**HCMC University Of Technology**

**Faculty of Computer Science  Engineering**


## course: Operating Systems

# Assignment
# Simple Operating System

Instructor: Nguyễn Lê Duy Lai - Nguyễn Mạnh Thìn

Class: CC07

Supervised by:
Lương Thị Minh Oanh - 1950031
Nguyễn Đặng Tứ Hải - 1952045

Date submitted: 6th May 2022

# Contents

# 1 Scheduler

## 1.1 Question

### 1.1.1 Question

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

### 1.1.2 Answer

The Priority Feedback Queue (PFQ) algorithm uses the ideas of a number of other algorithms including Priority Scheduling - each process has a priority to execute, Multilevel Queue algorithm - uses multi-level process queues, Round Robin algorithm - uses quantum time for processes to execute. Here are other scheduling algorithms we have learned:

First Come First Served (FCFS)

Shortest Job First (SJF)

Priority Scheduling (PS)

Multilevel Queue Scheduling (MLQS)

Round Robin (RR)

Multilevel Feedback Queue (MLFQ)

Shortest Remaining Time First (SRTF)

Priority Feedback Queue (PFQ) use two queue: `ready_queue` and `run_queue`:

• `ready_queue`: This queue contains processes that will go to the CPU. And this queue's priority is higher than `run_queue`.
• `run_queue`: This queue contains the waiting processes to be executed after their time slot runs off. The processes in this queue only move to the `ready_queue` when the `ready_queue` is free.
• These two queues both have priority based on the process priority.

**Advantages of using priority feedback queue:**

- Avoiding starvation: Using the idea of RR with a quantum time. Each process is served by the CPU for a fixed time quantum. After that, the process will be moved to `run_queue` and the lower-priority processes can use the CPU. Therefore, we can avoid the problem that some processes have to wait for too long (starvation) in some methods such as:
+ SJF (Shortest Job First): Longer processes will have more waiting time, eventually they'll suffer starvation.
+ SRTF (Shortest Remaining Time First), Priority based scheduling: Processes with low priority may not be allowed to use the CPU.
- Using two queues, like the idea of MLQS and MLFQ, where two queues are cycle through processes until the process is completed, increasing the response time for processes (the Processes with lower priority that come later can still be executed before processes with higher priority after you have finished your slot).
- Processes are executed based on priority: Important process can be executed first by assigning its priority higher than those processes waiting in `ready_queue`. It helps us to control which process will be executed first, and which will be executed later. Some scheduling algorithms such as: FCFS (First Come First Served), SJF (Shortest Job First), SRTF (Shortest Remaining Time First), RR (Round Robin) can not do this

## 1.2 code

Firstly, enqueue() in queue.c: we only need to put a new process to the end of the queue [q] if it is not full, otherwise, we return and do nothing. Here is the implementation:

```
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++;
    }
}
```

Secondly, dequeue() in queue.c: we look for the element of the queue [q] with the highest priority. If the queue [q] is empty, we return NULL. Otherwise, we assign its position into variable position. Then we shift every element from position + 1 to the end of the queue [q] to its front one position. Finally, we get the process with the

highest priority is the variable result equal the element at the position in the queue [q].Here is the implementation:

```
struct pcb_t * dequeue(struct queue_t * q) {
    /* TODO: return a pcb whose prioprity is the highest
    * in the queue [q] and remember to remove it from q
    * */
    if (q->size == 0) return NULL;
    struct pcb_t* temp_proc = NULL;
    /* For finding the highest priority process in queue */
    int current_max_priority = q->proc[0]->priority;
    temp_proc = q->proc[0];
    /* For saving index while searching highest priority process */
    int found_index = 0;
    int i;
    for (i = 1; i < q->size; i++) {
        if (q->proc[i]->priority > current_max_priority) {
            temp_proc = q->proc[i];
            current_max_priority = q->proc[i]->priority;
            found_index = i;
        }
    }
    /* Re-Update queue */
    for (i = found_index; i < q->size - 1; i++)
        q->proc[i] = q->proc[i + 1];
    q->proc[q->size - 1] = NULL;
    --q->size;
    return temp_proc;
}
```

Finally, `get_proc` in shed.c: The task of the scheduler is to manage the updating of the processes that will be executed to the CPU. In this assignment, we just need to implement the function to find a process for the CPU to execute. In more details, with the function `get_proc()`, if the `ready_queue` is empty, we move all processes waiting in `run_queue` back to `ready_queue`. After that, we use dequeue() function to return the process that has the highest priority from `ready_queue`.
Here is the implementation:

```
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 is empty,
    * move all processes in run queue to ready queue
    */
    if (ready_queue.size == 0) {
        int i;
        for (i = 0; i < MAX_QUEUE_SIZE; i++){
            ready_queue.proc[i] = run_queue.proc[i];
            run_queue.proc[i] = NULL;
        }
        ready_queue.size = run_queue.size;
        run_queue.size = 0;
    }
    proc = dequeue(&ready_queue);
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

## 1.3 Results

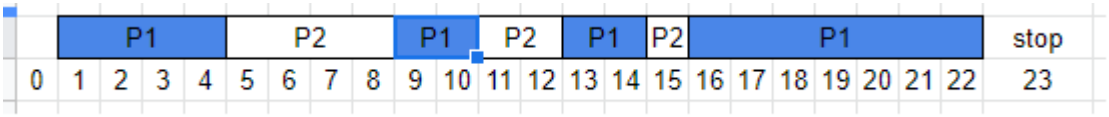
To open Terminal in Makefile, run "make sched" to check code, and run "make `test_sched`" to run program. The result of `sched_0` (an example of different results):

```
Time slot   0
Loaded a process at input/proc/s0, PID: 1
CPU 0: Dispatched process  1
Time slot   1
Time slot   2
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  1
Time slot   3
```

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

```
CPU 0: Dispatched process  1
Time slot  22
CPU 0: Processed  1 has finished
CPU 0 stopped
```

We use Gantt chart to illustrate this test ./os `sched_0`:




In this test, CPU runs on 2 processes which are P1 and P2 in 23 time slot as indicated on the Gantt chart above.

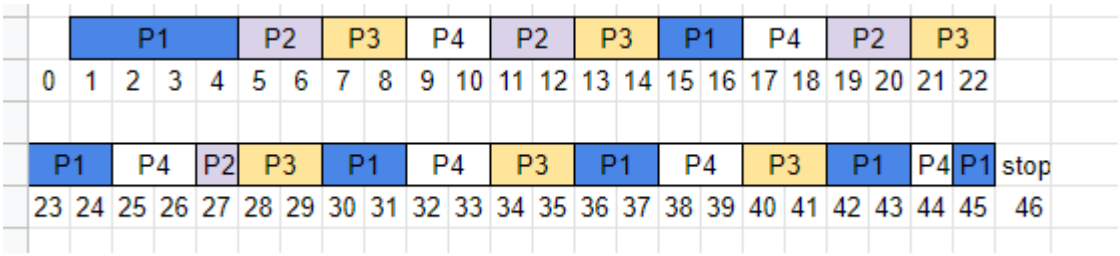
And the result of `sched_1`:

```
Time slot   0
Loaded a process at input/proc/s0, PID: 1
Time slot   1
CPU 0: Dispatched process  1
Time slot   2
Time slot   3
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  1
Time slot   4
Loaded a process at input/proc/s1, PID: 2
Time slot   5
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  2
Time slot   6
Loaded a process at input/proc/s2, PID: 3
Time slot   7
CPU 0: Put process  2 to run queue
CPU 0: Dispatched process  3
Loaded a process at input/proc/s3, PID: 4
Time slot   8
Time slot   9
CPU 0: Put process  3 to run queue
CPU 0: Dispatched process  4
Time slot  10
```

```
Time slot  11
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  2
Time slot  12
Time slot  13
CPU 0: Put process  2 to run queue
CPU 0: Dispatched process  3
Time slot  14
Time slot  15
CPU 0: Put process  3 to run queue
CPU 0: Dispatched process  1
Time slot  16
Time slot  17
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  4
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
```

We also use Gantt chart to illustrate this test ./os `sched_1`:




In this test, CPU runs on 4 processes which are P1, P2, P3 and P4 in 46 time slot as indicated on Gantt chart above.

# 2 Memory Management

## 2.1 Question

### 2.1.1 Question

What is the advantage and disadvantage of segmentation with paging.

### 2.1.2 Answer

Advantages:
- Saving memory
- Page size is limited by segment size
- Simplify memory allocation
- Overcome external fragmentation

Disadvantages:
- Causing internal fragmentation
- Complex level is higher than segmented paging
- Page table needs to store continuously

## 2.2 Implementation

### 2.2.1 Search for page table from the a segment table

In this assignment, address is represented by 20 bits, we use the first 5 bits for segment index, the next 5 bits for page index and the last 10 bits for offset. We

perform `get_page_table`() function in `mem.c` as below:

```
/* Search for page table table from the a segment table */
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
    *
    * */
    if (seg_table == NULL) return NULL;
    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;
}
```

From first 5 bits, which is index of segment table, if we find table that has `v_index equal` to index of transformed table, we will return corresponding page .

### 2.2.2 Translate virtual address into physical address

Each address contains 20 bits with mentioned organization. To create physical address, we take first 10 bits (segment and page) and connect it with last 10 bits (offset). Each `page_table_t` store elements with `p_index` of first 10 bits. We need to shift left 10 bits or multiple with `PAGE_SIZE` then add with offset. `translate`() function is implemented as below:

```
/* 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);

    /* 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] */
            *physical_addr = page_table->table[i].p_index * PAGE_SIZE + offset;
            return 1;
        }
    }
    return 0;
}
```

### 2.2.3 Memory allocation

Firstly, we check if both virtual and physical address are ready to use. In physical address, we find and count empty pages, if the quantity of empty pages exceeds the limitation, physical address is ready to use. In virtual address, we use Break Point of process, if it satisfy RAM-size condition, virtual address is ready to use. We create

`check_available_memory(unit32_tnum_page, struct * pcb_t proc):`

```
int check_available_memory(uint32_t num_pages, struct pcb_t *proc) {
    int num_of_valid_pages = 0;
    int i;
    for (i = 0; i < NUM_PAGES; i++) {
        if (_mem_stat[i].proc == 0) { // No process here
            num_of_valid_pages++;
        // If find enough space for current process size
        if (num_of_valid_pages == num_pages && proc->bp + num_pages * PAGE_
        SIZE <= RAM_SIZE) {
            return 1;
            break;
            }
        }
    }
    return 0;
}
```

if the result is 1, memory zone is enable, then we compile `alloc_mem()`

```
addr_t alloc_mem(uint32_t size, struct pcb_t* proc) {
    pthread_mutex_lock(&mem_lock);
    addr_t ret_mem = 0;
    /* TODO: Allocate [size] byte in the memory for the
        * process [proc] and save the address of the first
        * byte in the allocated memory region to [ret_mem].
        * */

    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?

    /* First we must check if the amount of free memory in
        * virtual address space and physical address space is
        * large enough to represent the amount of required
        * memory. If so, set 1 to [mem_avail].
        * Hint: check [proc] bit in each page of _mem_stat
```

```
    * to know whether this page has been used by a process.
    * For virtual memory space, check bp (break pointer).
    * */
mem_avail = check_available_memory(num_pages, proc);
if (mem_avail) {
    /* We could allocate new memory region to the process */
    ret_mem = proc->bp;
    proc->bp += num_pages * PAGE_SIZE;
    /* Update status of physical pages which will be allocated
            * to [proc] in _mem_stat. Tasks to do:
            * - Update [proc], [index], and [next] field
            * - Add entries to segment table page tables of [proc]
            * to ensure accesses to allocated memory slot is
            * valid. */
    int count_alloc_pages = 0;
    int prev_index = -1;
    addr_t virtual_addr;
    int first_lv, second_lv;
    int i;
    for (i = 0; i < NUM_PAGES; i++) {
        if (_mem_stat[i].proc) continue;

        // If found a free hole
        _mem_stat[i].proc = proc->pid;
        _mem_stat[i].index = count_alloc_pages;

        if (prev_index > -1)
            _mem_stat[prev_index].next = i;
        prev_index = i;

        int isFound = 0;
        struct seg_table_t *seg_table = proc->seg_table;
        // Seg_table is empty
        if (seg_table->table[0].pages == NULL)
            seg_table->size = 0;

        virtual_addr = ret_mem + (count_alloc_pages * PAGE_SIZE);
```

```
            // Contruct page tables
            first_lv = get_first_lv(virtual_addr);
            second_lv = get_second_lv(virtual_addr);
            int k;
            for (k = 0; k < seg_table->size; k++) {
                // If found a page table
                if (seg_table->table[k].v_index == first_lv) {
                    // Allocate new slot in current page table
                    struct page_table_t* cur_page_table = seg_table->table[k].pages;
                    cur_page_table->table[cur_page_table->size].v_index = second_lv;
                    cur_page_table->table[cur_page_table->size].p_index = i;
                    isFound = 1;
                    break;
                }
            }
            // When segment hasn't been made before
            if (isFound == 0) {
                // Allocate new page table slot
                seg_table->table[seg_table->size].v_index = first_lv;
                seg_table->table[seg_table->size].pages =
                (struct page_table_t*)malloc(sizeof(struct page_table_t));

                // Allocate new page translator in corresponding table
                seg_table->table[seg_table->size].pages->table[0].v_index = second_lv
                seg_table->table[seg_table->size].pages->table[0].p_index = i;

                seg_table->table[seg_table->size].pages->size = 1;

                seg_table->size++;
            }
            count_alloc_pages++;
            if (count_alloc_pages == num_pages) {
                _mem_stat[i].next = -1;
                break;
            }
        }
    }
    pthread_mutex_unlock(&mem_lock);
```

```
    return ret_mem;
}
```

### 2.2.4 Free memory

From transformed address, we find all allocated pages and delete them, then updating page table, if page table is deleted, we update seg table again. The content of `free_mem()`:

```
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);
    struct page_table_t* page_table = get_page_table(get_first_lv(address),
    proc->seg_table);

    int valid = 0;
    if (page_table != NULL) {
        int i;
        for (i = 0; i < page_table->size; i++) {
            if (page_table->table[i].v_index == get_second_lv(address)) {
            addr_t physical_addr;
                if (translate(address, &physical_addr, proc)) {
                    int p_index = physical_addr >> OFFSET_LEN; // 10 bits page index
                    int num_free_pages = 0;
                    addr_t cur_vir_addr = (num_free_pages << OFFSET_LEN) + address;
                    addr_t seg_idx, page_idx;
                    do {
                        _mem_stat[p_index].proc = 0;
                        int found = 0;
                        int k;
                        seg_idx = get_first_lv(cur_vir_addr);
```

```
            page_idx = get_second_lv(cur_vir_addr);
            for (k = 0; k < proc->seg_table->size && !found; k++) {
                if (proc->seg_table->table[k].v_index == seg_idx) {
                    int l;
                    for (l = 0; l < proc->seg_table->table[k].pages
                    ->size; l++) {
                        if (proc->seg_table->table[k].pages->table[l].
                        v_index == page_idx) {
                            int m;
                            //Rearrange page table
                            for (m = l; m <  proc->seg_table->table[k].
                            pages->size - 1; m++)
                                proc->seg_table->table[k].pages->table[m]
                                = proc->seg_table->table[k].pages->table
                                [m+1];
                            proc->seg_table->table[k].pages->size--;
                            //If page empty
                            if(proc->seg_table->table[k].pages->size==0)
                            {
                                free(proc->seg_table->table[k].pages);
                            //Rearrange segment table
                            for (m = k; m < proc->seg_table->size - 1;
                            m++)
                                proc->seg_table->table[m] =
                                proc->seg_table->table[m + 1];
                                proc->seg_table->size--;
                            }
                            found = 1;
                            break;
                        }
                    }
                }
            }

            p_index = _mem_stat[p_index].next;
            num_free_pages++;
        } while (p_index != -1);
        valid = 1;
```

```
                }
                break;
            }
        }
    }

    pthread_mutex_unlock(&mem_lock);
    return valid ? 0 : 1;
}
```

### 2.3 Test run

After compiling `alloc_mem()` and `free_mem()`, we run program by "make mem" trong Terminal. After that, run "make `test_mem`" to show result:

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
       003e8: 14
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: 64
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
```

## 3 Put all together

To ensure processes We will insert lock mechanism in `read_mem`() and `Write_mem`():

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

Next, run "make all" to check work by fist compiling the whole source code. Result of combination of scheduling and memory management:

```
Time slot   0
Loaded a process at input/proc/p0, PID: 1
Time slot   1
CPU 0: Dispatched process  1
Loaded a process at input/proc/p1, PID: 2
Time slot   2
Time slot   3
CPU 1: Dispatched process  2
Loaded a process at input/proc/p1, PID: 3
Time slot   4
Loaded a process at input/proc/p1, PID: 4
Time slot   5
Time slot   6
Time slot   7
CPU 0: Put process  1 to run queue
CPU 0: Dispatched process  4
Time slot   8
Time slot   9
CPU 1: Put process  2 to run queue
```

/

```
CPU 1: Dispatched process  3
Time slot  10
Time slot  11
Time slot  12
Time slot  13
CPU 0: Put process  4 to run queue
CPU 0: Dispatched process  1
Time slot  14
Time slot  15
CPU 1: Put process  3 to run queue
CPU 1: Dispatched process  2
Time slot  16
Time slot  17
CPU 0: Processed  1 has finished
CPU 0: Dispatched process  4
Time slot  18
Time slot  19
CPU 1: Processed  2 has finished
CPU 1: Dispatched process  3
Time slot  20
Time slot  21
CPU 0: Processed  4 has finished
CPU 0 stopped
Time slot  22
Time slot  23
CPU 1: Processed  3 has finished
CPU 1 stopped

MEMORY CONTENT:
000: 00000-003ff - PID: 04 (idx 000, nxt: 001)
001: 00400-007ff - PID: 04 (idx 001, nxt: 002)
002: 00800-00bff - PID: 04 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 04 (idx 003, nxt: -01)
004: 01000-013ff - PID: 03 (idx 000, nxt: 005)
005: 01400-017ff - PID: 03 (idx 001, nxt: 006)
006: 01800-01bff - PID: 03 (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: 03 (idx 003, nxt: -01)
014: 03800-03bff - PID: 04 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 04 (idx 001, nxt: 016)
016: 04000-043ff - PID: 04 (idx 002, nxt: 017)
041e7: 0a
017: 04400-047ff - PID: 04 (idx 003, nxt: 018)
018: 04800-04bff - PID: 04 (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: 03 (idx 000, nxt: 058)
058: 0e800-0ebff - PID: 03 (idx 001, nxt: 059)
059: 0ec00-0efff - PID: 03 (idx 002, nxt: 060)
0ede7: 0a
060: 0f000-0f3ff - PID: 03 (idx 003, nxt: 061)
061: 0f400-0f7ff - PID: 03 (idx 004, nxt: -01)
```

# References

[1] Operating System Concept - Abraham Silberschatz, Greg Gagne, Peter B. Galvin

[2] GNU Operating System "https://gnu.org/".

[3] Slides OS Concept - chapter 3, 4, 5, 6, 9, 10