HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
**FACULTY OF COMPUPTER SCIENCE AND TECHNOLOGY**
- - - - - - - - - - - -

# OPERATING SYSTEM

# BUILD A SIMULATED OPERATING SYSTEM

| | |
|---|---|
| **Student**: | NGUYEN Anh Khoa – 1611617 |
| | TRINH Thi Thu Thao – 1613232 |
| | TRAN Dang Khoi – 1611660 |
| | TRAN Quoc Dinh – 1610721 |
| **Class**: | L03 |
| **Teacher**: | PHAM Trung Kien |

## Contents

## Introduction

In this assignment, student are given a code that simulate the load and run of processes of an operating system. The code pre-given is incomplete and students need to implement as a practice.

There are three parts that the student must implement,

- Scheduling
- Memory management
- Overall app

Our group will go into details of how to implement the three section describe above.

## Scheduling

In this section, we must implement two things, the queue and the function to get a process.

### Queue

The queue is given in two files, queue.h as a header and queue.c as implementation. This part requires us to create a queue of process control block (PCB) as a priority queue. As the implementation, our group has create a max heap where the priority of the PCB is used to level the PCB.

### Get Process

The scheduling system is given in two files, sched.h as a header and sched.c as implementation. We have two global variables, the ready_queue, and run_queue. We also have a mutex, queue_lock. Our goal is to implement the missing function, `struct pcb_t* get_proc(void)`.

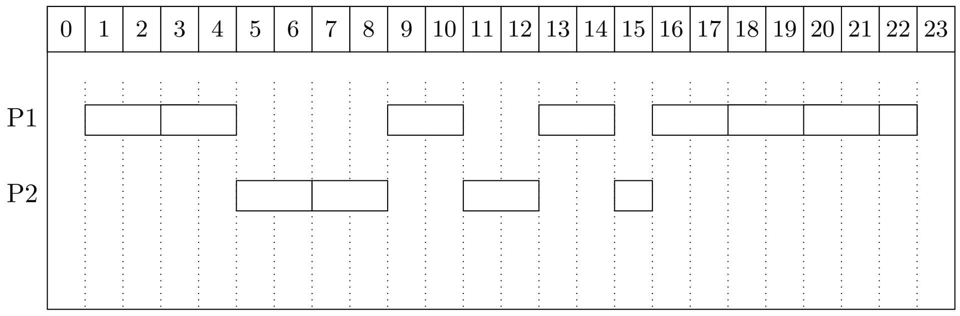
As describe in the assignment, the operating system uses a multilevel feedback queue, i.e Ready Queue, and Run Queue. The operating system always take the process from the Ready Queue and put it to Run Queue, when the Ready Queue is empty and Run Queue is not, the operating system must put all process in Run Queue back to Ready Queue, and get from Ready Queue.

```c
struct pcb_t* get_proc(void) {
        if (queue_empty())
                return NULL;

        pthread_mutex_lock(&queue_lock);
        if (empty(&ready_queue)) {
                while (!empty(&run_queue)) {
                        struct pcb_t* temp = dequeue(&run_queue);
                        enqueue(&ready_queue, temp);
                }
        }
        proc = dequeue(&ready_queue);
        pthread_mutex_unlock(&queue_lock);

        return proc;
}
```

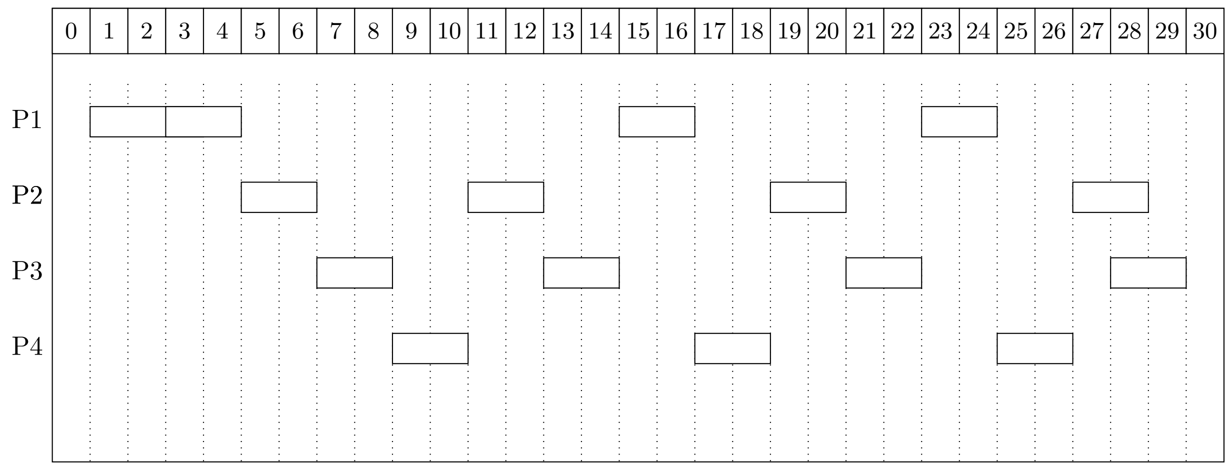
The schedule system is implemented, we run a few test to verify it's validity. The sample output from the assignment is the exact same with what we will print when run with the same input.

**Gantt diagram for scheduling**

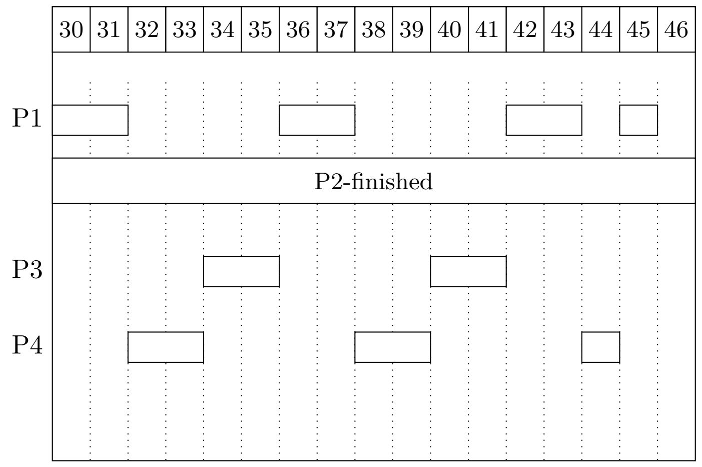
**sched**$_0$



**sched**$_1$$(part1)$



**sched**$_1$(*part*2)



## Memory

In this section, we must implement four things, get the page table of a process given the first level, translate a virtual address to a physical address, allocate ram for a process, free memory pages from an address.

### How virtual memory works

Before coding, it is good that we look again on how virtual memory works, in pagination, segmentation and a system using both.

### Pagination

In the pagination system, we will split the ram into sections, called a page. When allocating, we will give the process linearly-placed pages, pages are continuously placed one after another. This implementation causes an internal-fragmentation, when we allocate more than we really need. The pages often are very big in size, and giving away a page to a small program causes the system to waste valuable resources.

### Segmentation

In the segmentation, we give a part of the ram pages that may not continuously placed to the process. The process now will see itself as a collection of segment, i.e code segment, data segment… While this process solve the internal-fragmentation, it issued another problem, external-fragmentation, the ram is free but pages are not continuously aligned, and the system has to map the pages together to make the process see the memory as one big memory segment.

### Combination of both

When using the pagination, external-fragmentation is no issued, while using segmentation solve internal-fragmentation problem in the pagination method. We combine the two method to create a better memory management system.

The paging system is used to divide ram, we still give process pages of ram, but in a different way. When call for allocating a memory, we will search through ram to look for free-pages. Then we will chain these pages together to create a segment. In the segment, we will have a table of indexes of pages in the real ram that is allocated.

### Dealing with virtual memory

We must use virtual memory when we use a segment, because the pages might not continuously aligned. In the combination method, a process has first a table of segment, and in each segment, we have a table of pages map to physical pages. This is a two level index table, the common algorithm to manage the tables are a two index and offset method.

In this assignment, we are given the 20 bit address, the first 5 are segment index, the second 5 are the page index in the segment and the last 10 are offset.

We keep the index in the segment table and page table to quickly get the page, and the physical page index.

## Implementation

### Get page table

Given the index of the segment, we can get the page table quickly.

```
static struct page_table_t*
get_page_table(addr_t index,                    // Segment level index
               struct seg_table_t* seg_table) { // first level table
        int i;
        for (i = 0; i < seg_table->size; i++) {
                if (seg_table->table[i].v_index == index)
                        return seg_table->table[i].pages;
        }
        return NULL;
}
```

### Translate

From the virtual address, we can get the segment index (as first level), the page index (as second level), and the offset.

Given those we can get the physical index of a page in the page table where the page index is equal the second-level, and we can get a page from a segment through the `get_page_table` function above.

Because the physical index is only the index of pages in ram, we can get the real address by or-ing with the shifted index.

```
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) {
                        *physical_addr =
                            (page_table->table[i].p_index << OFFSET_LEN) |
                            offset;

                        return 1;
                }
        }
        return 0;
}
```

**Allocating**

The process of allocating can be dived into 6 steps:

1. Check for free pages
2. Get the correct page-table
3. Iterate the list of free pages to edit
4. If all pages are mapped, done
5. Update the virtual address to map
6. Go back step 2

In the steps above, step number 2 is the core. Our group has implement the get segment to get the segment of the process and from the segment we can get the pages accordingly to the second level of the virtual address.

```
struct {
        addr_t v_index;
        struct page_table_t* pages;
} * get_segment(addr_t first_lv, struct pcb_t* proc) {
        // get segment in process
        struct {
                addr_t v_index;
                struct page_table_t* pages;
        }* segment = NULL;
        int idx = 0;
        for (idx = 0; idx < proc->seg_table->size; idx++) {
                // find the seg_table with v_index = first_level
                if (first_lv == proc->seg_table->table[idx].v_index)
                        break;
        }

        if (idx == proc->seg_table->size) {
                proc->seg_table->size++;
        }

        segment = &(proc->seg_table->table[idx]);
        segment->v_index = first_lv;

        if (segment->pages == NULL) {
                segment->pages =
                    (struct page_table_t*)malloc(sizeof(struct page_table_t));
        }

        return segment;
}
```

And a declaration:

```
addr_t page_anchor = ret_mem;
addr_t first_lv = get_first_lv(page_anchor);
addr_t second_lv = get_second_lv(page_anchor);
addr_t new_first_lv = 0;

const int max_page = 1 << SEGMENT_LEN;
int* page_size = NULL;
struct {
addr_t v_index;
addr_t p_index;
}* page_table = NULL;

struct {
addr_t v_index;
struct page_table_t* pages;
}* segment = NULL;

segment = get_segment(first_lv, proc);
page_table = segment->pages->table;
page_size = &(segment->pages->size);
```

And step 3 to 6:

```
for (int mem_page = 0; mem_page < NUM_PAGES; mem_page++) {
      if (_mem_stat[mem_page].proc != 0)
            continue;

      // update _mem_stat
      _mem_stat[mem_page].proc = proc->pid;
      _mem_stat[mem_page].index = free_space++;
      if (free_space != 1)
            _mem_stat[prev].next = mem_page;
      if (free_space == num_pages)
```

```
            _mem_stat[mem_page].next = -1;
        prev = mem_page;

        // update in page_table
        page_table[*page_size].v_index = second_lv;
        page_table[*page_size].p_index = mem_page;
        (*page_size)++;

        if (free_space == num_pages)
            break;

        // move to next page
        page_anchor += PAGE_SIZE;
        new_first_lv = get_first_lv(page_anchor);
        second_lv = get_second_lv(page_anchor);

        if (new_first_lv != first_lv) {
            // get a new segment people
            first_lv = new_first_lv;
            segment = get_segment(first_lv, proc);
            page_table = segment->pages->table;
            page_size = &(segment->pages->size);
        }
}
```

The check on step 1 is easy to implement. Check for free pages on ram, and check for out of bound break pointer.

```
int free_space = 0;
for (int i = 0; i < NUM_PAGES; i++) {
    if (_mem_stat[i].proc == 0)
        free_space++;
    if (free_space == num_pages) {
        mem_avail = 1;
        break;
    }
}

// if break pointer is out of bound
if (proc->bp + num_pages * PAGE_SIZE > (1 << ADDRESS_SIZE)) {
    mem_avail = 1;
}

if (!mem_avail) {
    pthread_mutex_unlock(&mem_lock);
    return ret_mem;
}
```

```
// step 2 to 6
```

**Free**

As the `_mem_stat` table keeps a proc as a process id to indicate the use status, we can loop to find pages with proc is our pid and set clear.

```
int free_mem(addr_t address, struct pcb_t* proc) {
        addr_t physical_addr;
        if (translate(address, &physical_addr, proc)) {
                int i = physical_addr >> OFFSET_LEN;
                pthread_mutex_lock(&mem_lock);
                while (i != -1) {
                        _mem_stat[i].proc = 0;
                        i = _mem_stat[i].next;
                }
                pthread_mutex_unlock(&mem_lock);
                return 0;
        } else {
                return 1;
        }
        return 0;
}
```

**Checking**

After all is done, we run the memory check, the result map is identical so we are heading the right track.

We can see the status of ram after each call to `alloc` and `free`.

**m0**

```
Ram after free 00400 -> 00000
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
---
Ram after allocating 1 pages -> 03c00
000: 00000-003ff - PID: 01 (idx 000, nxt: -01)
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
---
Ram after allocating 4 pages -> 04000
000: 00000-003ff - PID: 01 (idx 000, nxt: -01)
001: 00400-007ff - PID: 01 (idx 000, nxt: 002)
002: 00800-00bff - PID: 01 (idx 001, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 002, nxt: 004)
```

```
004: 01000-013ff - PID: 01 (idx 003, nxt: -01)
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
---
000: 00000-003ff - PID: 01 (idx 000, nxt: -01)
    003e8: 15
001: 00400-007ff - PID: 01 (idx 000, nxt: 002)
002: 00800-00bff - PID: 01 (idx 001, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 002, nxt: 004)
004: 01000-013ff - PID: 01 (idx 003, nxt: -01)
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
    03414: 66
```

**m1**

```
Ram after allocating 13 pages -> 00400
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: -01)
---
Ram after allocating 1 pages -> 03800
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: -01)
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
---
```

```
Ram after free 00400 -> 00000
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
---
Ram after allocating 1 pages -> 03c00
000: 00000-003ff - PID: 01 (idx 000, nxt: -01)
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
---
Ram after allocating 4 pages -> 04000
000: 00000-003ff - PID: 01 (idx 000, nxt: -01)
001: 00400-007ff - PID: 01 (idx 000, nxt: 002)
002: 00800-00bff - PID: 01 (idx 001, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 002, nxt: 004)
004: 01000-013ff - PID: 01 (idx 003, nxt: -01)
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
---
Ram after free 03c00 -> 00000
001: 00400-007ff - PID: 01 (idx 000, nxt: 002)
002: 00800-00bff - PID: 01 (idx 001, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 002, nxt: 004)
004: 01000-013ff - PID: 01 (idx 003, nxt: -01)
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
---
Ram after free 04000 -> 00400
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
---
Ram after free 03800 -> 03400
---
```

## Overall view

The last task requires us to find critical section in the program and protect it, I have only found one.

```
static void* cpu_routine(void* args) {
    ...
        pthread_mutex_lock(&ahihi);
    time_left = time_slot;
    pthread_mutex_unlock(&ahihi);
    ...
}
```

## Conclusion

The operating system we implement uses the following which worth mentioning.

1. Multilevel feedback queue
2. Priority Queue
3. Segmentation with Pagination
4. Round Robin

Our operating system simulator using the Multilevel feedback queue to not make a process to be run multiple times. If there's only one queue, the process with a higher priority will be run over and over, because when it is enqueued, it is now the process with highest priority. The priority queue, keeps a good performance for our operating system. Segmentation use with Pagination will eliminate both down side about fragmentation of the two algorithms. Round Robin is effective as there will be no process use the operating system forever.