Assignment report

SEM 212

OPERATING SYSTEMS

CLASS CC03 – GROUP 11

LECTURE INSTRUCTOR: NGUYỄN LÊ DUY LAI
LAB INSTRUCTOR: LÊ THANH VÂN

Group member
Nguyễn Hải Đăng 2052444
Trần Quốc Bảo 2052038
Nguyễn Thanh Bảo Danh 2052416

INTRODUCTION

An operating system (OS) is the software component of a computer system that is responsible for the management and coordination of activities and the sharing of the resources of the computer. The OS acts as a host for application programs that are run on the machine. As a host, one of the purposes of an OS is to handle the details of the operation of the hardware. This relieves application programs from having to manage these details and makes it easier to write applications. Almost all computers use an OS of some type.

OSs offer several services to application programs and users. Among them the two most important but invisible to users are memory managing and process scheduling.

Table of content

I. Scheduler

1. Mechanism:

Priority feedback queue works with the following mechanism:

- The loader creates a new process and assigns a new PCB to it, then reads and copies the program content to the text segment of the new process.
- The PCB is pushed to *read_queue* and waits for the CPU.
- Each process is allowed to run for a given period of time in round robin style, then paused and forced to be pushed to *run_queue* if it has not done its work during the period.
- The CPU then continues picking processes in priority orders from *ready_queue* to run until it is empty.
- When the *ready_queue* is empty, all the processes waiting at *run_queue* will be moved back to *ready_queue* so the CPU can continue running paused process again.

The cycle continues until all the processes are executed completely.

2. Implementation:

The following functions are needed to be completed:

- In file *queue.c*:
- *enqueue()*: add a new PCB to the queue.

```
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    /* TODO: put a new process to queue [q] */
    if(q->size < MAX_QUEUE_SIZE){
        q->proc[q->size] = proc;
        q->size++;
    }
    return NULL;
}
```

- *dequeue()*: searching for a process with the highest priority and getting it out of the queue.

```c
struct pcb_t * dequeue(struct queue_t * q) {
    /* TODO: return a pcb whose priority is the highest
     * in the queue [q] and remember to remove it from q
     * */
    int pos = 0;
    struct pcb_t* dump;

    //check if the queue is empty
    //if so pass this section and return NULL
    //if not, find the proc with highest priority
    if(q->size > 0){
        uint32_t max = q->proc[0]->priority;
        for(int i = 1; i < q->size; i++){
            if(q->proc[i]->priority > max){
                max = q->proc[i]->priority;
                pos = i;
            }
        }
        dump = q->proc[pos];

    //replace the dequeued proc with the last index proc and decrease size
        q->proc[pos] = q->proc[q->size-1];
        q->size--;

    return dump;
    }
    return NULL;
}
```

- In file *sched.c*:

- *get_proc()*: get a process from the *ready_queue*. If the *ready_queue* is empty, we move all the processes waiting in the *run_queue* back to it, then try getting processes again.

```c
struct pcb_t * get_proc(void) {
    struct pcb_t * proc = NULL;
    /*TODO: get a process from [ready_queue]. If ready queue
     * is empty, push all processes in [run_queue] back to
     * [ready_queue] and return the highest priority one.
     * Remember to use lock to protect the queue.
     * */
    pthread_mutex_lock(&queue_lock);
    if(ready_queue.size == 0){
        while(run_queue.size != 0){
            enqueue(&ready_queue, dequeue(&run_queue));
        }
    }
    proc = dequeue(&ready_queue);
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

3. Test result:

Compile code by makefile: **make sched**

Run the test: **make test_sched**

Result:

- file *sched_0*

Terminal:

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

Gantt chart:

CPU 0																							
Process		P1				P2				P1		P2		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	23

Average waiting time = 1

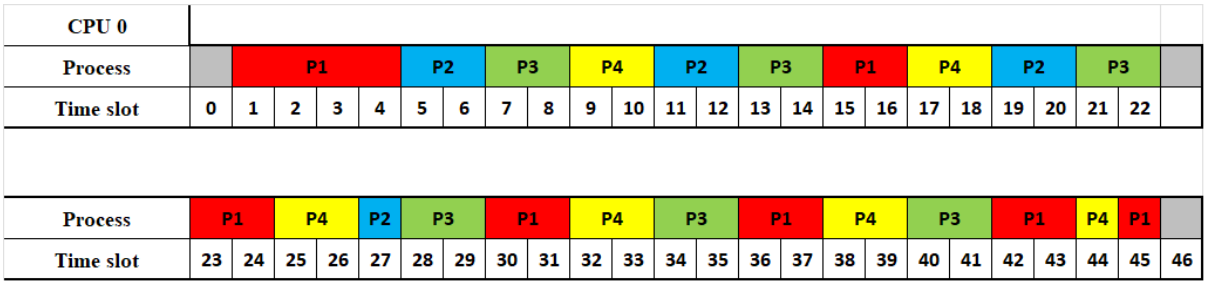
Average turnaround time = 17.5

- file *sched_1*

Terminal:

```
------ SCHEDULING TEST 1 ---------------------------------
./os sched_1
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s1, PID: 2
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/s2, PID: 3
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
        Loaded a process at input/proc/s3, PID: 4
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  10
Time slot  11
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  14
Time slot  15
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  16
Time slot  17
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  18
Time slot  19
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  20
Time slot  21
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  22
Time slot  23
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  24
Time slot  25
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  26
Time slot  27
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  28
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
Time slot  29
Time slot  30
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  31
Time slot  32
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  33
Time slot  34
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  35
Time slot  36
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  37
Time slot  38
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  39
Time slot  40
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  41
Time slot  42
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  1
Time slot  43
Time slot  44
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  45
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  46
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Gantt chart:



CPU 0																							
Process		P1				P2		P3		P4		P2		P3		P1		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	P1		P4		P2	P3		P1		P4		P3		P1		P4		P3		P1		P4	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	46

Average waiting time = 1.25

Average turnaround time = 36

4. Answer question:

Question: What is the advantage of using priority feedback queue in comparison with other scheduling
algorithms you have learned such as FIFO, Round Robin? Explain clearly your answer.

Answer:

Compared to FIFO and Round Robin:

- Priority feedback queue exploits processes based on prioritized aspects other than just the CPU burst time such as memory requirement, time requirement or user preference.
- With preemption, processes with higher priority will preempt the current running process at the end of the time slot.

Compared to Priority scheduling:

- Designed based on multilevel feedback queue, priority feedback queue can prevent the starvation problem. Starvation is a phenomenon usually associated with priority scheduling algorithms, in which a process might wait indefinitely because of low priority. Here, all the processes within the *ready_queue* must be executed first before the CPU continues with processes in *run_queue*, a feature which is not included in priority queue.

II. Memory management

1. Mechanism and implementation

In this assignment, the size of virtual RAM is 1 MB so we must use 20 bits to present the address of each byte. With paged segmentation, we use the first 5 bits for segment index, the next 5 for page index and the last remaining 10 bits for offset.

Next thing to do is translating virtual address of a process to physical one by implementing the following function in file *mem.c*:

- *get_page_table()*: Find the page table given a segment index of a process

```
static struct page_table_t * get_page_table(
        addr_t index,   // Segment level index
        struct seg_table_t * seg_table) { // first level table
    /*
    * TODO: Given the Segment index [index], you must go through each
    * row of the segment table [seg_table] and check if the v_index
    * field of the row is equal to the index
    *
    * */

    int i;
    for (i = 0; i < seg_table->size; i++) {
        // Enter your code here
        if(seg_table->table[i].v_index == index){
            return seg_table->table[i].pages;
        }
    }
    return NULL;

}
```

- *translate()*: use *get_page_table()* to convert from virtual address to physical address

```c
/* Translate virtual address to physical address. If [virtual_addr] is valid,
 * return 1 and write its physical counterpart to [physical_addr].
 * Otherwise, return 0 */
static int translate(
        addr_t virtual_addr,    // Given virtual address
        addr_t * physical_addr, // Physical address to be returned
        struct pcb_t * proc) {  // Process uses given virtual address

    /* Offset of the virtual address */
    addr_t offset = get_offset(virtual_addr);
    /* The first layer index */
    addr_t first_lv = get_first_lv(virtual_addr);
    /* The second layer index */
    addr_t second_lv = get_second_lv(virtual_addr);
```

```c
    /* Search in the first level */
    struct page_table_t * page_table = NULL;
    page_table = get_page_table(first_lv, proc->seg_table);
    if (page_table == NULL) {
        return 0;
    }

    int i;
    for (i = 0; i < page_table->size; i++) {
        if (page_table->table[i].v_index == second_lv) {
            /* TODO: Concatenate the offset of the virtual addess
             * to [p_index] field of page_table->table[i] to
             * produce the correct physical address and save it to
             * [*physical_addr]  */
            //concatenate_virtual_addr(virtual_addr, physical_addr, proc);
            uint32_t p_part = page_table->table[i].p_index << OFFSET_LEN;
            *physical_addr = p_part + offset;
            return 1;
        }
    }
    return 0;
}
```

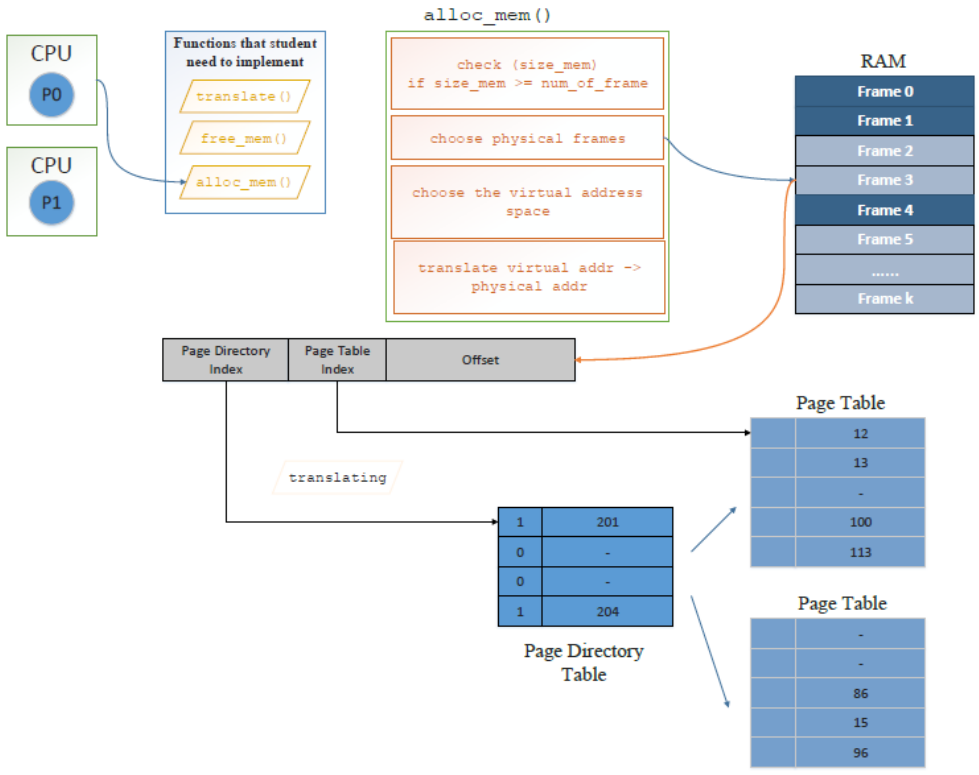
For memory allocation and deallocation, the OS maintains a special structure called *_mem_stat* to track the status of physical pages.

According to paged segmentation, RAM is splitted into multiple pages and the system allocates memory to processes by pages. The responsibility of *_mem_stat* is to maintain the status (used/free) of those pages:

- **struct** *_mem_stat:*

```
static struct {
    uint32_t proc;  // ID of process currently uses this page
    int index;  // Index of the page in the list of pages allocated
            // to the process.
    int next;   // The next page in the list. -1 if it is the last
            // page.
} _mem_stat [NUM_PAGES];
```

The next figure shows the allocation of new memory regions and creating new entries in the segment and page tables inside a process:



Next task is to implementing the following functions also in file *mem.c*:

- *alloc_mem()*: allocate memory to a process

```c
addr_t alloc_mem(uint32_t size, struct pcb_t * proc) {
    pthread_mutex_lock(&mem_lock);
    addr_t ret_mem = 0;

    uint32_t num_pages = ((size % PAGE_SIZE) == 0) ?
        size / PAGE_SIZE : size / PAGE_SIZE + 1; // Number of pages we will use
    int mem_avail = 0; // We could allocate new memory region or not?


    int i, spc_avail = 0;
    for (i = 0; i < NUM_PAGES; i++)
    {
        if (_mem_stat[i].proc == 0)
        {
            spc_avail++;
        }
        if (spc_avail == num_pages)
        {
            mem_avail = 1;
            break;
        }
    }
    if(proc->bp + num_pages * PAGE_SIZE > RAM_SIZE){ mem_avail = 0;}

    if (mem_avail) {
        /* We could allocate new memory region to the process */
        ret_mem = proc->bp;
        proc->bp += num_pages * PAGE_SIZE;
        addr_t vmem_addr = ret_mem;
        int spc_need = 0;
        int seg_idx = -1;
        int p_last = -1;
        int frame_prev = -1;
        for (i = 0; i < NUM_PAGES; i++){
            if(_mem_stat[i].proc == 0){
                _mem_stat[i].proc = proc->pid;
                _mem_stat[i].index = spc_need;
                if(frame_prev != -1){
                    _mem_stat[frame_prev].next = i;
                }
                frame_prev = i;
                seg_idx = get_first_lv(vmem_addr);
                if(proc->seg_table->table[seg_idx].pages == NULL){
                    proc->seg_table->table[seg_idx].pages = malloc(sizeof(struct page_table_t));
                    proc->seg_table->table[seg_idx].pages->size = 0;
                }
                proc->seg_table->table[seg_idx].pages->size++;
                p_last = proc->seg_table->table[seg_idx].pages->size - 1;

                proc->seg_table->table[seg_idx].v_index = seg_idx;
                proc->seg_table->table[seg_idx].pages->table[p_last].v_index = get_second_lv(vmem_addr);
                proc->seg_table->table[seg_idx].pages->table[p_last].p_index = i;

                vmem_addr += PAGE_SIZE;
                spc_need++;
                proc->seg_table->size++;
                if(spc_need == num_pages){
                    _mem_stat[i].next = -1;
                    break;
                }
            }
            else continue;
        }
    }
    pthread_mutex_unlock(&mem_lock);
    return ret_mem;
}
```

- *free_mem()*: free the allocated memory area

```
int free_mem(addr_t address, struct pcb_t * proc) {

    pthread_mutex_lock(&mem_lock);
    addr_t paddr;
    addr_t vaddr = address;
    int i;

    if(translate(address, &paddr, proc)){
        addr_t ppage = paddr >> OFFSET_LEN;

        while(ppage != -1){
            _mem_stat[ppage].proc = 0;
            addr_t seg_idx = get_first_lv(vaddr);
            for(i=0; i<proc->seg_table->table[seg_idx].pages->size; i++){
                if(proc->seg_table->table[seg_idx].pages->table[i].p_index == ppage){
                    proc->seg_table->table[seg_idx].pages->table[i].v_index = 0;
                    proc->seg_table->table[seg_idx].pages->table[i].p_index = 0;
                }
            }
            ppage = _mem_stat[ppage].next;
            vaddr += PAGE_SIZE;
        }
    }
    pthread_mutex_unlock(&mem_lock);
    return 0;
}
```

2. Test result:

Compile code by makefile: **make mem**

Run the test: **make test_mem**

Result:

Records of the state of RAM in the program:

```
------ 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
```

3. Answer question:

Question: In which system is segmentation with paging used (give an example of at least one system)?

Explain clearly the advantage and disadvantage of segmentation with paging.

Answer:

The Intel IA-32 systems use segmentation with paging with each segment can be 4GB

Advantages and disadvantages of segmentation with paging:

Advantages:

- Reducing page table size as it is limited to the number of segments. Therefore reducing memory requirements.
- Eliminating the external fragmentation as a fixed-size page is implemented.
- No need to swap the entire segment out.

Disadvantages:

- The problem of internal fragmentation still remains. If the memory required is less than or not divisible by the page size, there will be a RAM frame not put in use totally.
- Much more complex system to implement than normal paging technique.
- Longer memory access time because the system needs to access the segment table and the page table before reaching the needed memory.

III. Put it all together

Finally, we combine scheduler and Virtual Memory Engine to form a complete OS.

Since the OS runs on multiple processors, it is possible that shared resources could be concurrently accessed by more than one process at a time.

Therefore, it is essential to find shared resources and use lock mechanism to protect them.

Final test:

Compile code by makefile: **make all**

Run the test: **make test_all**

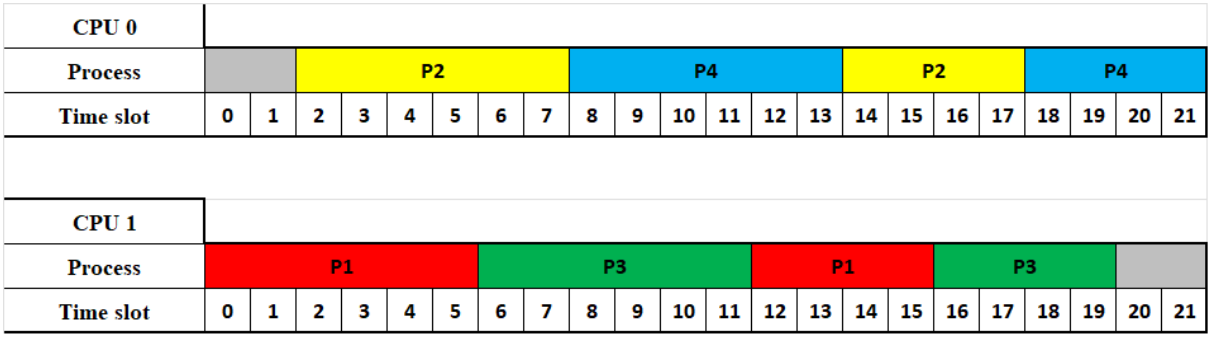
Result:

- file *os_0*

Terminal:

```
----- OS TEST 0 -----------------------------------------
./os os_0
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
        CPU 1: Dispatched process  1
Time slot   1
        Loaded a process at input/proc/p1, PID: 2
Time slot   2
        CPU 0: Dispatched process  2
Time slot   3
        Loaded a process at input/proc/p1, PID: 3
Time slot   4
        Loaded a process at input/proc/p1, PID: 4
Time slot   5
Time slot   6
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  3
Time slot   7
Time slot   8
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot   9
Time slot  10
Time slot  11
Time slot  12
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  1
Time slot  13
Time slot  14
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  15
Time slot  16
        CPU 1: Processed  1 has finished
        CPU 1: Dispatched process  3
Time slot  17
Time slot  18
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  19
Time slot  20
        CPU 1: Processed  3 has finished
        CPU 1 stopped
Time slot  21
Time slot  22
        CPU 0: Processed  4 has finished
        CPU 0 stopped
```

Gantt chart:



Memory state:

```
MEMORY CONTENT:
000: 00000-003ff - PID: 03 (idx 000, nxt: 001)
001: 00400-007ff - PID: 03 (idx 001, nxt: 002)
002: 00800-00bff - PID: 03 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 03 (idx 003, nxt: -01)
004: 01000-013ff - PID: 04 (idx 000, nxt: 005)
005: 01400-017ff - PID: 04 (idx 001, nxt: 006)
006: 01800-01bff - PID: 04 (idx 002, nxt: 012)
007: 01c00-01fff - PID: 02 (idx 000, nxt: 008)
008: 02000-023ff - PID: 02 (idx 001, nxt: 009)
009: 02400-027ff - PID: 02 (idx 002, nxt: 010)
       025e7: 0a
010: 02800-02bff - PID: 02 (idx 003, nxt: 011)
011: 02c00-02fff - PID: 02 (idx 004, nxt: -01)
012: 03000-033ff - PID: 04 (idx 003, nxt: -01)
014: 03800-03bff - PID: 03 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 03 (idx 001, nxt: 016)
016: 04000-043ff - PID: 03 (idx 002, nxt: 017)
       041e7: 0a
017: 04400-047ff - PID: 03 (idx 003, nxt: 018)
018: 04800-04bff - PID: 03 (idx 004, nxt: -01)
023: 05c00-05fff - PID: 02 (idx 000, nxt: 024)
024: 06000-063ff - PID: 02 (idx 001, nxt: 025)
025: 06400-067ff - PID: 02 (idx 002, nxt: 026)
026: 06800-06bff - PID: 02 (idx 003, nxt: -01)
047: 0bc00-0bfff - PID: 01 (idx 000, nxt: -01)
       0bc14: 64
057: 0e400-0e7ff - PID: 04 (idx 000, nxt: 058)
058: 0e800-0ebff - PID: 04 (idx 001, nxt: 059)
059: 0ec00-0efff - PID: 04 (idx 002, nxt: 060)
       0ede7: 0a
060: 0f000-0f3ff - PID: 04 (idx 003, nxt: 061)
061: 0f400-0f7ff - PID: 04 (idx 004, nxt: -01)
NOTE: Read file output/os_0 to verify your result
```

- file *os_1*

Terminal:

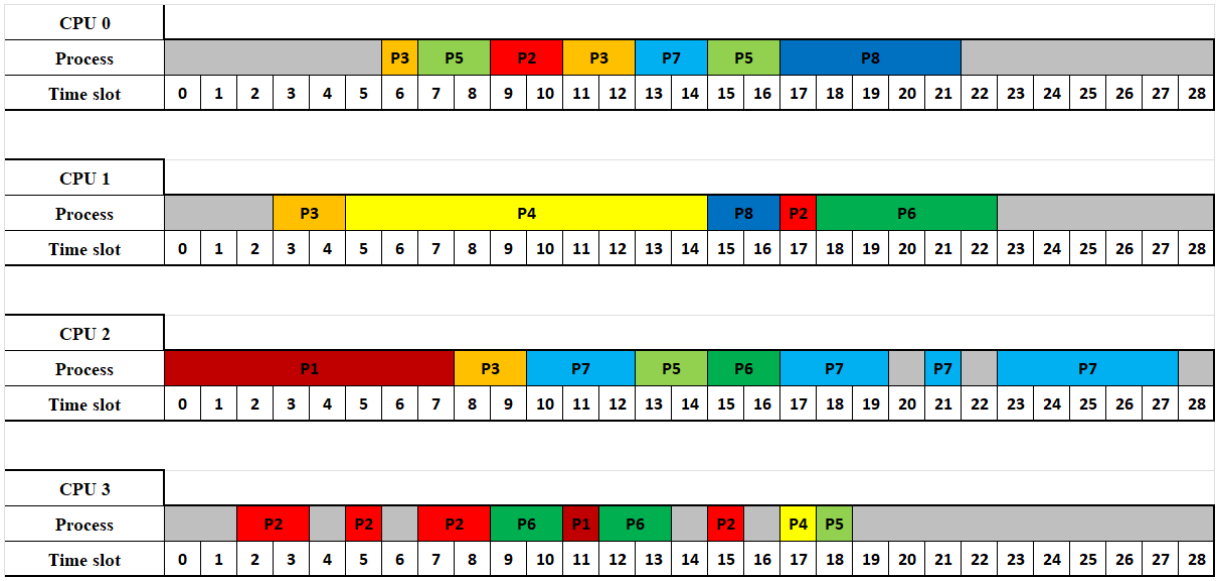
```
----- OS TEST 1 ----------------------------------------
./os os_1
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
        CPU 2: Dispatched process  1
Time slot   1
Time slot   2
        Loaded a process at input/proc/s3, PID: 2
        CPU 3: Dispatched process  2
Time slot   3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

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

```
Time slot  14
        CPU 3: Put process  6 to run queue
Time slot  15
        CPU 3: Dispatched process  2
        CPU 2: Put process  5 to run queue
        CPU 2: Dispatched process  6
        Loaded a process at input/proc/s1, PID: 8
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  8
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  5
Time slot  16
        CPU 3: Put process  2 to run queue
Time slot  17
        CPU 3: Dispatched process  4
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  7
        CPU 1: Put process  8 to run queue
        CPU 1: Dispatched process  2
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  8
Time slot  18
        CPU 3: Processed  4 has finished
        CPU 3: Dispatched process  5
        CPU 1: Processed  2 has finished
        CPU 1: Dispatched process  6
Time slot  19
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
        CPU 3: Processed  5 has finished
        CPU 3 stopped
Time slot  20
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
        CPU 2: Put process  7 to run queue
```

```
Time slot  21
        CPU 2: Dispatched process  7
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  6
Time slot  22
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
        CPU 2: Put process  7 to run queue
Time slot  23
        CPU 1: Processed  6 has finished
        CPU 0: Processed  8 has finished
        CPU 0 stopped
        CPU 2: Dispatched process  7
        CPU 1 stopped
Time slot  24
Time slot  25
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
Time slot  26
Time slot  27
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
Time slot  28
        CPU 2: Processed  7 has finished
        CPU 2 stopped
```

Gantt chart:



Memory state:

```
MEMORY CONTENT:
004: 01000-013ff - PID: 05 (idx 000, nxt: 005)
005: 01400-017ff - PID: 05 (idx 001, nxt: 006)
006: 01800-01bff - PID: 05 (idx 002, nxt: 009)
007: 01c00-01fff - PID: 05 (idx 000, nxt: 008)
        01fe8: 15
008: 02000-023ff - PID: 05 (idx 001, nxt: -01)
009: 02400-027ff - PID: 05 (idx 003, nxt: 010)
010: 02800-02bff - PID: 05 (idx 004, nxt: -01)
011: 02c00-02fff - PID: 06 (idx 000, nxt: 012)
012: 03000-033ff - PID: 06 (idx 001, nxt: 013)
013: 03400-037ff - PID: 06 (idx 002, nxt: 014)
014: 03800-03bff - PID: 06 (idx 003, nxt: -01)
021: 05400-057ff - PID: 01 (idx 000, nxt: -01)
        05414: 64
024: 06000-063ff - PID: 05 (idx 000, nxt: 025)
        06014: 66
025: 06400-067ff - PID: 05 (idx 001, nxt: -01)
031: 07c00-07fff - PID: 06 (idx 000, nxt: 032)
032: 08000-083ff - PID: 06 (idx 001, nxt: 033)
033: 08400-087ff - PID: 06 (idx 002, nxt: 034)
        085e7: 0a
034: 08800-08bff - PID: 06 (idx 003, nxt: 035)
035: 08c00-08fff - PID: 06 (idx 004, nxt: -01)
NOTE: Read file output/os_1 to verify your result
```

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

Answer:

A problem can occur in queues. When there are two threads left that need to enqueue, but the ready queue has only one slot, an error will happen. For further explanation, there is a case when thread 1 and thread 2 check the queue is not full (one slot left), after thread 1 enqueue validly, thread 2 will cause an error when enqueueing next. Familiar problems could occur when the queue is almost empty, but many CPUs try to dequeue.

In terms of memory allocation, if synchronization is not handle, there may be two thread decide to take the same page for allocation, leading to a conflict (error).

On the other hand, the timer will not work correctly without mutex mechanism. That leads to incrementation of timeslot although tasks are not done.

IV. CONCLUSION

The assignment has given us basic knowledge about how an operating system works as well as chances to practice implementing it. During the course of it, we have learnt the fundamental concepts of scheduling, memory management, synchronization, etc.

However, there is still a lot to be desired in both our work and the report so we would love to receive your feedback.