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

OPERATING SYSTEMS (CO2017)

Assignment

Simple Operating System

Advisor: Nguyễn Lê Duy Lai
Students: Nguyễn Minh Hùng - 1952737
Phạm Nhật Hoàng - 1952703
Bùi Quang Tiến - 1953018

HO CHI MINH CITY, SEPTEMBER 2021

Contents

1 Scheduler

1.1 Scheduling algorithms (Priority Feedback Queue)

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

Solution

The **Priority Feedback Queue** has a number of **distinct queues**, each assigned a **different priority level**. It allows a process to move between queues. Also, it uses the ideas of other scheduling algorithms we have learned for each queue such as: First Come First Served (FCFS), Round Robin (RR), Priority Scheduling (PS), Multilevel Feedback Queue, etc...

The following algorithms are what we have learned:

- First Come First Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF)
- Priority Scheduling (PS)
- Round Robin (RR)
- Multilevel Queue Scheduling (MLQS)
- Multilevel Feedback Queue (MLFQ)

In this assignment, we implemented 2 queues `ready_queue` and `run_queue` as follows:

- `ready_queue`: storing processes with assigned priority and the important point is that this has higher priority than `run_queue`'s.
- `run_queue`: storing processes that are wating to execute. In case `ready_queue` is empty, we push all into `ready_queue` to continue paused processes.

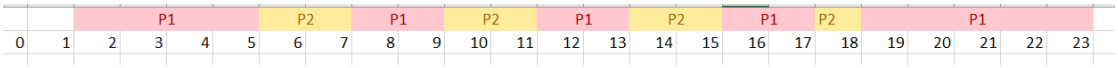
These two queues are both **Priority Queues**, the assigned priority are given depending on process characteristics.

Compared to others, PFQ has some advantages as follows:

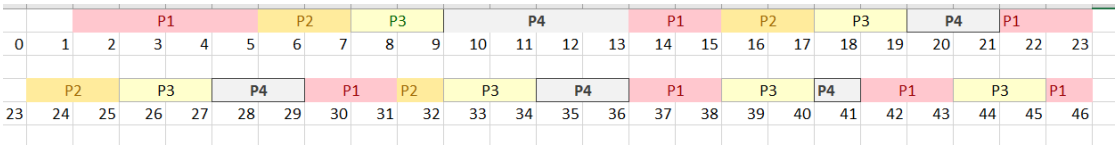
- The CPU runs processes in **round-robin** style. Every process gets an equal share of the CPU. RR is cyclic in nature, so there is **no starvation**.
- Using 2 priority queues with each assigned priority processes, it is based on MLQS and MLFQ. The processes are permanently assigned to the queue, so it has advantage of low scheduling overhead, moreover, it prevents starvation by moving a process that waits too long for lower priority queue (`run_queue`) to the higher priority queue (`ready_queue`).

1.2 Gantt diagrams results

REQUIREMENT: Draw Gantt diagram describing how processes are executed by the CPU.



Hình 1: Gantt chart for processes in test sched_0



Hình 2: Gantt chart for processes in test sched_1

You can also see the result in `my_output/sched.txt`

1.3 Implementation

1.3.1 Priority queue

The **Priority Queue** accepts the following properties:

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

In this assignment, we simplify it by implementing **ArrayQueue** with `MAX_QUEUE_SIZE` = 10 , as follows:

- `enqueue()` : just simply add to the end of array in O(1) time.
- `dequeue()` : linearly searching an item with highest priority, then removing the item by moving all subsequent items one position back.

The following is the implementation of `queue.c`

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

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;  // if queue size = 0, do nothing
  int highest_priority_idx = 0, j;
  for (j = 1; j < q->size; j++) { // traverse to find highest prior. index
     if (q->proc[j]->priority < q->proc[highest_priority_idx]->priority) {
        highest_priority_idx = j;
     }
  }
  struct pcb_t * res = q->proc[highest_priority_idx]; // get it out
  for (j = highest_priority_idx+1; j < q->size; j++) {
     q->proc[j-1] = q->proc[j]; // shift back to rearrange the queue
  }
  q->size--;

  return res;
}
```

1.3.2 Scheduler

The task of **scheduler** is to manage and update new process (in `loader()` function) which is described in Figure 2. However, in this assignment, we just need to complete finding a process for CPU with 2 available queues.

- `get_proc()`: return the process in `ready_queue`, if the queue is empty, we push all processes from `run_queue`. Then, if we continue to `dequeue` and get the highest priority process from `ready_queue`.

The following is the implementation of `sched.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 (empty(&ready_queue)) {
     // move all process is waiting in run_queue back to ready_queue
     while (!empty(&run_queue)) {
        enqueue(&ready_queue, dequeue(&run_queue));
     }
  }

  if (!empty(&ready_queue)) {
     proc = dequeue(&ready_queue);
  }
  pthread_mutex_unlock(&queue_lock);

  return proc;
}
```

2 Memory management

2.1 Segmentation with Paging

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

Solution

In **Segmentation with Paging**, the main memory is divided into variable size segments which are further divided into fixed size pages.

- Pages are smaller than segments.
- Each Segment has a page table which means every program has multiple page tables.
- The logical address is represented as **Segment Number (base address), Page number and page offset.**

Advantages of Segmentation with Paging

- It reduces memory usage.
- Page table size is limited by the segment size.
- Segment table has only one entry corresponding to one actual segment.
- External Fragmentation is not there.
- It simplifies memory allocation.

Disadvantages of Segmentation with Paging

- Internal Fragmentation will be there.
- The complexity level will be much higher as compare to paging.
- Page Tables need to be contiguously stored in the memory.

2.2 Memory status results (RAM)

REQUIREMENT: Show the status of RAM after each memory allocation and deallocation function call.

The following result illustrates MEM_STAT after each **allocation and deallocation** function call.

TEST 0

```
------ MEMORY MANAGEMENT TEST 0 --------------
./mem input/proc/m0
----------------ALLOCATION--------------------

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: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
----------------ALLOCATION--------------------

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: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
---------------DEALLOCATION-------------------

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
----------------ALLOCATION--------------------

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
----------------ALLOCATION--------------------
```

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
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)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
Final MEM stat:

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
```

TEST 1:

```
------ MEMORY MANAGEMENT TEST 1 --------------
./mem input/proc/m1
----------------ALLOCATION--------------------

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: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
----------------ALLOCATION--------------------
```

```
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: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
---------------DEALLOCATION------------------

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
----------------ALLOCATION-------------------

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
----------------ALLOCATION-------------------

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
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)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
---------------DEALLOCATION------------------

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)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
---------------DEALLOCATION------------------

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
```

```
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
--------------DEALLOCATION------------------

Final MEM stat:

NOTE: Read file output/m1 to verify your result (your implementation should
 print nothing)
```

2.3 Implementation

2.3.1 Search for page table: `get_page_table()`

In this assignment, we assumed that the size of virtual RAM is 1 MB so we must use 20 bit to represent the address of each of its byte including:

- First 5 bits for **segment index**.
- Next 5 bits for **page index**.
- Last 10 bits for **offset**.

This function receives `addr_t index` which is 5-bit segment level index and `struct seg_table_t * seg_table` which is first level table, the task is to find and return the page table.

According to **Segmentation with Paging**, we kept seg_table as a list of elements assigned to `v_index` (search key) and a pointer pointing to `page_table`. Hence, we traverse the list and we traverse and search using the given index.

The following is the implementation of `get_page_table()` :

```
/* 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

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

2.3.2 Address Translation Scheme: `translate()`

To translate from a given `addr_t virtual_addr` to `physical_addr`, we first search first level in the process to get its page table, then continue to search in the second level to get its `p_index`.

The `physical_addr` to return is created by first 10-bit `p_index` and 10-bit left for the `virtual_addr` offset.

Therefore, we shift left 10 bits and add with the offset.

The following is the implementation of `translation()` :

```
    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;
```

```
    }
```

2.3.3 Memory Allocation

2.3.3.a Check if Memory available to Allocate ?

First we have to check whether memory are available on both physical (`_mem_stat`) and virtual space (`break pointer`).

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

2.3.3.b Memory Allocation

- The variable `num_pages_alloc` counts for the number of pages which had been allocated.
- Var `prev_idx` is to keep track of previous index of page and to update the `mem_stat`
- Then, we create a current virtual address for this page

```
curr_v_addr = ret_mem + (num_pages_alloc << OFFSET_LEN);
```

- If update successful, we continue to save it in the process. To do that, we first get the page table and update the table by adding one row, in case we couldn't find the page table, just add new entry in `seg_table`.

```
int num_pages_alloc = 0;
addr_t curr_v_addr;
int seg_idx,page_idx;
int prev_idx;
for(i = 0; i < NUM_PAGES; i++){
   if(_mem_stat[i].proc == 0){     // page that can be used to
                           // allocate
      _mem_stat[i].proc = proc->pid;         /*------------------*/
      _mem_stat[i].index = num_pages_alloc; /*------------------*/
                                            /*------------------*/
      if(_mem_stat[i].index != 0){          /* update RAM status*/
         _mem_stat[prev_idx].next = i;     /*------------------*/
      }                                     /*------------------*/
      prev_idx = i;
      struct seg_table_t  *seg_table = proc->seg_table;
      if(!seg_table->table[0].pages) seg_table->size = 0;
      curr_v_addr = ret_mem + (num_pages_alloc << OFFSET_LEN);
      seg_idx = get_first_lv(curr_v_addr);
      page_idx = get_second_lv(curr_v_addr);
      struct page_table_t  *curr_page_table = get_page_table(seg_idx,
seg_table);
      if(curr_page_table){
         curr_page_table->table[curr_page_table->size].v_index = page_idx;
         curr_page_table->table[curr_page_table->size].p_index = i;
         curr_page_table->size++;
      }
      else{
         seg_table->table[seg_table->size].v_index = seg_idx;
         seg_table->table[seg_table->size].pages =(struct page_table_t*)
malloc (sizeof(struct page_table_t));
         seg_table->table[seg_table->size].pages->table[0].v_index =
page_idx;
         seg_table->table[seg_table->size].pages->table[0].p_index = i;
         seg_table->table[seg_table->size].pages->size = 1;
         seg_table->size++;
      }

      num_pages_alloc++;
      if(num_pages_alloc == num_pages) {  // last page in mem list
         _mem_stat[i].next = -1;
```

```
            break;
          }
       }
    }
}
```

2.3.4 Memory Deallocation

2.3.4.a Updating MEM_STAT

- Given an virtual address - `addr_t address` and process, we can translate it into **physical address**, using `translation()` method we have discussed above.
- If successful, we begin to traverse the list (_mem_stat) and update the memory by setting the flag [proc] to 0.

```
   addr_t   physic_address = 0;      // physical address to free
   /* check if valid address (had been allocated) */
   int isValid = translate(address,&physic_address,proc);
   if(!isValid) return 1; // invalid v_address
   /*if valid, */
   int p_index = physic_address >> OFFSET_LEN;    // remove OFFSET part
   int num_free_pages = 0;
   for(int i=p_index;i!=-1;i=_mem_stat[i].next){  // traverse and update
       num_free_pages++;                          // mem_stat
      _mem_stat[i].proc = 0;
   }
```

2.3.4.b Updating virtual space

Doing the same idea as **allocation**, we search and find the page table' s row that we need to free. If found, we remove it and rearrange the page table (shifting), however in case that after removing, the page table is **empty**, thus we should delete the table and rearrange the **segment table** (shifting).

```
   // clear virtual page stored in process
   for(int i=0;i<num_free_pages;i++){
      addr_t curr_v_address = address + PAGE_SIZE * i;
      int seg_index = get_first_lv(curr_v_address);
      int page_index = get_second_lv(curr_v_address);
      struct page_table_t  *page_table = get_page_table(seg_index,proc->
    seg_table);
```

```
    if(!page_table) {
       puts("-----------ERROR␣DEALLOCATION-----------\n");
       continue;
    }
    for(int j=0;j<page_table->size;j++){
       if(page_table->table[j].v_index == page_index){
          page_table->size--;
          for(int k = j; k<page_table->size;k++){
             page_table->table[k] = page_table->table[k+1];
          }
       }
    }
    if(page_table->size == 0){ // if empty after removing
       free(page_table);      // remove the whole page table
       int m;                 // and also remove the segment row that points
 to it
       for(m = 0;m<proc->seg_table->size;m++){
          if(seg_index == proc->seg_table->table[m].v_index) break;
       }
       int n;
       for(n = m; n<proc->seg_table->size-1; n++){
          proc->seg_table->table[n] = proc->seg_table->table[n + 1];
       }

       proc->seg_table->size--;
    }
}
```

2.3.4.c Updating break pointer - bp

Just in case we free the last block of virtual space (managed by bp, then we traverse back to update and stop when meets the using page.

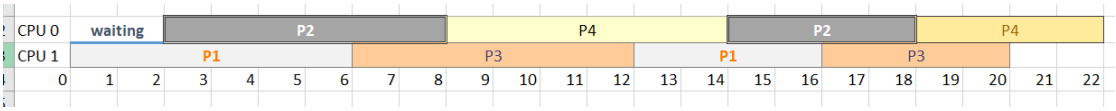
```
// Update break pointer
addr_t seg_page = address >> OFFSET_LEN;
if (seg_page + num_free_pages * PAGE_SIZE == proc->bp) {
   while (proc->bp >= PAGE_SIZE) {
       addr_t last_addr = proc->bp - PAGE_SIZE;
       addr_t last_segment = get_first_lv(last_addr);
       addr_t last_page = get_second_lv(last_addr);
       struct page_table_t * page_table = get_page_table(last_segment, proc
 ->seg_table);
       if (page_table == NULL) break;
       while (last_page >= 0) {
          int i;
          for (i = 0; i < page_table->size; i++) {
```

```
                if (page_table->table[i].v_index == last_page) {
                    proc->bp -= PAGE_SIZE;
                    last_page--;
                    break;
                }
            }
            if (i == page_table->size) break;
        }
        if (last_page >= 0) break;
    }
}
```

3 Put it all together

Finally, we combine scheduler and Virtual Memory Engine to form a complete OS. The overall result (**make all**) is extracted into `my_output/all.txt`. The following figure is the Gantt chart in general of all cases.

TEST `os_0`

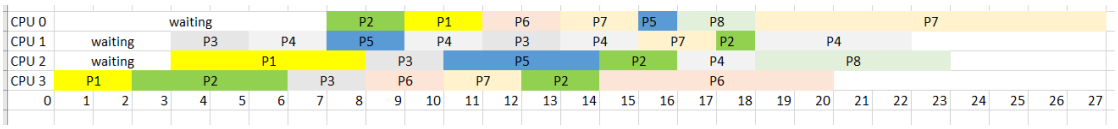


Hình 3: Gantt chart for processes in test os_0

```
# MEM status after doing allocation and deallocation of processes
   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
```

TEST os_1



Hình 4: Gantt chart for processes in test os_1

```
# MEM status after doing allocation and deallocation of processes
  MEMORY CONTENT:
000: 00000-003ff - PID: 05 (idx 000, nxt: 001)
001: 00400-007ff - PID: 05 (idx 001, nxt: 002)
002: 00800-00bff - PID: 05 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 05 (idx 003, nxt: 004)
004: 01000-013ff - PID: 05 (idx 004, nxt: -01)
009: 02400-027ff - PID: 06 (idx 000, nxt: 010)
010: 02800-02bff - PID: 06 (idx 001, nxt: 011)
011: 02c00-02fff - PID: 06 (idx 002, nxt: 012)
012: 03000-033ff - PID: 06 (idx 003, nxt: -01)
019: 04c00-04fff - PID: 01 (idx 000, nxt: -01)
  04c14: 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)
052: 0d000-0d3ff - PID: 05 (idx 000, nxt: 053)
  0d3e8: 15
053: 0d400-0d7ff - PID: 05 (idx 001, nxt: -01)
```

Danh sách hình vẽ

References

[1] ...

[2] ...