

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



OPERATING SYSTEMS (CO2017)

Assignment (Semester 221)

" Simple Operating System "

Lecturer: Lê Thanh Vân

Students: Nguyễn Tôn Minh - 2052600 - CC04
Đỗ Nguyễn Huy Hiệu - 2053003 - CC01
Bùi Ngọc Nam Anh - 2052833 - CC04

HO CHI MINH CITY, NOVEMBER 2022



Table of Contents

1	TEAM MEMBER	2
2	THEORY OF THE ASSIGNMENT	3
2.1	Schedulers, Dispatchers and their uses	3
2.2	Synchronization	4
2.3	Multilevel Feedback Queue	4
2.4	Memory Management	4
3	SCHEDULER	6
3.1	Implementation	6
3.1.1	queue.c	6
3.1.2	sched.c	7
3.2	Result	8
3.3	Question	11
4	Memory Management	12
4.1	Implementation	12
4.1.1	Mapping from virtual address to physical address	12
4.1.2	Allocate and free memory	14
4.1.2.a	Memory allocation	14
4.1.2.b	Free memory	16
4.2	Result	18
4.3	Question	21
5	PUT IT ALL TOGETHER	23
5.1	Question	32
6	CONCLUSION	33



1 TEAM MEMBER

Student Name	Student ID
Nguyễn Tôn Minh	2052600
Đỗ Nguyễn Huy Hiệu	2053003
Bùi Ngọc Nam Anh	2052833

2 THEORY OF THE ASSIGNMENT

2.1 Schedulers, Dispatchers and their uses

Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types:

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Bảng 1: Comparison among Schedulers

Dispatcher

It is a special type of program that comes into play only after the scheduler. Once a scheduler completes selecting its processes, the dispatcher then takes that intended process to its desired queue/ state. A dispatcher is basically a module that provides a process with total control over the CPU (after the short-term scheduler finally selects it). This function requires the following:

- Context switching
- Switching to the user mode
- Jumping into the user program's proper location for restarting that given program

2.2 Synchronization

Definition:

An operating system is a software that manages all applications on a device and basically helps in the smooth functioning of our computer. Because of this reason, the operating system has to perform many tasks, and sometimes simultaneously. This isn't usually a problem unless these simultaneously occurring processes use a common resource.

Uses in OS:

Let us take a look at why exactly we need Process Synchronization. For example, If a processA is trying to read the data present in a memory location while another processB is trying to change the data present at the same location, there is a high chance that the data read by the processA will be incorrect.

Different elements/sections of a program:

- Entry Section: The entry Section decides the entry of a process.
- Critical Selection: Critical section allows and makes sure that only one process is modifying the shared data.
- Exit Section: The entry of other processes in the shared data after the execution of one process is handled by the Exit section.
- Remainder Section: The remaining part of the code which is not categorized as above is contained in the Remainder section.

2.3 Multilevel Feedback Queue

In computer science, a multilevel feedback queue is a scheduling algorithm. Scheduling algorithms are designed to have some process running at all times to keep the central processing unit (CPU) busy. The multilevel feedback queue extends standard algorithms with the following design requirements:

- Separate processes into multiple ready queues based on their need for the processor.
- Give preference to processes with short CPU bursts.
- Give preference to processes with high I/O bursts. (I/O bound processes will sleep in the wait queue to give other processes CPU time.)

2.4 Memory Management

In a multiprogramming computer, the operating system resides in a part of memory and the rest is used by multiple processes. The task of subdividing the memory among different processes is called memory management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.



Memory Management is required by:

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

3 SCHEDULER

3.1 Implementation

3.1.1 queue.c

```
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) return;
4     // find correct position for proc
5     int pos;
6     for (pos=0; pos < q->size; pos++){
7         if (proc->prio <= q->proc[pos]->prio) break;
8     }
9     // Shift other proc , leave space for new proc
10    for (int i = q->size; i > pos; i--) {
11        q->proc[i] = q->proc[i-1];
12    }
13
14    q->proc[pos] = proc;
15    q->size++;
16 }
17 struct pcb_t * dequeue(struct queue_t * q) {
18     /* TODO: return a pcb whose priority is the highest
19      * in the queue [q] and remember to remove it from q
20      * */
21     if (! q->size) return NULL;
22     q->size --;
23     return q->proc[q->size];
24 }
```

3.1.2 sched.c

```
1 struct pcb_t *get_mlq_proc(void)
2 {
3     struct pcb_t * proc = NULL;
4     /*TODO: get a process from PRIORITY [ready_queue].
5      * Remember to use lock to protect the queue
6      * */
7     pthread_mutex_lock(&queue_lock);
8     int i;
9     for (i = MAX_PRIO - 1; i >= 0; i--) {
10         if (mlq_ready_queue[i].size != 0)
11         {
12             proc = dequeue(&mlq_ready_queue[i]);
13             break;
14         }
15     }
16     // If no proc in mlq_ready_queue
17     if(i == -1)
18     {
19         // Put back proc from run_queue to mlq_ready_queue if have any
20         for(int j=0; j < run_queue.size; j++){
21             enqueue(&mlq_ready_queue[run_queue.proc[j]->prio], run_queue.proc[j]);
22         }
23         run_queue.size = 0;
24
25         // dequeue proc after put back if have any
26         for(int k = MAX_PRIO - 1; k >= 0; k--){
27             {
28                 if(mlq_ready_queue[k].size != 0)
29                 {
30                     proc = dequeue(&mlq_ready_queue[k]);
31                     break;
32                 }
33             }
34         }
35         pthread_mutex_unlock(&queue_lock);
36         return proc;
37     }
```


3.2 Result

Here is our output of sched.c

```
1 ----- SCHEDULING TEST 0 -----
2 ./os sched_0
3 Time slot 0
4   Loaded a process at input/proc/s0, PID: 1 PRI0: 0
5 Time slot 1
6   CPU 0: Dispatched process 1
7 Time slot 2
8 Time slot 3
9   CPU 0: Put process 1 to run queue
10  CPU 0: Dispatched process 1
11 Time slot 4
12   Loaded a process at input/proc/s1, PID: 2 PRI0: 3
13 Time slot 5
14   CPU 0: Put process 1 to run queue
15   CPU 0: Dispatched process 2
16 Time slot 6
17 Time slot 7
18   CPU 0: Put process 2 to run queue
19   CPU 0: Dispatched process 2
20 Time slot 8
21 Time slot 9
22   CPU 0: Put process 2 to run queue
23   CPU 0: Dispatched process 2
24 Time slot 10
25 Time slot 11
26   CPU 0: Put process 2 to run queue
27   CPU 0: Dispatched process 2
28 Time slot 12
29   CPU 0: Processed 2 has finished
30   CPU 0: Dispatched process 1
31 Time slot 13
32 Time slot 14
33   CPU 0: Put process 1 to run queue
34   CPU 0: Dispatched process 1
35 Time slot 15
36 Time slot 16
37   CPU 0: Put process 1 to run queue
38   CPU 0: Dispatched process 1
39 Time slot 17
40 Time slot 18
41   CPU 0: Put process 1 to run queue
42   CPU 0: Dispatched process 1
43 Time slot 19
44 Time slot 20
45   CPU 0: Put process 1 to run queue
46   CPU 0: Dispatched process 1
47 Time slot 21
48 Time slot 22
49   CPU 0: Put process 1 to run queue
50   CPU 0: Dispatched process 1
51 Time slot 23
52   CPU 0: Processed 1 has finished
53   CPU 0 stopped
54 NOTE: Read file output/sched_0 to verify your result

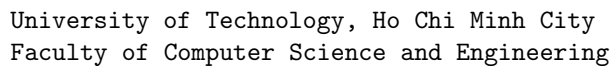
1 ----- SCHEDULING TEST 1 -----
2 ./os sched_1
3 Time slot 0
```



PROCESS		p1				p2						p1											
TIME SLOT	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Hình 1:

```
4   Loaded a process at input/proc/s0, PID: 1 PRI0: 1
5   Time slot 1
6   CPU 0: Dispatched process 1
7   Time slot 2
8   Time slot 3
9   CPU 0: Put process 1 to run queue
10  CPU 0: Dispatched process 1
11  Time slot 4
12  Loaded a process at input/proc/s1, PID: 2 PRI0: 2
13  Time slot 5
14  CPU 0: Put process 1 to run queue
15  CPU 0: Dispatched process 2
16  Loaded a process at input/proc/s2, PID: 3 PRI0: 2
17  Time slot 6
18  Time slot 7
19  CPU 0: Put process 2 to run queue
20  CPU 0: Dispatched process 3
21  Loaded a process at input/proc/s3, PID: 4 PRI0: 4
22  Time slot 8
23  Time slot 9
24  CPU 0: Put process 3 to run queue
25  CPU 0: Dispatched process 4
26  Time slot 10
27  Time slot 11
28  CPU 0: Put process 4 to run queue
29  CPU 0: Dispatched process 4
30  Time slot 12
31  Time slot 13
32  CPU 0: Put process 4 to run queue
33  CPU 0: Dispatched process 4
34  Time slot 14
35  Time slot 15
36  CPU 0: Put process 4 to run queue
37  CPU 0: Dispatched process 4
38  Time slot 16
39  Time slot 17
40  CPU 0: Put process 4 to run queue
41  CPU 0: Dispatched process 4
42  Time slot 18
43  Time slot 19
44  CPU 0: Put process 4 to run queue
45  CPU 0: Dispatched process 4
46  Time slot 20
47  CPU 0: Processed 4 has finished
48  CPU 0: Dispatched process 2
49  Time slot 21
50  Time slot 22
51  CPU 0: Put process 2 to run queue
52  CPU 0: Dispatched process 3
53  Time slot 23
54  Time slot 24
55  CPU 0: Put process 3 to run queue
56  CPU 0: Dispatched process 2
57  Time slot 25
58  Time slot 26
```



PROCESS	p1					p2		p3		p4										p2		p3	
TIME SLOT	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
PROCESS	p2		p3		p2		p3							p1									
TIME SLOT	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45

Assignment for Operating Systems - Semester 221 - Academic year 2022 - 2023

3.3 Question

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

In multilevel feedback scheduling, the processes are allowed to move in between the queues, the idea behind is to separate the processes with different CPU – burst characteristics. If any process uses too much CPU then it'll be moved to the lower priority queue and if a process waiting too much for the CPU is moved to the high priority queue this prevents starvation. Multilevel feedback queue is the most general scheme and also the most complex.

Advantages of MLFQ compare to another scheduling algorithms:

- MLFQ allows different processes to move between different queues while MLQ processes are assigned permanently into queues.
- It prevents starvation.
- In MLQ priority is fixed while in MLFQ, priority can be changed during runtime as process are allowed to move between queues. There for, lower the scheduling overhead and more flexible.

4 Memory Management

4.1 Implementation

4.1.1 Mapping from virtual address to physical address

```
1 #define ADDRESS_SIZE 20
2 #define OFFSET_LEN 10
3 #define FIRST_LV_LEN
4 #define SECOND_LV_LEN 5
5 #define SEGMENT_LEN FIRST_LV_LEN
6 #define PAGE_LEN SECOND_LV_LEN
7
8 #define NUM_PAGES (1 << (ADDRESS_SIZE - OFFSET_LEN))
9 #define PAGE_SIZE (1 << OFFSET_LEN)
```

As can be seen that, we define that 20 bits are used to represent the address, the first 5 bits encode the segment length and mean the next 5 bits represent page length and the last 10 bits denote the offset.

Furthermore, we will complete 2 function `get_trans_table()` and `translate()` to make the transition from the virtual address of a process to the address.

get_trans_table(): Finding a translate table with a segment index for a process.

```
1 static struct trans_table_t *get_trans_table(
2     addr_t index, // Segment level index
3     struct page_table_t *page_table)
4 { // first level table
5
6     /*
7      * TODO: Given the Segment index [index], you must go through each
8      * row of the segment table [page_table] and check if the v_index
9      * field of the row is equal to the index
10     */
11     /* */
12     int i;
13     for (i = 0; i < page_table->size; i++)
14     {
15         // Enter your code here
16         if (page_table->table[i].v_index == index)
17         {
18             return page_table->table[i].next_lv;
19         }
20     }
21     return NULL;
22 }
```

translate(): use `get_trans_table` to convert from virtual address to physical address.

```
1 static int translate(
2     addr_t virtual_addr, // Given virtual address
3     addr_t *physical_addr, // Physical addresses to be returned
4     struct pcb_t *proc)
5 { // Process uses given virtual address
6
7     /* Offset of the virtual address */
8     addr_t offset = get_offset(virtual_addr);
9     /* The first layer index */
10    addr_t first_lv = get_first_lv(virtual_addr);
```

```
11  /* The second layer index */
12  addr_t second_lv = get_second_lv(virtual_addr);
13  /* Search in the first level */
14  struct trans_table_t * trans_table = NULL;
15  trans_table = get_trans_table(first_lv, proc->page_table);
16  if (trans_table == NULL)
17  {
18      return 0;
19  }
20  int i;
21  for (i = 0; i < trans_table->size; i++)
22  {
23      if (trans_table->table[i].v_index == second_lv)
24      {
25          /* TODO: Concatenate the offset of the virtual address
26           * to [p_index] field of trans_table->table[i] to
27           * produce the correct physical address and save it to
28           * [*physical_addr] */
29          *physical_addr = (trans_table->table[i].p_index << OFFSET_LEN) | offset
30          ;
31          return 1;
32      }
33  }
34  return 0;
```

4.1.2 Allocate and free memory

The idea is that we must check if there is enough free memory in both physical and virtual memory space, by comparing the number of free pages with the number of needed pages, and ensure that the size of virtual memory does not expand beyond the maximum size of the RAM. First, we will loop through every page in RAM to find available pages to allocate. To do that we need some new variables *v_i*, *p_i*, *prev_v_i* and *num_allocated_pages* to keep track of the current physical index, the current virtual index, the physical index of the last allocated page, and the total number of allocated pages. If finding a free pages, update the *proc*, index and the next of the previous allocated page. Then we update the corresponding page table by adding a new row. If the table is not found, we simply creating a new one. And after a successful allocation, we increment *v_i* to allocate the next page until *num_allocated_pages* is equal to the desired number.

For the *free_memory()* we found the physical page index corresponding to the given virtual address by using *translate()* function and shift right operator, and store it in index variable and this is the index of the first page that we need to free. To delete, we set *_mem_stat[index].proc* = 0 and delete page from the page table. If the table is empty, we also delete it. After finishing deleting the page, we go to the next

4.1.2.a Memory allocation

alloc_mem(): We allocate memory to a process by updating the ‘_mem_stat’ array, and updating the page table of the process

```
1  addr_t alloc_mem(uint32_t size, struct pcb_t *proc)
2  {
3      pthread_mutex_lock(&mem_lock);
4      addr_t ret_mem = 0;
5      /* TODO: Allocate [size] byte in the memory for the
6       * process [proc] and save the address of the first
7       * byte in the allocated memory region to [ret_mem].
8       * */
9
10     uint32_t num_pages = (size % PAGE_SIZE == 0) ? size / PAGE_SIZE : size /
        PAGE_SIZE + 1; // 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     uint32_t num_free_pages = 0;
22     for (int i = 0; i < NUM_PAGES; i++)
23     {
24         if (_mem_stat[i].proc == 0)
25         {
26             num_free_pages++;
27         }
28     }
29     if (num_free_pages >= num_pages && proc->bp + num_pages * PAGE_SIZE <
        RAM_SIZE)
```

```
30     mem_avail = 1;
31
32     if (mem_avail)
33     {
34         /* We could allocate new memory region to the process */
35         ret_mem = proc->bp;
36         proc->bp += num_pages * PAGE_SIZE;
37         /* Update status of physical pages which will be allocated
38          * to [proc] in _mem_stat. Tasks to do:
39          * - Update [proc], [index], and [next] field
40          * - Add entries to segment table page tables of [proc]
41          *   to ensure accesses to allocated memory slot is
42          *   valid. */
43         uint32_t num_allocated_pages = 0;
44         for (int p_i = 0, v_i = 0, prev_p_i = 0; p_i < NUM_PAGES; p_i++)
45         {
46             if (_mem_stat[p_i].proc == 0)
47             {
48                 _mem_stat[p_i].proc = proc->pid;
49                 _mem_stat[p_i].index = v_i;
50                 if (v_i != 0)
51                     _mem_stat[prev_p_i].next = p_i;
52
53                 addr_t physical_addr = p_i << OFFSET_LEN;
54                 addr_t seg_idx = get_first_lv(ret_mem + v_i * PAGE_SIZE);
55                 addr_t page_idx = get_second_lv(ret_mem + v_i * PAGE_SIZE);
56
57                 struct trans_table_t *table = get_trans_table(seg_idx, proc->
page_table);
58                 if (table)
59                 {
60                     table->table[table->size].v_index = page_idx;
61                     table->table[table->size].p_index = physical_addr >> OFFSET_LEN;
62                     table->size++;
63                 }
64                 else
65                 {
66                     /* This is the code that allocates a new page table for a new
segment. */
67                     struct page_table_t *t = proc->page_table;
68                     int n = t->size;
69                     t->size++;
70                     t->table[n].next_lv = (struct trans_table_t *)malloc(sizeof(struct
trans_table_t));
71                     t->table[n].next_lv->size++;
72                     t->table[n].v_index = seg_idx;
73                     t->table[n].next_lv->table[0].v_index = page_idx;
74                     t->table[n].next_lv->table[0].p_index = physical_addr >> OFFSET_LEN
;
75                 }
76
77                 prev_p_i = p_i;
78                 v_i++;
79                 num_allocated_pages++;
80                 if (num_allocated_pages == num_pages)
81                 {
82                     _mem_stat[prev_p_i].next = -1;
83                     break;
84                 }
85             }
86         }
87     }
```



```
88 pthread_mutex_unlock(&mem_lock);
89 // printf("-----Allocated-----\n");
90 // dump();
91 return ret_mem;
92 }
```

4.1.2.b Free memory

free_mem(): Free up the allocated memory area

```
1 int free_mem(addr_t address, struct pcb_t *proc)
2 {
3     /*TODO: Release memory region allocated by [proc]. The first byte of
4      * this region is indicated by [address]. Task to do:
5      * - Set flag [proc] of physical page use by the memory block
6      *   back to zero to indicate that it is free.
7      * - Remove unused entries in segment table and page tables of
8      *   the process [proc].
9      * - Remember to use lock to protect the memory from other
10     *   processes. */
11     pthread_mutex_lock(&mem_lock);
12     addr_t phys_addr;
13     if (translate(address, &phys_addr, proc))
14     {
15         int v_i = 0;
16         int index = phys_addr >> OFFSET_LEN;
17         struct page_table_t *t = proc->page_table;
18         while (index != -1)
19         {
20             _mem_stat[index].proc = 0;
21             addr_t seg_idx = get_first_lv(address + v_i * PAGE_SIZE);
22             addr_t page_idx = get_second_lv(address + v_i * PAGE_SIZE);
23             for (int n = 0; n < t->size; n++)
24             {
25                 if (t->table[n].v_index == seg_idx)
26                 {
27                     for (int m = 0; m < t->table[n].next_lv->size; m++)
28                     {
29                         if (t->table[n].next_lv->table[m].v_index == page_idx)
30                         {
31                             int k = 0;
32                             for (k = m; k < t->table[n].next_lv->size - 1; k++)
33                             {
34                                 t->table[n].next_lv->table[k].v_index = t->table[n].next_lv->
35                                 table[k + 1].v_index;
36                                 t->table[n].next_lv->table[k].p_index = t->table[n].next_lv->
37                                 table[k + 1].p_index;
38                             }
39                             t->table[n].next_lv->table[k].v_index = 0;
40                             t->table[n].next_lv->table[k].p_index = 0;
41                             t->table[n].next_lv->size--;
42                             break;
43                         }
44                     }
45                 }
46                 if (t->table[n].next_lv->size == 0)
47                 {
48                     free(t->table[n].next_lv);
49                     int m = 0;
50                     for (m = n; m < t->size - 1; m++)
51                     {
52                         t->table[m].v_index = t->table[m + 1].v_index;
```

```
50         t->table[m].next_lv = t->table[m + 1].next_lv;
51     }
52     t->table[m].v_index = 0;
53     t->table[m].next_lv = NULL;
54     t->size--;
55 }
56     break;
57 }
58 }
59     v_i++;
60     index = _mem_stat[index].next;
61 }
62 }
63 pthread_mutex_unlock(&mem_lock);
64 // printf("-----Freed-----\n");
65 // dump();
66 return 0;
67 }
```

4.2 Result

Here is our output of mem.c

```
1 ----- MEMORY MANAGEMENT TEST 0 -----
2 ./mem input/proc/m0
3 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
4   003e8: 15
5 001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
6 002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
7 003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
8 004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
9 005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
10 006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
11 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
12   03814: 66
13 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
14 NOTE: Read file output/m0 to verify your result
15 ----- MEMORY MANAGEMENT TEST 1 -----
16 ./mem input/proc/m1
17 NOTE: Read file output/m1 to verify your result (your implementation should print
    nothing)
```

To show the status of RAM after each memory allocation and deallocation function as it call in the requirement, we call dump() before exiting alloc_mem() and free_mem(). So this is the output.

```
1 ----- MEMORY MANAGEMENT TEST 0 -----
2 ./mem input/proc/m0
3 -----Allocated-----
4 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
5 001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
6 002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
7 003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
8 004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
9 005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
10 006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
11 007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
12 008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
13 009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
14 010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
15 011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
16 012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
17 013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
18 -----Allocated-----
19 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
20 001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
21 002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
22 003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
23 004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
24 005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
25 006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
26 007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
27 008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
28 009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
29 010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
30 011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
31 012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
32 013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
33 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
34 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
35 -----Freed-----
```



```
36 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
37 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
38 -----Allocated-----
39 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
40 001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
41 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
42 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
43 -----Allocated-----
44 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
45 001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
46 002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
47 003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
48 004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
49 005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
50 006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
51 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
52 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
53 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
54 003e8: 15
55 001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
56 002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
57 003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
58 004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
59 005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
60 006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
61 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
62 03814: 66
63 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
64 NOTE: Read file output/m0 to verify your result
65 ----- MEMORY MANAGEMENT TEST 1 -----
66 ./mem input/proc/m1
67 -----Allocated-----
68 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
69 001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
70 002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
71 003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
72 004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
73 005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
74 006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
75 007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
76 008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
77 009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
78 010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
79 011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
80 012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
81 013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
82 -----Allocated-----
83 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
84 001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
85 002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
86 003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
87 004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
88 005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
89 006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
90 007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
91 008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
92 009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
93 010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
94 011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
95 012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
96 013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
97 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
```



```
98 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
99 -----Freed-----
100 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
101 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
102 -----Allocated-----
103 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
104 001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
105 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
106 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
107 -----Allocated-----
108 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
109 001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
110 002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
111 003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
112 004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
113 005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
114 006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
115 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
116 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
117 -----Freed-----
118 002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
119 003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
120 004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
121 005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
122 006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
123 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
124 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
125 -----Freed-----
126 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
127 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
128 -----Freed-----
129 NOTE: Read file output/m1 to verify your result (your implementation should print
      nothing)
```

4.3 Question

What is the advantage and disadvantage of segmentation with paging?

To reduce the size of the page table in RAM, we use Segmented Paging that combines both
A logical address contains:

- Segment index
- Page table index
- Offset

A physical address includes:

- Physical address
- Offset

We will have segment table and page table. CPU request memory in logical address form. MMU get the page table index based on segment index. When have a table index, MMU lookup on this table and find the actual index in physical memory

Advantages of segmentation with paging:

- It need less memory usage.
- The segment size limited page table size.
- Segment table has only one entry corresponding to one actual segment. (more higher security for the systems)
- No more External Fragmentation.
- It simplifies memory allocation.
- Reduce memory usage compared to the paging technique. Because it can reduce the size of the page table which is store in RAM. A program can be divided into segments and its segments have their page table. Therefore, the page table size will be reduced depending on the number of segments we divided
- Solve the external fragmentation problem because it uses a fix-sized page for the memory block. Every hole between allocated memory can be assigned to processes. It can be discontiguous, but it has a table for managing the pages

Disadvantages of segmentation with paging:

- There is a chance that it will cause Internal Fragmentation..
- The complexity leve is much more higher than paging
- There is internal fragmentation. This problem will occur when the memory required is not divisible by the page size. That is, a frame in RAM will not be used totally. For example, the page size is 5 byte and the program wants to allocate 11 bytes, there will be 3 frames allocated in the RAM and there is a frame that just needs 1 byte of it
- More complex than the paging technique because it requires the implementation of a page table and segment table



- Increase the memory access time because it has to access the segment table and the page table to reach the actual memory it needs
- The page table must be contiguous in the memory because we access the frame in RAM through the offset of the base address of the page table. This may lead to external fragmentation

5 PUT IT ALL TOGETHER

Compiling the whole source code:

```
1 ----- MEMORY MANAGEMENT TEST 0 -----
2 ./mem input/proc/m0
3 000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
4   003e8: 15
5 001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
6 002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
7 003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
8 004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
9 005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
10 006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
11 014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
12   03814: 66
13 015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
14 NOTE: Read file output/m0 to verify your result
15 ----- MEMORY MANAGEMENT TEST 1 -----
16 ./mem input/proc/m1
17 NOTE: Read file output/m1 to verify your result (your implementation should print
    nothing)
18 ----- SCHEDULING TEST 0 -----
19 ./os sched_0
20 Time slot 0
21   Loaded a process at input/proc/s0, PID: 1 PRI0: 0
22   CPU 0: Dispatched process 1
23 Time slot 1
24 Time slot 2
25   CPU 0: Put process 1 to run queue
26   CPU 0: Dispatched process 1
27 Time slot 3
28   Loaded a process at input/proc/s1, PID: 2 PRI0: 3
29 Time slot 4
30   CPU 0: Put process 1 to run queue
31   CPU 0: Dispatched process 2
32 Time slot 5
33 Time slot 6
34   CPU 0: Put process 2 to run queue
35   CPU 0: Dispatched process 2
36 Time slot 7
37 Time slot 8
38   CPU 0: Put process 2 to run queue
39   CPU 0: Dispatched process 2
40 Time slot 9
41 Time slot 10
42   CPU 0: Put process 2 to run queue
43   CPU 0: Dispatched process 2
44 Time slot 11
45   CPU 0: Processed 2 has finished
46   CPU 0: Dispatched process 1
47 Time slot 12
48 Time slot 13
49   CPU 0: Put process 1 to run queue
50   CPU 0: Dispatched process 1
51 Time slot 14
52 Time slot 15
53   CPU 0: Put process 1 to run queue
54   CPU 0: Dispatched process 1
55 Time slot 16
56 Time slot 17
```




```
57 CPU 0: Put process 1 to run queue
58 CPU 0: Dispatched process 1
59 Time slot 18
60 Time slot 19
61 CPU 0: Put process 1 to run queue
62 CPU 0: Dispatched process 1
63 Time slot 20
64 Time slot 21
65 CPU 0: Put process 1 to run queue
66 CPU 0: Dispatched process 1
67 Time slot 22
68 CPU 0: Processed 1 has finished
69 CPU 0 stopped
70 NOTE: Read file output/sched_0 to verify your result
71 ----- SCHEDULING TEST 1 -----
72 ./os sched_1
73 Loaded a process at input/proc/s0, PID: 1 PRI0: 1
74 Time slot 0
75 Time slot 1
76 CPU 0: Dispatched process 1
77 Time slot 2
78 Time slot 3
79 CPU 0: Put process 1 to run queue
80 CPU 0: Dispatched process 1
81 Time slot 4
82 Loaded a process at input/proc/s1, PID: 2 PRI0: 2
83 Time slot 5
84 CPU 0: Put process 1 to run queue
85 CPU 0: Dispatched process 2
86 Time slot 6
87 Loaded a process at input/proc/s2, PID: 3 PRI0: 2
88 Time slot 7
89 CPU 0: Put process 2 to run queue
90 CPU 0: Dispatched process 3
91 Loaded a process at input/proc/s3, PID: 4 PRI0: 4
92 Time slot 8
93 Time slot 9
94 CPU 0: Put process 3 to run queue
95 CPU 0: Dispatched process 4
96 Time slot 10
97 Time slot 11
98 CPU 0: Put process 4 to run queue
99 CPU 0: Dispatched process 4
100 Time slot 12
101 Time slot 13
102 CPU 0: Put process 4 to run queue
103 CPU 0: Dispatched process 4
104 Time slot 14
105 Time slot 15
106 CPU 0: Put process 4 to run queue
107 CPU 0: Dispatched process 4
108 Time slot 16
109 Time slot 17
110 CPU 0: Put process 4 to run queue
111 CPU 0: Dispatched process 4
112 Time slot 18
113 Time slot 19
114 CPU 0: Put process 4 to run queue
115 CPU 0: Dispatched process 4
116 Time slot 20
117 CPU 0: Processed 4 has finished
118 CPU 0: Dispatched process 2
```



```
119 Time slot 21
120 Time slot 22
121 CPU 0: Put process 2 to run queue
122 CPU 0: Dispatched process 3
123 Time slot 23
124 Time slot 24
125 CPU 0: Put process 3 to run queue
126 CPU 0: Dispatched process 2
127 Time slot 25
128 Time slot 26
129 CPU 0: Put process 2 to run queue
130 CPU 0: Dispatched process 3
131 Time slot 27
132 Time slot 28
133 CPU 0: Put process 3 to run queue
134 CPU 0: Dispatched process 2
135 Time slot 29
136 CPU 0: Processed 2 has finished
137 CPU 0: Dispatched process 3
138 Time slot 30
139 Time slot 31
140 CPU 0: Put process 3 to run queue
141 CPU 0: Dispatched process 3
142 Time slot 32
143 Time slot 33
144 CPU 0: Put process 3 to run queue
145 CPU 0: Dispatched process 3
146 Time slot 34
147 Time slot 35
148 CPU 0: Processed 3 has finished
149 CPU 0: Dispatched process 1
150 Time slot 36
151 Time slot 37
152 CPU 0: Put process 1 to run queue
153 CPU 0: Dispatched process 1
154 Time slot 38
155 Time slot 39
156 CPU 0: Put process 1 to run queue
157 CPU 0: Dispatched process 1
158 Time slot 40
159 Time slot 41
160 CPU 0: Put process 1 to run queue
161 CPU 0: Dispatched process 1
162 Time slot 42
163 Time slot 43
164 CPU 0: Put process 1 to run queue
165 CPU 0: Dispatched process 1
166 Time slot 44
167 Time slot 45
168 CPU 0: Put process 1 to run queue
169 CPU 0: Dispatched process 1
170 Time slot 46
171 CPU 0: Processed 1 has finished
172 CPU 0 stopped
173 NOTE: Read file output/sched_1 to verify your result
174 ----- OS TEST 0 -----
175 ./os os_0
176 Loaded a process at input/proc/p0, PID: 1 PRI0: 2
177 Time slot 0
178 CPU 1: Dispatched process 1
179 Time slot 1
180 Loaded a process at input/proc/p0, PID: 2 PRI0: 0
```



```
181 Time slot 2
182 CPU 0: Dispatched process 2
183 Loaded a process at input/proc/p0, PID: 3 PRI0: 0
184 Time slot 3
185 Loaded a process at input/proc/p0, PID: 4 PRI0: 0
186 Time slot 4
187 Time slot 5
188 Time slot 6
189 CPU 1: Put process 1 to run queue
190 CPU 1: Dispatched process 1
191 Time slot 7
192 Time slot 8
193 CPU 0: Put process 2 to run queue
194 CPU 0: Dispatched process 3
195 Time slot 9
196 Time slot 10
197 CPU 1: Processed 1 has finished
198 CPU 1: Dispatched process 4
199 Time slot 11
200 Time slot 12
201 Time slot 13
202 Time slot 14
203 CPU 0: Put process 3 to run queue
204 CPU 0: Dispatched process 2
205 Time slot 15
206 Time slot 16
207 CPU 1: Put process 4 to run queue
208 CPU 1: Dispatched process 3
209 Time slot 17
210 Time slot 18
211 CPU 0: Processed 2 has finished
212 CPU 0: Dispatched process 4
213 Time slot 19
214 Time slot 20
215 CPU 1: Processed 3 has finished
216 CPU 1 stopped
217 Time slot 21
218 Time slot 22
219 CPU 0: Processed 4 has finished
220 CPU 0 stopped
221 NOTE: Read file output/os_0 to verify your result
222 ----- OS TEST 1 -----
223 ./os os_1
224 Time slot 0
225 Loaded a process at input/proc/p0, PID: 1 PRI0: 2
226 Time slot 1
227 CPU 2: Dispatched process 1
228 Time slot 2
229 Loaded a process at input/proc/p0, PID: 2 PRI0: 0
230 Time slot 3
231 CPU 2: Put process 1 to run queue
232 CPU 2: Dispatched process 1
233 CPU 1: Dispatched process 2
234 Loaded a process at input/proc/p0, PID: 3 PRI0: 0
235 CPU 0: Dispatched process 3
236 Time slot 4
237 Loaded a process at input/proc/p0, PID: 4 PRI0: 0
238 CPU 3: Dispatched process 4
239 Time slot 5
240 CPU 2: Put process 1 to run queue
241 CPU 1: Put process 2 to run queue
242 CPU 0: Put process 3 to run queue
```



```
243 CPU 2: Dispatched process 1
244 CPU 0: Dispatched process 3
245 CPU 1: Dispatched process 2
246 Loaded a process at input/proc/p0, PID: 5 PRI0: 0
247 Time slot 6
248 Loaded a process at input/proc/p0, PID: 6 PRI0: 0
249 CPU 3: Put process 4 to run queue
250 Time slot 7
251 CPU 3: Dispatched process 5
252 CPU 2: Put process 1 to run queue
253 CPU 1: Put process 2 to run queue
254 CPU 0: Put process 3 to run queue
255 CPU 2: Dispatched process 1
256 CPU 0: Dispatched process 6
257 CPU 1: Dispatched process 4
258 Loaded a process at input/proc/p0, PID: 7 PRI0: 0
259 Time slot 8
260 Loaded a process at input/proc/p0, PID: 8 PRI0: 0
261 CPU 3: Put process 5 to run queue
262 Time slot 9
263 CPU 3: Dispatched process 3
264 CPU 2: Put process 1 to run queue
265 CPU 1: Put process 4 to run queue
266 CPU 0: Put process 6 to run queue
267 CPU 2: Dispatched process 1
268 CPU 1: Dispatched process 2
269 CPU 0: Dispatched process 7
270 Time slot 10
271 CPU 3: Put process 3 to run queue
272 Time slot 11
273 CPU 3: Dispatched process 8
274 CPU 2: Processed 1 has finished
275 CPU 1: Put process 2 to run queue
276 CPU 2: Dispatched process 5
277 CPU 0: Put process 7 to run queue
278 CPU 1: Dispatched process 4
279 CPU 0: Dispatched process 6
280 Time slot 12
281 CPU 3: Put process 8 to run queue
282 Time slot 13
283 CPU 3: Dispatched process 3
284 CPU 1: Put process 4 to run queue
285 CPU 2: Put process 5 to run queue
286 CPU 0: Put process 6 to run queue
287 CPU 1: Dispatched process 2
288 CPU 2: Dispatched process 7
289 CPU 0: Dispatched process 8
290 Time slot 14
291 CPU 3: Put process 3 to run queue
292 Time slot 15
293 CPU 3: Dispatched process 4
294 CPU 2: Put process 7 to run queue
295 CPU 1: Put process 2 to run queue
296 CPU 0: Put process 8 to run queue
297 CPU 2: Dispatched process 5
298 CPU 0: Dispatched process 6
299 CPU 1: Dispatched process 3
300 Time slot 16
301 CPU 3: Put process 4 to run queue
302 Time slot 17
303 CPU 3: Dispatched process 7
304 CPU 1: Processed 3 has finished
```



```
305 CPU 2: Put process 5 to run queue
306 CPU 0: Put process 6 to run queue
307 CPU 1: Dispatched process 8
308 CPU 2: Dispatched process 2
309 CPU 0: Dispatched process 4
310 Time slot 18
311 CPU 3: Put process 7 to run queue
312 CPU 3: Dispatched process 6
313 CPU 2: Processed 2 has finished
314 CPU 1: Put process 8 to run queue
315 Time slot 19
316 CPU 0: Processed 4 has finished
317 CPU 1: Dispatched process 5
318 CPU 0: Dispatched process 7
319 CPU 2: Dispatched process 8
320 Time slot 20
321 CPU 3: Put process 6 to run queue
322 CPU 3: Dispatched process 6
323 CPU 2: Put process 8 to run queue
324 Time slot 21
325 CPU 2: Dispatched process 8
326 CPU 1: Put process 5 to run queue
327 CPU 1: Dispatched process 5
328 CPU 0: Put process 7 to run queue
329 CPU 0: Dispatched process 7
330 Time slot 22
331 CPU 3: Processed 6 has finished
332 CPU 3 stopped
333 CPU 2: Processed 8 has finished
334 Time slot 23
335 CPU 2 stopped
336 CPU 1: Processed 5 has finished
337 CPU 1 stopped
338 CPU 0: Processed 7 has finished
339 CPU 0 stopped
340 NOTE: Read file output/os_1 to verify your result
```

And here is our make os output

```
1 ----- OS TEST 0 -----
2 ./os os_0
3 Time slot 0
4   Loaded a process at input/proc/p0, PID: 1 PRI0: 2
5   CPU 1: Dispatched process 1
6 Time slot 1
7   Loaded a process at input/proc/p0, PID: 2 PRI0: 0
8 Time slot 2
9   Loaded a process at input/proc/p0, PID: 3 PRI0: 0
10  CPU 0: Dispatched process 2
11 Time slot 3
12  Loaded a process at input/proc/p0, PID: 4 PRI0: 0
13 Time slot 4
14 Time slot 5
15 Time slot 6
16  CPU 1: Put process 1 to run queue
17  CPU 1: Dispatched process 1
18 Time slot 7
19 Time slot 8
20  CPU 0: Put process 2 to run queue
21  CPU 0: Dispatched process 3
22 Time slot 9
23 Time slot 10
24  CPU 1: Processed 1 has finished
```



```
25 CPU 1: Dispatched process 4
26 Time slot 11
27 Time slot 12
28 Time slot 13
29 Time slot 14
30 CPU 0: Put process 3 to run queue
31 CPU 0: Dispatched process 2
32 Time slot 15
33 Time slot 16
34 CPU 1: Put process 4 to run queue
35 CPU 1: Dispatched process 3
36 Time slot 17
37 Time slot 18
38 CPU 0: Processed 2 has finished
39 CPU 0: Dispatched process 4
40 Time slot 19
41 Time slot 20
42 CPU 1: Processed 3 has finished
43 CPU 1 stopped
44 Time slot 21
45 Time slot 22
46 CPU 0: Processed 4 has finished
47 CPU 0 stopped
48 NOTE: Read file output/os_0 to verify your result
49 ----- OS TEST 1 -----
50 ./os os_1
51 Time slot 0
52 Time slot 1
53 Loaded a process at input/proc/p0, PID: 1 PRI0: 2
54 CPU 0: Dispatched process 1
55 Time slot 2
56 Loaded a process at input/proc/p0, PID: 2 PRI0: 0
57 CPU 2: Dispatched process 2
58 Time slot 3
59 CPU 0: Put process 1 to run queue
60 CPU 0: Dispatched process 1
61 Loaded a process at input/proc/p0, PID: 3 PRI0: 0
62 Time slot 4
63 CPU 3: Dispatched process 3
64 Loaded a process at input/proc/p0, PID: 4 PRI0: 0
65 CPU 2: Put process 2 to run queue
66 CPU 2: Dispatched process 4
67 Time slot 5
68 CPU 1: Dispatched process 2
69 CPU 0: Put process 1 to run queue
70 Loaded a process at input/proc/p0, PID: 5 PRI0: 0
71 CPU 0: Dispatched process 1
72 CPU 3: Put process 3 to run queue
73 CPU 3: Dispatched process 5
74 Time slot 6
75 Loaded a process at input/proc/p0, PID: 6 PRI0: 0
76 CPU 2: Put process 4 to run queue
77 CPU 2: Dispatched process 3
78 Time slot 7
79 CPU 1: Put process 2 to run queue
80 CPU 0: Put process 1 to run queue
81 CPU 1: Dispatched process 6
82 CPU 0: Dispatched process 1
83 Loaded a process at input/proc/p0, PID: 7 PRI0: 0
84 Time slot 8
85 CPU 3: Put process 5 to run queue
86 CPU 3: Dispatched process 4
```



```
87 CPU 2: Put process 3 to run queue
88 CPU 2: Dispatched process 2
89 Loaded a process at input/proc/p0, PID: 8 PRI0: 0
90 Time slot 9
91 CPU 1: Put process 6 to run queue
92 CPU 1: Dispatched process 7
93 CPU 0: Put process 1 to run queue
94 CPU 0: Dispatched process 1
95 CPU 3: Put process 4 to run queue
96 Time slot 10
97 CPU 3: Dispatched process 5
98 CPU 2: Put process 2 to run queue
99 CPU 2: Dispatched process 3
100 CPU 1: Put process 7 to run queue
101 Time slot 11
102 CPU 1: Dispatched process 8
103 CPU 0: Processed 1 has finished
104 CPU 0: Dispatched process 6
105 CPU 2: Put process 3 to run queue
106 Time slot 12
107 CPU 2: Dispatched process 4
108 CPU 3: Put process 5 to run queue
109 CPU 3: Dispatched process 2
110 Time slot 13
111 CPU 1: Put process 8 to run queue
112 CPU 0: Put process 6 to run queue
113 CPU 1: Dispatched process 7
114 CPU 0: Dispatched process 3
115 CPU 2: Put process 4 to run queue
116 CPU 2: Dispatched process 5
117 Time slot 14
118 CPU 3: Put process 2 to run queue
119 CPU 3: Dispatched process 8
120 CPU 1: Put process 7 to run queue
121 Time slot 15
122 CPU 1: Dispatched process 6
123 CPU 0: Put process 3 to run queue
124 CPU 0: Dispatched process 4
125 CPU 2: Put process 5 to run queue
126 CPU 2: Dispatched process 2
127 Time slot 16
128 CPU 3: Put process 8 to run queue
129 CPU 3: Dispatched process 7
130 Time slot 17
131 CPU 1: Put process 6 to run queue
132 CPU 0: Put process 4 to run queue
133 CPU 1: Dispatched process 3
134 CPU 0: Dispatched process 5
135 CPU 2: Processed 2 has finished
136 Time slot 18
137 CPU 2: Dispatched process 8
138 CPU 3: Put process 7 to run queue
139 CPU 3: Dispatched process 6
140 CPU 1: Processed 3 has finished
141 Time slot 19
142 CPU 1: Dispatched process 4
143 CPU 0: Put process 5 to run queue
144 CPU 0: Dispatched process 7
145 CPU 2: Put process 8 to run queue
146 CPU 2: Dispatched process 5
147 Time slot 20
148 CPU 3: Put process 6 to run queue
```



```
149 CPU 3: Dispatched process 8
150 Time slot 21
151 CPU 1: Processed 4 has finished
152 CPU 1: Dispatched process 6
153 CPU 0: Put process 7 to run queue
154 CPU 0: Dispatched process 7
155 CPU 2: Processed 5 has finished
156 CPU 2 stopped
157 Time slot 22
158 CPU 3: Put process 8 to run queue
159 CPU 3: Dispatched process 8
160 CPU 1: Processed 6 has finished
161 Time slot 23
162 CPU 1 stopped
163 CPU 0: Processed 7 has finished
164 CPU 0 stopped
165 Time slot 24
166 CPU 3: Processed 8 has finished
167 CPU 3 stopped
168 NOTE: Read file output/os_1 to verify your result
```


5.1 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.

Synchronization is important in every operating system. It helps coordinates the execution of processes so that not two concurrent process can access the same shared data. Especially in the multi-process environment when several processes are activated at once and attempt to access the same shared data.

One of the particular example is the **Critical Section problem**. The problem occur when a shared data is accessed and modified by different process or thread in same time, result in data inconsistency, raise error during execution.

For specific, our OS is running under multi-process environment using the **multilevel feedback queue** scheduler system. While many processes are run concurrently, the scheduler use only one **same queues system** to determine which processes is allowed to run on which CPU. Therefore, task for synchronization is to protect the consistent of queues. For example, the queue size only contains 1 left slot while there are two or more processes need to be put in. Without synchronization, they first check for the queue size at the same time and both processes is valid to be put in. However the queue can contain one more process, so the second processes could be missed or error. But with the help of synchronization, no two processes can access queue at the same time, they have to wait for the other to finish their actions (check for queue size and enqueue) so that ensure the thread safe.

Secondly, a problem may occur within the memory. There will be conflicts in getting the allocated memory.

Suppose that two different threads try to allocate the same page to itself, the first thread may find that the page at location 0x00 has not been used and so does the second thread. Then, both threads decided to take that page for allocation. The result is that two processes will record the same memory frame as theirs. The `_mem_stat` table will only record one of the two threads as the true owner, causing one process to not have enough space for its memory. There may also be a problem in deallocating memory. For example, thread A tries to set the flag to indicate that the page is free. Then, thread B sees that page is free, so it gets that page to be allocated. However, thread A continues its job, which is to remove the unused entries in the page table and segment table of that page. Therefore, thread B will fail when it tries to reference that page.

Thirdly, if we do not use the mutex mechanism, there is no way the timer will work. The timeslot keeps incrementing despite unfinished tasks.



6 CONCLUSION

This assignment provides us with knowledge about how a simple operating system works by implementing Scheduler and Virtual Memory Engine(VME) and enhancing our teamwork skills. The combination of Scheduler and VME makes fundamental concepts of scheduling, synchronization, and memory management, allowing the CPU to run Multilevel Queue (MLQ) CPU Scheduling.



References

- [1] *Operating System Concepts*, Abraham Silberschatz, Greg Gagne, Peter B. Galvin, 10th Edition, John Wiley & Sons, 2018. ISBN1119439256, 9781119439257, 976 pages.
- [2] *Operating Systems: Three Easy Pieces*, Remzi H. Arpaci.Dusseau, Andrea C. Arpaci-Dusseau, CreateSpace Independent Publishing Platform, 2018. ISBN198508659X, 9781985086593, 714 pages.