VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY

OPERATING SYSTEMS

Assignment

Simple Operating System

Advisor: Dr Lê Thanh Vân
Students:
Nguyễn Bá Minh Hưng(CC02) - 1952748
Phạm Mạnh Dũng(CC02) - 1952633
Võ Lê Hải Đăng(CC02) - 1852321

HO CHI MINH CITY, MAY 2021

Contents

1 Scheduler

1.1 Question - Priority Feedback Queue

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

In operating systems course, there are 7 scheduling algorithms that students have learned.

• First-Come, First-Served(FCFS) Scheduling

• Shortest-Job-First (SJF) Scheduling

• Shortest-Remaining-Time-First (SRTF)

• Round Robin (RR) Scheduling

• Priority Scheduling

• Multilevel Queue

• Multilevel Feedback Queue

Compared to those algorithms above, there are some advantages of using Priority Feedback Queue:

• Since the CPU runs processes in round-robin style, each process is allowed to run up to a given period of time. Therefore, every process gets an equal share of the CPU.

• Stravation could be prevented. Once the process has finished its given time in run_queue and move to ready_queue, it must wait all process in run_queue finish their turn before moving back to run_queue.

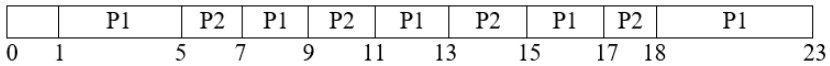
1.2 Result of Gantt Diagrams

• **Requirement:** Draw Gantt diagram describing how processes are executed by the CPU.

The following diagrams are Gantt diagram of executed processes in two test case.

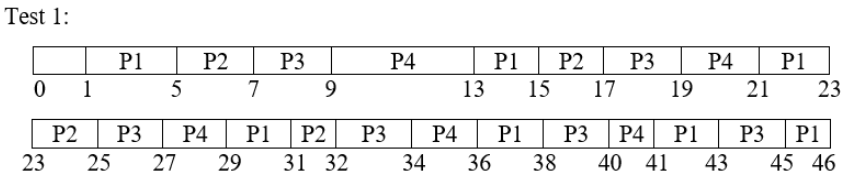
In test 0, the CPU execute two processes P1 and P2.

Test 0:



Firgue 1: *Gantt diagram – test 0*

In test 1, the CPU execute four processes P1, P2, P3 and P4.



Firgue 2: *Gantt diagram – test 1*

1.3 Implementation

1.3.1 Priority Queue

The following are two functions: **enqueue** and **dequeue** which are implemented to put a new process to the queue and to take the highest priority process out of the queue respectively.

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

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 (empty(q)) return NULL;
    else{
        int removePosition = 0;
        for (int i=0; i < (q->size - 1); i++){
            if (q->proc[i+1]->priority < q->proc[i]->priority){
                removePosition = i+1;
            }
        }
        struct pcb_t * temp = q->proc[removePosition];
        for (int i = removePosition + 1; i < q->size; i++){
            q->proc[i-1] = q->proc[i];
        }
        q->size--;

        return temp;
    }
}
```

1.3.2 Scheduler

The following is **get_proc** function. The task of this function is to 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.

```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)) {
      while(!empty(&run_queue))
          enqueue(&ready_queue, dequeue(&run_queue));
  }
    proc = dequeue(&ready_queue);
    pthread_mutex_unlock(&queue_lock);
  return proc;
}
```

2 Memory Management

2.1 Question - Segmentation with Paging

Question: What is the advantage and disadvantage of segmentation with paging.

_ **Advantage:**

- Exploit good properties of paging: no external fragmentation and faster allocation.
- Exploit good properties of segmentation: sharing and protection.
- Save and use memory efficiently.

_ **Disadvantage:**

- Internal fragmentation of paging still exists.

2.2 Status of RAM

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

The following are status of RAM in two test cases.

Test 0

```
------------==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)
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)
```

Test 1

```
------------==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==-----------
```

2.3 Implementation

2.3.1 Search for page table from segment table

In this assignment, we use the first 5 bits for segment index, the next 5 bits for page index and the last 10 bits for offset.

Function receive 5 bits index [index] and segment table [seg_table] and we will find out ap-

propriate page table from that segment table. In order to find the page table, we will compare the index in segment table [v_index] with the given index [index]. If the [v_index] equals to [index], we will return the appropriate table.

The following is the implementation.

```
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;
}
```

2.3.2 Translate virtual address to physical address

Each address contains 20 bits; therefore, to get the physical address , we will concatenate 10 bits of page and segment table with 10 bits of offset.

The following is the implementation of translating virtual address to physical address.

The following is the implementation of this function.

```
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]  */
        addr_t p_index = page_table->table[i].p_index;
        *physical_addr = (p_index << OFFSET_LEN) | offset;
        return 1;
      }
    }
    return 0;
}
```

2.3.3 Allocate memory

2.3.3.a Check amount of free memory in virtual address space and physical address space.

We will create a function to check if amount of free memory is large enough to allocate.

In physical address space, we will traverse each page of [_mem_stat] and check whether that page has been used by process. We will count the number of free pages and if the number of free pages are large enough, it means that physical address space are ready to allocate.

In virtual address space, we will check whether the break pointers are over allowed memory.

The following is the implementation of this function.

```
int check_amount_memmory(struct pcb_t * proc , uint32_t num_pages){

  uint32_t number_mem_free = 0;
  for(int i = 0 ; i < NUM_PAGES ; i++){
    if(_mem_stat[i].proc == 0){

      if(++number_mem_free == num_pages)
        break;
    }
  }
  if(number_mem_free < num_pages) return 0;
  if(proc->bp + num_pages*PAGE_SIZE >= RAM_SIZE) return 0;

  return 1;
}
```

2.3.3.b Allocate memory

To allocate memory, we will create the following function.

In this function, we traverse on the physical memory to search for free pages and then update [proc], [index] and [next] in [_mem_stat]. We use the last_alloc_pages to help update [next].

The following is the implementation of this function.

```
void alloc_ret_mem(addr_t ret_mem, uint32_t num_pages, struct pcb_t * proc){

    int count_pages = 0;  //count pages
    int last_alloc_pages = -1;      //get [next] = -1 at last page

    for(int i = 0 ; i< NUM_PAGES ; i++)
```

```
    {
      if(_mem_stat[i].proc != 0) continue; //page have used
      _mem_stat[i].proc = proc->pid;       // the page is used by process

      _mem_stat[i].index = count_pages;   //index list of allocated pages
      if(last_alloc_pages > -1){
        _mem_stat[last_alloc_pages].next = i;   //update last page

      }
      last_alloc_pages = i;     //update last page
      //find and creat virtual page_table
      addr_t virtual_address = ret_mem + count_pages*PAGE_SIZE;

      addr_t segment_addr = get_first_lv(virtual_address);
      addr_t page_addr = get_second_lv(virtual_address);
      struct page_table_t* v_page_table = get_page_table(segment_addr, proc->
    seg_table);
      if(v_page_table == NULL){
        proc->seg_table->table[proc->seg_table->size].v_index = segment_addr;
        v_page_table = proc->seg_table->table[proc->seg_table->size].pages
        = (struct page_table_t*)malloc(sizeof(struct page_table_t));
                proc->seg_table->size++;
      }
      int idx = v_page_table->size++;
      v_page_table->table[idx].v_index = page_addr;
      v_page_table->table[idx].p_index = i;
            ++count_pages;
      if(count_pages == num_pages){
        _mem_stat[i].next = -1; break;
      }

    }
}
```

2.3.4 Release memory

The following is implementation of release memory.

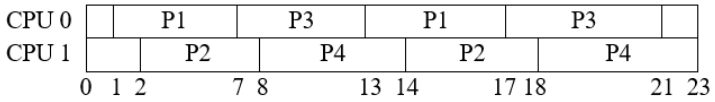
```
int free_mem(addr_t address, struct pcb_t * proc) {
  /**
   * TODO: Release memory region allocated by [proc].
   * The first byte of this region is indicated by [address].
   * Tasks to do:
   *  + Set flag [proc] of physical page use by the memory block
   *  + back to zero to indicate that it 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);
  addr_t v_address = address; // virtual address to free in process
  addr_t p_address = 0;   // physical address to free in memory
  // Find physical page in memory
  if (!translate(v_address, &p_address, proc)) return 1;
  // Clear physical page in memory
  addr_t p_segment_page_index = p_address >> OFFSET_LEN;
  int num_pages = 0; // number of pages in list
  int i;
  for (i=p_segment_page_index; i!=-1; i=_mem_stat[i].next) {
    num_pages++;
    _mem_stat[i].proc = 0; // clear physical memory
```

```
  }
  // Clear virtual page in process
  for (i = 0; i < num_pages; i++) {
    addr_t v_addr = v_address + i * PAGE_SIZE;
    addr_t v_segment = get_first_lv(v_addr);
    addr_t v_page = get_second_lv(v_addr);
    struct page_table_t * page_table = get_page_table(v_segment, proc->seg_table);
    if (page_table == NULL) {
      continue;
    }
    int j;
    for (j = 0; j < page_table->size; j++) {
      if (page_table->table[j].v_index == v_page) {
        int last = --page_table->size;
        page_table->table[j] = page_table->table[last];
        break;
      }
    }
    if (page_table->size == 0) {
      remove_page_table(v_segment, proc->seg_table);
    }
  }
  // Update break pointer
  proc->bp = proc->bp - num_pages*PAGE_SIZE;
  pthread_mutex_unlock(&mem_lock);
  return 0;
}
```

3 Overall

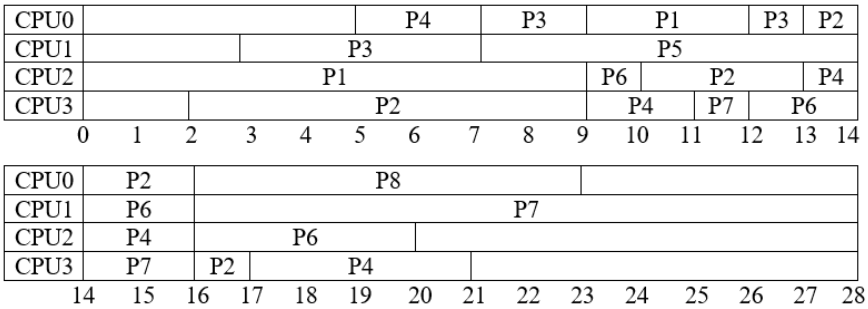
After finishing all implementation, we will put all the code together and run completely simple operating system. The following is the running process of each CPU in two different cases which are performed in Gantt diagram.

Test 0:



Firgue 1: *Gantt diagram – test 0*

Test 1:



Firgue 2: *Gantt diagram – test 1*