
Time slot 34
    CPU 0: Processed 4 has finished
    CPU 0: Dispatched process 1
Time slot 35
Time slot 36
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 37
Time slot 38
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 39
Time slot 40
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 41
Time slot 42
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 43
Time slot 44
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 45
    CPU 0: Processed 1 has finished
    CPU 0 stopped
Time slot 46
NOTE: Read file output/sched_1 to verify your result
```

Figure 7: Result of sched1 (3)

2 Memory Management

2.1 Answer the question

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

♣ Paging

Paging is a memory management technique in which process address space is broken into blocks of the same size called pages (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called frames and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.

♣ Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.

Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging pages are of fixed size.

♣ Segmentation with paging

Pure segmentation is not very popular and not being used in many of the operating systems. However, segmentation can be combined with paging to get the best features out of both the techniques.

In segmentation with paging, the main memory is divided into variable size segments which are further divided into fixed size pages.

1. Pages are smaller than segments.
2. Each segment has a page table which means every program has multiple page tables.
3. The logical address is represented as:
 - Segment number: Points to the appropriate segment number.
 - Page number: Points to the exact page within the segment.
 - Page offset: Used as an offset within the page frame.

Each page table contains the various information about every page of the segment. The segment table contains the information about every segment. Each segment table entry points to a page table entry and every page table entry is mapped to one of the page within a segment.

♣ Advantages and disadvantages of segmentation with paging

• Advantages

- It reduces memory usage and simplifies memory allocation.
- Page table size is limited by the segment size.
- Segment table has only one entry corresponding to one actual segment.
- No external fragmentation, which means that there are no small unusable separated blocks when we free memory by some certain storage allocation algorithms. Therefore, segmented paging simplifies memory allocation.

• Disadvantages

- The complexity level will be much higher as compare to paging.

- Page tables need to be contiguously stored in the memory.
- Internal fragmentation still exists in pages, which means that there are unused small memory chunks left on allocated blocks if a program needs space less than allocated space on memory.

2.2 Implementation

♣ Get page table from section

This function takes in 5 bit segment level index *index* and first level table *seg_table*, needs to find the page table of the corresponding segment in the mentioned segment table.

```

1  static struct trans_table_t * get_trans_table(
2  addr_t index, // Segment level index
3  struct page_table_t * page_table) { // first level table
4
5  /*
6   * TODO: Given the Segment index [index], you must go through each
7   * row of the segment table [page_table] and check if the v_index
8   * field of the row is equal to the index
9   *
10  * */
11  int i;
12  for (i = 0; i < page_table->size; i++) {
13      // Enter your code here
14      if (page_table->table[i].v_index == index) {
15          return page_table->table[i].next_lv;
16      }
17  }
18  return NULL;
19 }

```

♣ Translate virtual address to physical address

To create the physical address, we only need to shift left 10-bit *p_index* of *page_table* and use OR operator to generate the corresponding address.

```

1  /* Translate virtual address to physical address. */
2  static int translate(
3  addr_t virtual_addr, // Given virtual address
4  addr_t * physical_addr, // Physical address to be returned
5  struct pcb_t * proc) { // Process uses given virtual address
6  /* Offset of the virtual address */
7  addr_t offset = get_offset(virtual_addr);
8  /* The first layer index */
9  addr_t first_lv = get_first_lv(virtual_addr);
10 /* The second layer index */
11 addr_t second_lv = get_second_lv(virtual_addr);
12 /* Search in the first level */
13 struct trans_table_t * trans_table = NULL;
14 trans_table = get_trans_table(first_lv, proc->seg_table);
15 if (trans_table == NULL) {
16     return 0;
17 }
18 int i;
19 for (i = 0; i < trans_table->size; i++) {
20     if (trans_table->table[i].v_index == second_lv) {
21         /* TODO: Concatenate the offset of the virtual address
22          * to [p_index] field of trans_table->table[i] to
23          * produce the correct physical address and save it to
24          * [*physical_addr] */
25         *physical_addr = trans_table->table[i].p_index << OFFSET_LEN | offset;
26         return 1;
27     }
28 }
29 return 0;
30 }

```

♣ Allocate Memory

- Check memory availability

- On the physical memory, we check the number of empty pages, if not used by any process, we count blank pages by 1. If there are enough pages to be allocated, the physical area is ready.
- On the virtual memory, we also check based on the break point of the process, do not exceed the allowed memory area.

```

1 addr_t alloc_mem(uint32_t size, struct pcb_t * proc) {
2     pthread_mutex_lock(&mem_lock);
3     addr_t ret_mem = 0;
4     /* TODO: Allocate [size] byte in the memory for the
5      * process [proc] and save the address of the first
6      * byte in the allocated memory region to [ret_mem].
7      * */
8
9     uint32_t num_pages = (size % PAGE_SIZE) ? size / PAGE_SIZE + 1 :
10         size / PAGE_SIZE; // Number of pages we will use
11     int mem_avail = 0; // We could allocate new memory region or not?
12
13     /* First we must check if the amount of free memory in
14      * virtual address space and physical address space is
15      * large enough to represent the amount of required
16      * memory. If so, set 1 to [mem_avail].
17      * Hint: check [proc] bit in each page of _mem_stat
18      * to know whether this page has been used by a process.
19      * For virtual memory space, check bp (break pointer).
20      * */
21     int free_pages = 0;
22     for (int i = 0; i < NUM_PAGES; ++i)
23     {
24         if (_mem_stat[i].proc == 0)
25         {
26             ++free_pages;
27         }
28     }
29     mem_avail = free_pages >= num_pages;

```

• Allocate

- Iterate through physical memory, find free pages, assign that process has been used on these pages.
- Update [proc], [index] and [next] field in mem_stat
- On physical memory, we find its segments and pages. From there we update the corresponding paging and segment tables.

```

1 if (mem_avail) {
2     /* We could allocate new memory region to the process */
3     ret_mem = proc->bp;
4     proc->bp += num_pages * PAGE_SIZE;
5     /* Update status of physical pages which will be allocated
6      * to [proc] in _mem_stat. Tasks to do:
7      * - Update [proc], [index], and [next] field
8      * - Add entries to segment table page tables of [proc]
9      * to ensure accesses to allocated memory slot is
10     * valid. */
11
12     uint32_t num_pages_use = 0;
13     for(int i = 0, j = 0, k = 0; i < NUM_PAGES ; i++) {
14         if(_mem_stat[i].proc == 0) { // page can use
15             _mem_stat[i].proc = proc->pid;
16             _mem_stat[i].index = j;
17             if (j != 0) {
18                 _mem_stat[k].next = i;
19             }
20             /*
21             addr_t physical_addr = i;
22             addr_t first_lv = get_first_lv (ret_mem + j * PAGE_SIZE);
23             addr_t second_lv = get_second_lv (ret_mem + j * PAGE_SIZE);
24             int booler = 0;

```

```

25
26     for(int n = 0; n < proc->seg_table->size; n++) {
27         if(proc->seg_table->table[n].v_index == first_lv) {
28             proc->seg_table->table[n].next_lv->table[proc->seg_table->table[n].next_lv->
                size].v_index = second_lv;
29             proc->seg_table->table[n].next_lv->table[proc->seg_table->table[n].next_lv->
                size].p_index = physical_addr;
30             proc->seg_table->table[n].next_lv->size++;
31             booler = 1;
32             break;
33         }
34     }
35
36     /* if not having first index in page table */
37     if(booler == 0) {
38         int n = proc->seg_table->size;
39         proc->seg_table->size++;
40         proc->seg_table->table[n].next_lv = (struct trans_table_t *)malloc(sizeof(
            struct trans_table_t));
41         proc->seg_table->table[n].next_lv->size++;
42         proc->seg_table->table[n].v_index = first_lv;
43         proc->seg_table->table[n].next_lv->table[0].v_index = second_lv;
44         proc->seg_table->table[n].next_lv->table[0].p_index = physical_addr;
45     }
46     /*
47     k = i;
48     j++;
49     num_pages_use++;
50     if(num_pages_use == num_pages) {
51         _mem_stat[k].next = -1; // last page in list
52         break;
53     }
54 }
55 }
56 }
57 pthread_mutex_unlock(&mem_lock);
58 return ret_mem;
59 }

```

♣ Free Memory

- Check whether or not the given virtual address is valid. If yes, then convert logical address to physical address stored in *physical_addr*.
- Then use while loop to retrieve all frames needed to be free in physical memory(*mem_stat*) and set *mem_stat[next].proc* = 0 indicates that it is free.
- After that, we remove corresponding elements of the *page_table*. More specific, we shift left all elements beginning from the rejected element to the end and assign all values reasonably (*v_index*, *p_index*, *size*) in that *page_table*. If *page_table* is empty after implementing function above, we must free this *page_table* and use the same algorithm to remove correct the element in *seg_table* (*v_index* and *pages* variables).

```

1 int free_mem(addr_t address, struct pcb_t * proc) {
2     /*TODO: Release memory region allocated by [proc]. The first byte of
3     * this region is indicated by [address]. Task to do:
4     * - Set flag [proc] of physical page use by the memory block
5     *   back to zero to indicate that it is free.
6     * - Remove unused entries in segment table and page tables of
7     *   the process [proc].
8     * - Remember to use lock to protect the memory from other
9     *   processes. */
10
11     pthread_mutex_lock(&mem_lock);
12     addr_t v_addr = address;
13     addr_t phy_addr = 0;
14     if (translate(v_addr, &phy_addr, proc) == 0)
15     {

```

```

16 pthread_mutex_unlock(&mem_lock);
17 return 1;
18 }
19 /* next used to free all the next elements in physical address (mem_stat) */
20 int num_pages = 0;
21 int i = 0;
22 for (i = phy_addr >> OFFSET_LEN; i != -1; i = _mem_stat[i].next)
23 {
24     ++num_pages;
25     _mem_stat[i].proc = 0;
26 }
27 // clear virtual
28 addr_t v_seg, v_page;
29 for (int i = 0; i < num_pages; ++i)
30 {
31     /* remove seg table and page table */
32     v_seg = get_first_lv(v_addr + i * PAGE_SIZE);
33     v_page = get_second_lv(v_addr + i * PAGE_SIZE);
34     struct trans_table_t *trans_table = get_trans_table(v_seg, proc->seg_table);
35     if (!trans_table)
36     {
37         continue;
38     }
39     for (int j = 0; j < trans_table->size; ++j)
40     {
41         if (trans_table->table[j].v_index == v_page)
42         {
43             --trans_table->size;
44             trans_table->table[j] = trans_table->table[trans_table->size];
45             break;
46         }
47     }
48     if (trans_table->size == 0 && proc->seg_table)
49     {
50         for (int k = 0; k < proc->seg_table->size; ++k)
51         {
52             if (proc->seg_table->table[k].v_index == v_seg)
53             {
54                 --proc->seg_table->size;
55                 proc->seg_table->table[k] = proc->seg_table->table[proc->seg_table->size];
56                 proc->seg_table->table[proc->seg_table->size].v_index = 0;
57                 free(proc->seg_table->table[proc->seg_table->size].next_lv);
58             }
59         }
60     }
61 }
62 proc->bp -= num_pages * PAGE_SIZE;
63 pthread_mutex_unlock(&mem_lock);
64
65 return 0;
66 }

```

2.3 Result

```
----- 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 8: make test_mem

2.4 Explanation

By default, the size of virtual RAM is 1 MB so we must use 20 bit to represent the address of each of its byte. With the segmentation with paging mechanism, we use the first 5 bits for segment index, the next 5 bits for page index and the last 10 bits for offset. Therefore, the page's size is $2^{10} = 1024$ bytes, and number of pages in RAM is $\frac{2^{20}}{2^{10}} = 1024$ pages.

♣ Input m0

```
1 1 7
2 alloc 13535 0
3 alloc 1568 1
4 free 0
5 alloc 1386 2
6 alloc 4564 4
7 write 102 1 20
8 write 21 2 1000
```

1. alloc 13535 0

Process m0 allocates 13535 bytes (14 pages from 0 to 13) and stores the virtual address of the first page for register 0.

2. alloc 1568 1

Process m0 allocates 1568 bytes (2 pages from 14 to 15) and stores the virtual address of the first page for register 1.

3. free 0

Process m0 releases all pages held by register 0 (14 pages from 0 to 13) in RAM.

4. alloc 1386 2

Process m0 allocates 1386 bytes (2 pages from 0 to 1) and stores the virtual address of the first page for register 2.

5. alloc 4564 4

Process m0 allocates 4564 bytes (5 pages from 2 to 6) and stores the virtual address of the first page for register 4.

6. write 102 1 20

Process `m0` writes 102 (66 in hex) to memory.

We have: physical address = base address + offset,

- base address: physical address of the first byte held by the first register, address 0x03800.
- offset: 20 (14 in hex).

\Rightarrow physical address = 0x03814

7. write 21 2 1000

Process `m0` writes 21 (15 in hex) to memory.

\Rightarrow physical address = 0x00000 + 0x3e8 = 0x3e8

♣ Input m1

```
1 1 8
2 alloc 13535 0
3 alloc 1568 1
4 free 0
5 alloc 1386 2
6 alloc 4564 4
7 free 2
8 free 4
9 free 1
```

Because each command free is corresponding to each command alloc, the result will be empty.

3 Put It All Together

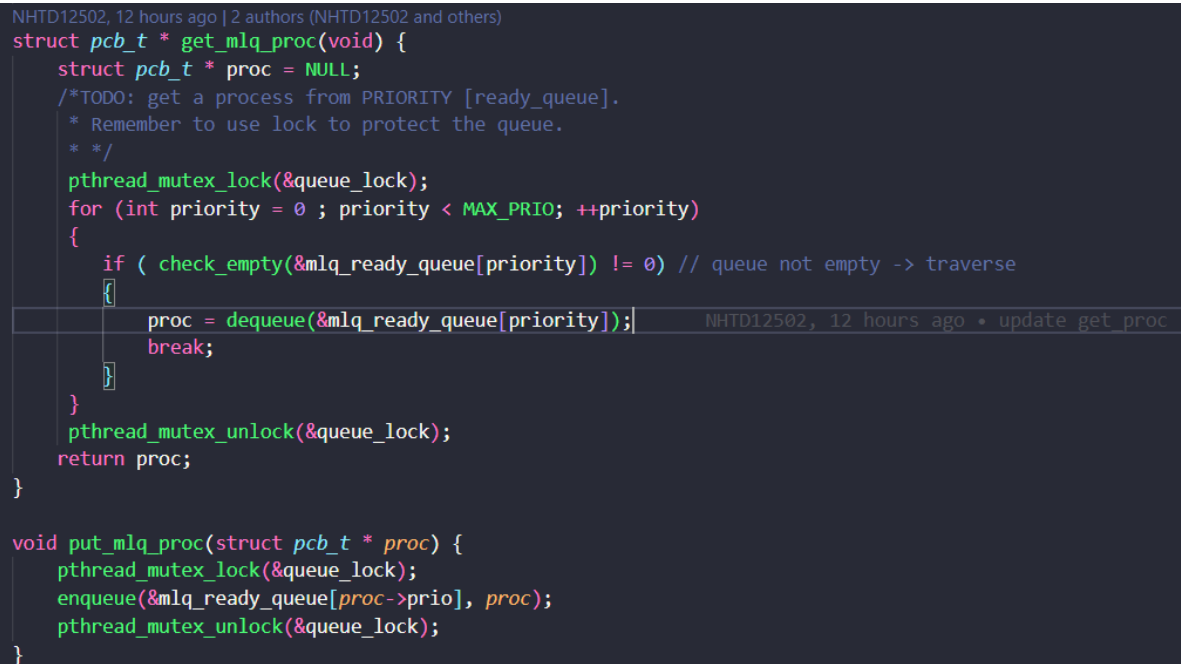
Synchronization

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 (synchronization). The solution is using 2 mutex locks to control the access of multiple CPUs to shared memory regions and prevent from unwanted behaviors.

In our code, we use 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.

For example in Sched.c:



```
NHTD12502, 12 hours ago | 2 authors (NHTD12502 and others)
struct pcb_t * get_mlq_proc(void) {
    struct pcb_t * proc = NULL;
    /*TODO: get a process from PRIORITY [ready_queue].
    * Remember to use lock to protect the queue.
    * */
    pthread_mutex_lock(&queue_lock);
    for (int priority = 0 ; priority < MAX_PRIO; ++priority)
    {
        if ( check_empty(&mlq_ready_queue[priority]) != 0) // queue not empty -> traverse
        {
            proc = dequeue(&mlq_ready_queue[priority]);
            break;
        }
    }
    pthread_mutex_unlock(&queue_lock);
    return proc;
}

void put_mlq_proc(struct pcb_t * proc) {
    pthread_mutex_lock(&queue_lock);
    enqueue(&mlq_ready_queue[proc->prio], proc);
    pthread_mutex_unlock(&queue_lock);
}
```

Figure 9: queue_lock

- **Mem_lock** - protect the physical as well as virtual memory: Whenever the processes call alloc(), free(), read(), write()

For example in mem.c:

```

int free_mem(addr_t address, struct pcb_t * proc) {
    /*TODO: Release memory region allocated by [proc]. The first byte of
    * this region is indicated by [address]. Task to do:
    * - Set flag [proc] of physical page use by the memory block
    * back to zero to indicate that it is free.
    * - Remove unused entries in segment table and page tables of
    * the process [proc].
    * - Remember to use lock to protect the memory from other
    * processes. */

    pthread_mutex_lock(&mem_lock);
    addr_t v_addr = address;
    addr_t phy_addr = 0;
    if (translate(v_addr, &phy_addr, proc) == 0)
    {
        pthread_mutex_unlock(&mem_lock);
        return 1;
    }
}

```

Figure 10: mem_lock

```

int read_mem(addr_t address, struct pcb_t * proc, BYTE * data) {
    addr_t physical_addr;
    if (translate(address, &physical_addr, proc)) {
        // add lock
        pthread_mutex_lock(&mem_lock);
        *data = _ram[physical_addr];
        pthread_mutex_unlock(&mem_lock);
        return 0;
    }else{
        return 1;
    }
}

int write_mem(addr_t address, struct pcb_t * proc, BYTE data) {
    addr_t physical_addr;
    if (translate(address, &physical_addr, proc)) {
        // add lock
        pthread_mutex_lock(&mem_lock);
        _ram[physical_addr] = data;
        pthread_mutex_unlock(&mem_lock);
        return 0;
    }else{
        return 1;
    }
}

```

Figure 11: mem_lock

Question: What will happen if the synchronization is not handled in your simple OS? Illustrate by example the problem of your simple OS if you have any.

Since our OS run on multiple processors, it is possible that share resources could be concurrently accessed by more than one process at a time. It could lead to data inconsistency and we may get unexpected result. For example: if enqueue() and dequeue() run at the same time, we expected to enqueue() a process first then dequeue() it, there is a chance that dequeue() run first → we may get wrong process from dequeue().

Conclusion: If we remove the synchronization, the result will be wrong due to the critical section. Therefore, using the synchronization will ensure OS to return the correct answer.

3.1 Result:

3.1.1 Os_mlq_0:

- Time slice = 6
- Number of CPU = 2
- Number of process = 4

Table 1: os_mlq_0

	Timestart	Prio	Number of instruction
p0	0	0	11
p1	2	15	10

```
----- OS TEST 0 -----
./os os_mlq_0
    Loaded a process at input/proc/p0, PID: 1 PRIO: 0
Time slot 0
    CPU 1: Dispatched process 1
Time slot 1
Time slot 2
    Loaded a process at input/proc/p1, PID: 2 PRIO: 15
Time slot 3
    CPU 0: Dispatched process 2
    Loaded a process at input/proc/p1, PID: 3 PRIO: 0
Time slot 4
    Loaded a process at input/proc/p1, PID: 4 PRIO: 0
Time slot 5
Time slot 6
    CPU 1: Put process 1 to run queue
    CPU 1: Dispatched process 3
Time slot 7
Time slot 8
Time slot 9
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 4
Time slot 10
Time slot 11
Time slot 12
    CPU 1: Put process 3 to run queue
    CPU 1: Dispatched process 1
Time slot 13
Time slot 14
Time slot 15
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 3
Time slot 16
    CPU 1: Processed 1 has finished
    CPU 1: Dispatched process 4
Time slot 17
Time slot 18
Time slot 19
    CPU 0: Processed 3 has finished
    CPU 0: Dispatched process 2
Time slot 20
    CPU 1: Processed 4 has finished
    CPU 1 stopped
```

Figure 12: Os_mlq_0 (1)

```

Time slot 19
    CPU 0: Processed 3 has finished
    CPU 0: Dispatched process 2
Time slot 20
    CPU 1: Processed 4 has finished
    CPU 1 stopped
Time slot 21
Time slot 22
Time slot 23
    CPU 0: Processed 2 has finished
    CPU 0 stopped
NOTE: Read file output/os_0 to verify your result

```

Figure 13: Os_mlq_0 (2)

3.1.2 Os_mlq_1

- Time slice = 2
- Number of CPU = 4
- Number of process = 8

Table 2: os_mlq_1

	Timestart	Prio	Number of instruction
p0	1	139	11
s3	2	39	17
m1	4	15	8
s2	6	120	13
m0	7	120	7
p1	9	15	10
s0	11	38	15
s1	16	0	7

```

----- OS TEST 1 -----
./os os_mfq_1
Time slot 0
Time slot 1
    Loaded a process at input/proc/p0, PID: 1 PRI0: 139
    CPU 0: Dispatched process 1
Time slot 2
    Loaded a process at input/proc/s3, PID: 2 PRI0: 39
Time slot 3
    CPU 3: Dispatched process 2
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 4
    Loaded a process at input/proc/m1, PID: 3 PRI0: 15
    CPU 2: Dispatched process 3
Time slot 5
    CPU 3: Put process 2 to run queue
    CPU 3: Dispatched process 2
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 6
    Loaded a process at input/proc/s2, PID: 4 PRI0: 120
    CPU 1: Dispatched process 4
    CPU 2: Put process 3 to run queue
    CPU 2: Dispatched process 3
Time slot 7
    CPU 3: Put process 2 to run queue
    CPU 3: Dispatched process 2
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
    Loaded a process at input/proc/m0, PID: 5 PRI0: 120
Time slot 8
    CPU 1: Put process 4 to run queue
    CPU 1: Dispatched process 5
    CPU 2: Put process 3 to run queue
    CPU 2: Dispatched process 3
Time slot 9
    CPU 3: Put process 2 to run queue
    CPU 3: Dispatched process 2
    Loaded a process at input/proc/p1, PID: 6 PRI0: 15
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 6

```

Figure 14: Os_mfq_1 (1)

```

CPU 2: Dispatched process 3
Time slot 9
CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 2
Loaded a process at input/proc/p1, PID: 6 PRIO: 15
CPU 0: Put process 1 to run queue
CPU 0: Dispatched process 6
Time slot 10
CPU 2: Put process 3 to run queue
CPU 2: Dispatched process 3
CPU 1: Put process 5 to run queue
CPU 1: Dispatched process 4
Time slot 11
CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 2
Loaded a process at input/proc/s0, PID: 7 PRIO: 38
CPU 0: Put process 6 to run queue
CPU 0: Dispatched process 6
Time slot 12
CPU 1: Put process 4 to run queue
CPU 1: Dispatched process 7
CPU 2: Processed 3 has finished
CPU 2: Dispatched process 5
Time slot 13
CPU 0: Put process 6 to run queue
CPU 0: Dispatched process 6
CPU 3: Put process 2 to run queue
CPU 3: Dispatched process 2
Time slot 14
CPU 3: Processed 2 has finished
CPU 3: Dispatched process 4
CPU 2: Put process 5 to run queue
CPU 2: Dispatched process 5
CPU 1: Put process 7 to run queue
CPU 1: Dispatched process 7
Time slot 15
CPU 0: Put process 6 to run queue
CPU 0: Dispatched process 6
Time slot 16
CPU 3: Put process 4 to run queue
CPU 3: Dispatched process 4
Loaded a process at input/proc/s1, PID: 8 PRIO: 0
CPU 2: Put process 5 to run queue
CPU 2: Dispatched process 8
CPU 1: Put process 7 to run queue
CPU 1: Dispatched process 7

```

Figure 15: Os_mlq_1 (2)

```

CPU 1: Dispatched process 7
Time slot 17
    CPU 0: Put process 6 to run queue
    CPU 0: Dispatched process 6
Time slot 18
    CPU 3: Put process 4 to run queue
    CPU 3: Dispatched process 5
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
    CPU 2: Put process 8 to run queue
    CPU 2: Dispatched process 8
Time slot 19
    CPU 3: Processed 5 has finished
    CPU 3: Dispatched process 4
    CPU 0: Processed 6 has finished
    CPU 0: Dispatched process 1
Time slot 20
    CPU 2: Put process 8 to run queue
    CPU 2: Dispatched process 8
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
Time slot 21
    CPU 3: Put process 4 to run queue
    CPU 3: Dispatched process 4
    CPU 0: Processed 1 has finished
    CPU 0 stopped
Time slot 22
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
    CPU 2: Put process 8 to run queue
    CPU 2: Dispatched process 8
Time slot 23
    CPU 2: Processed 8 has finished
    CPU 2 stopped
    CPU 3: Processed 4 has finished
    CPU 3 stopped
Time slot 24
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
Time slot 25
Time slot 26
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7

```

Figure 16: Os_mfq_1 (3)


```

Time slot 24
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
Time slot 25
Time slot 26
    CPU 1: Put process 7 to run queue
    CPU 1: Dispatched process 7
Time slot 27
    CPU 1: Processed 7 has finished
    CPU 1 stopped
NOTE: Read file output/os_1 to verify your result

```

Figure 17: Os_mlq_1 (4)

3.1.3 Os_mlq_2:

- Time slice = 2
- Number of CPU = 2
- Number of process = 8

Table 3: os_mlq_1

	Timestart	Prio	Number of instruction
s4	1	4	29
s3	2	3	17
m1	4	2	8
s2	6	3	13
m0	7	3	7
p1	9	2	10
s0	11	1	15
s1	16	0	7

```

----- OS TEST 2 -----
./os os_mlq_2
Time slot 0
Time slot 1
    Loaded a process at input/proc/s4, PID: 1 PRIO: 4
    CPU 0: Dispatched process 1
Time slot 2
    Loaded a process at input/proc/s3, PID: 2 PRIO: 3
    CPU 1: Dispatched process 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/m1, PID: 3 PRIO: 2
    CPU 1: Put process 2 to run queue
    CPU 1: Dispatched process 3
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: 4 PRIO: 3
    CPU 1: Put process 3 to run queue
    CPU 1: Dispatched process 3
Time slot 7
    Loaded a process at input/proc/m0, PID: 5 PRIO: 3
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 4
Time slot 8
    CPU 1: Put process 3 to run queue
    CPU 1: Dispatched process 3
Time slot 9
    Loaded a process at input/proc/p1, PID: 6 PRIO: 2
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 6
Time slot 10
    CPU 1: Put process 3 to run queue
    CPU 1: Dispatched process 3
Time slot 11
    Loaded a process at input/proc/s0, PID: 7 PRIO: 1
    CPU 0: Put process 6 to run queue
    CPU 0: Dispatched process 7
Time slot 12
    CPU 1: Processed 3 has finished
    CPU 1: Dispatched process 6
Time slot 13
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7

```

Figure 18: Os_mlq_2 (1)

```

Time slot 13
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 14
    CPU 1: Put process 6 to run queue
    CPU 1: Dispatched process 6
Time slot 15
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 16
    Loaded a process at input/proc/s1, PID: 8 PRIO: 0
    CPU 1: Put process 6 to run queue
    CPU 1: Dispatched process 8
Time slot 17
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 18
    CPU 1: Put process 8 to run queue
    CPU 1: Dispatched process 8
Time slot 19
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 20
    CPU 1: Put process 8 to run queue
    CPU 1: Dispatched process 8
Time slot 21
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
Time slot 22
    CPU 1: Put process 8 to run queue
    CPU 1: Dispatched process 8
Time slot 23
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
    CPU 1: Processed 8 has finished
    CPU 1: Dispatched process 6
Time slot 24
Time slot 25
    CPU 0: Put process 7 to run queue
    CPU 0: Dispatched process 7
    CPU 1: Put process 6 to run queue
    CPU 1: Dispatched process 6
Time slot 26
    CPU 0: Processed 7 has finished
    CPU 0: Dispatched process 5
Time slot 27
    CPU 1: Processed 6 has finished
    CPU 1: Dispatched process 2
Time slot 28
    CPU 0: Put process 5 to run queue

```

Figure 19: Os_mlq_2 (2)

```

Time slot 27
  CPU 1: Processed 6 has finished
  CPU 1: Dispatched process 2
Time slot 28
  CPU 0: Put process 5 to run queue
  CPU 0: Dispatched process 4
Time slot 29
  CPU 1: Put process 2 to run queue
  CPU 1: Dispatched process 5
Time slot 30
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 2
Time slot 31
  CPU 1: Put process 5 to run queue
  CPU 1: Dispatched process 4
Time slot 32
  CPU 0: Put process 2 to run queue
  CPU 0: Dispatched process 5
Time slot 33
  CPU 1: Put process 4 to run queue
  CPU 1: Dispatched process 2
Time slot 34
  CPU 0: Put process 5 to run queue
  CPU 0: Dispatched process 4
Time slot 35
  CPU 1: Put process 2 to run queue
  CPU 1: Dispatched process 5
Time slot 36
  CPU 0: Put process 4 to run queue
  CPU 0: Dispatched process 2
  CPU 1: Processed 5 has finished
  CPU 1: Dispatched process 4
Time slot 37
  CPU 0: Processed 2 has finished
  CPU 0: Dispatched process 1
Time slot 38
  CPU 1: Put process 4 to run queue
  CPU 1: Dispatched process 4
Time slot 39
  CPU 0: Put process 1 to run queue
  CPU 0: Dispatched process 1
Time slot 40
  CPU 0: Processed 1 has finished
  CPU 0 stopped
  CPU 1: Processed 4 has finished
  CPU 1 stopped
NOTE: Read file output/os_2 to verify your result

```

Figure 20: Os_mlq_2 (3)

4 Conclusion

In this assignment, we have shown:

- The simulation of scheduler in OS using Multi-level Queue (MLQ).
- The simulation of memory management, the relation between physical memory and virtual memory.
- Make data consistent using synchronization.