ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA KHOA HỌC - KỸ THUẬT MÁY TÍNH

OPERATING SYSTEMS

==

Assignment

Simple Operating System

==

Instructor : PhD.Nguyen Le Duy Lai
Class: CC05
Students :
Trinh Son Lam - 1852502

Contents

1. Scheduler

1.1. Question

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

Answer :

The Priority Feedback Queue (PFQ) algorithm uses each process has a priority to execute, particularly PFQ uses 2 queue are *ready_queue* and *run_queue* with meaning as follow:

- *ready_queue*: the queue contains processes that have a higher priority excution than *run_queue*. When the CPU moves to the next slot, it looks for the process in this queue.
- *run_queue*: the queue contains processes that are waiting to excute after their slot has not been completed yet. Processes in this queue can only continue to slot when *ready_queue* is free(idle) and is moved to the *ready_queue* to consider the next slot.
- Both queues are the priority queue, the priority is based on the priority of the process.

Advantage of using priority feedback queue:

- Using time slot, creating equity of execution time between processes, avoiding CPU usage, indefinitely delay.
- Using two queues and priority should be flexible in assigning tasks.
- Short-processes will quickly be completed, giving execution time to other processes.

1.2. Result

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

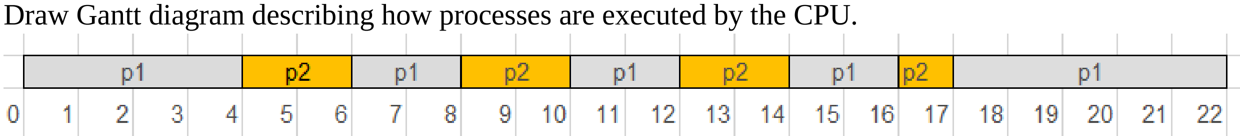
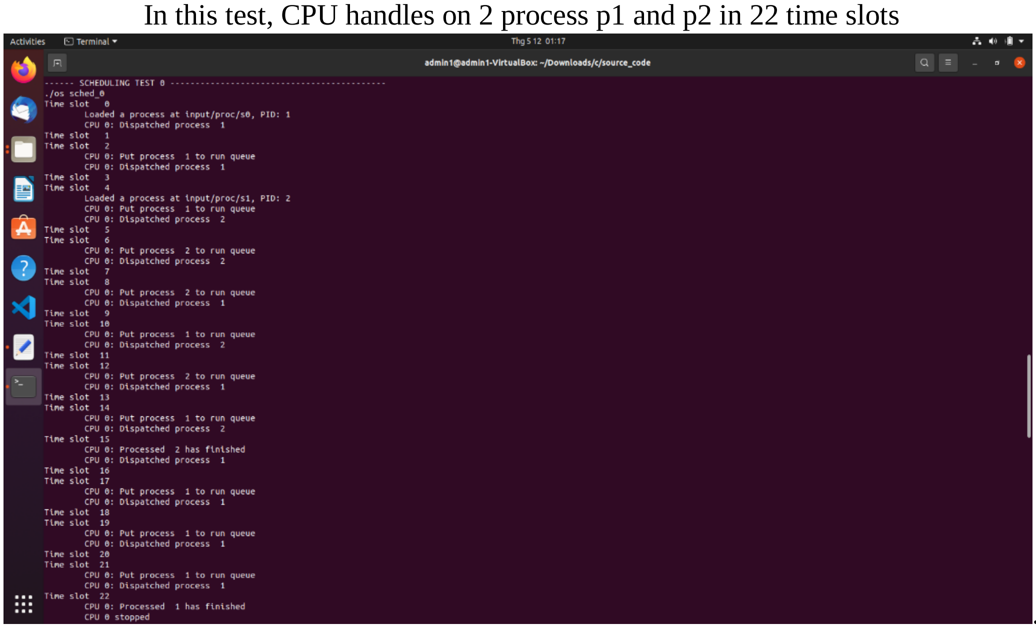


Figure1. Gantt CPU executes processes – test sched 0

In this test, CPU handles on 2 process p1 and p2 in 22 time slots


.

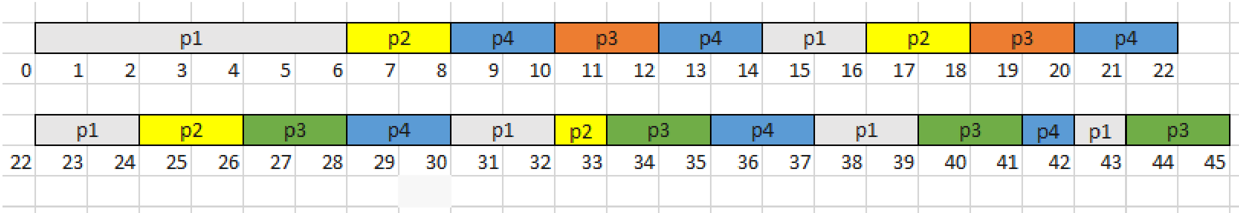
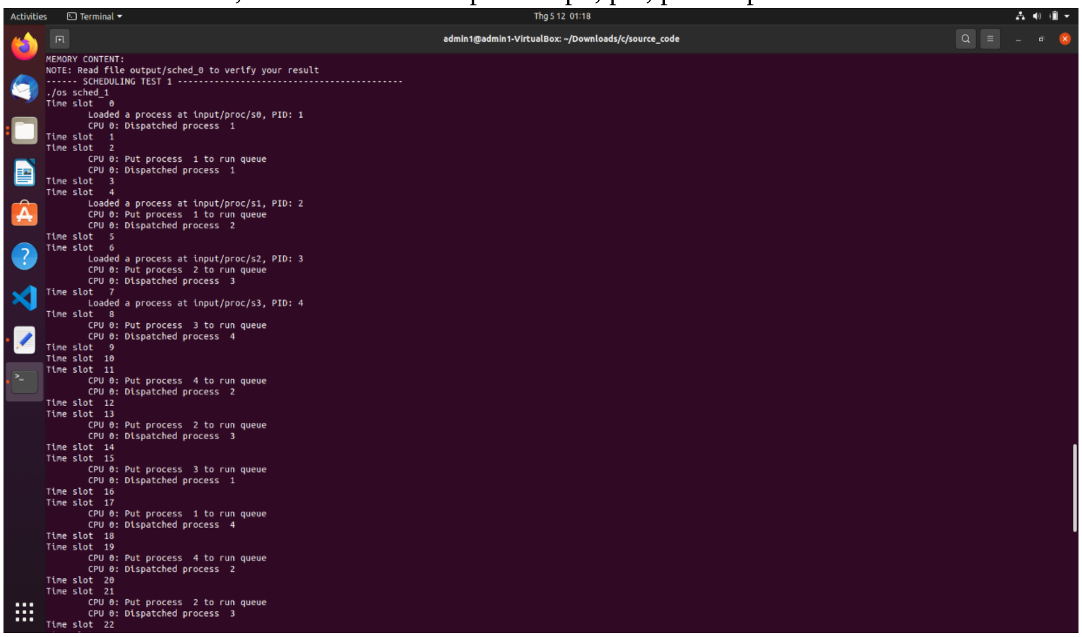
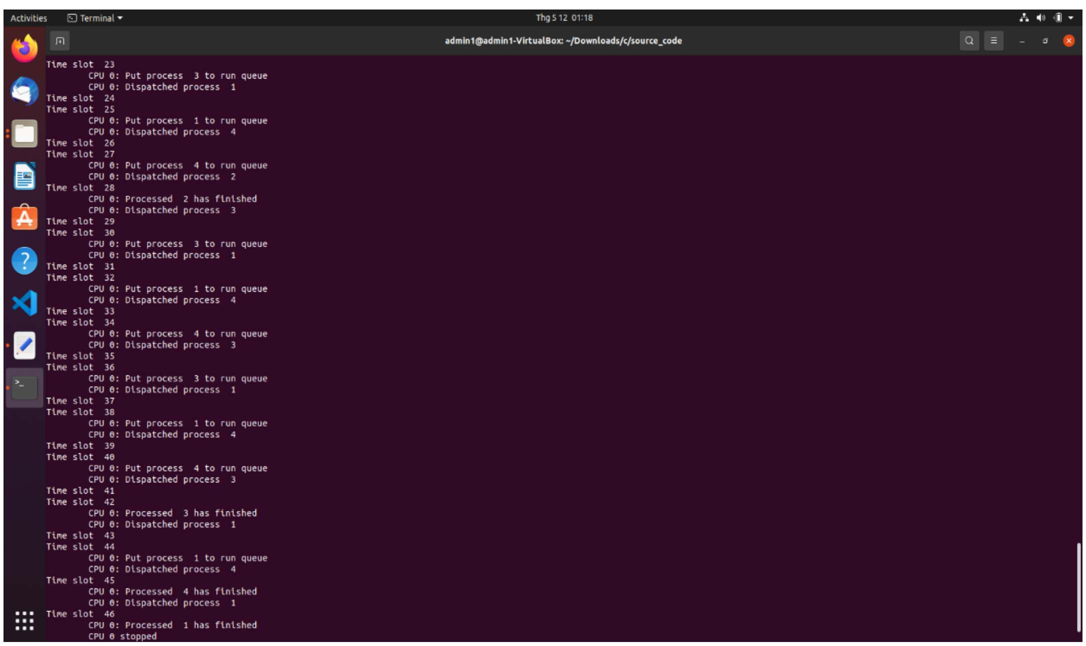


Figure2. Gantt CPU executes processes - test sched 1

In this test, CPU handles on 4 process p1, p2 , p3 and p4 in 46 time slots.





1.3. Implementation

1.3.1. Priority queue

The priority queue in this case handles no more than 10 processes, so we need to use the loop to handle the functionality that a priority queue needs. Specifically with the enqueue() function, we only add at the end of the queue if it is available (empty). With the dequeue () function, we search the process with the highest priority out, and at the same time update the queue's state when deleting an element. Below is the priority queue implementation for the scheduler**.**

```
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++;
        }
}

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)
        {
                struct pcb_t* temp_proc=NULL;
                int first_priority=0;
                int max_idx=0;
                int i;
                for(i=0;i<q->size;++i)
                {
                        if(q->proc[i]->priority>first_priority)
                        {
                                temp_proc=q->proc[i];
                                first_priority=temp_proc->priority;
                                max_idx=i;
                        }
                }
                for(i=max_idx;i<q->size-1;++i){
                        q->proc[i]=q->proc[i+1];
                }
                q->proc[q->size-1]=NULL;
                --q->size;
                return temp_proc;
        }
        return NULL;
}
```

1.3.2. Scheduler

The scheduler 's job is to manage updates to the processes that will be executed for the CPU. Specifically, the scheduler will manage two queues ready and run, we only need to execute the function to find a process for the CPU to execute. With the function get_proc (), which returns a process in the ready queue, if the ready queue is idle, we update the queue with the processes that are waiting for the next slot in the queue to run. By contrast, we find the process with high priority from this queue.

```
struct pcb_t * get_proc(void) {
        struct pcb_t * process = 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 process is waiting in run_queue back to ready_queue
                while (!empty(&run_queue)) {
                        enqueue(&ready_queue, dequeue(&run_queue));
                }
        }

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

        return process;
}
```

2. Memory Management

2.1. Question

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

Advantages :

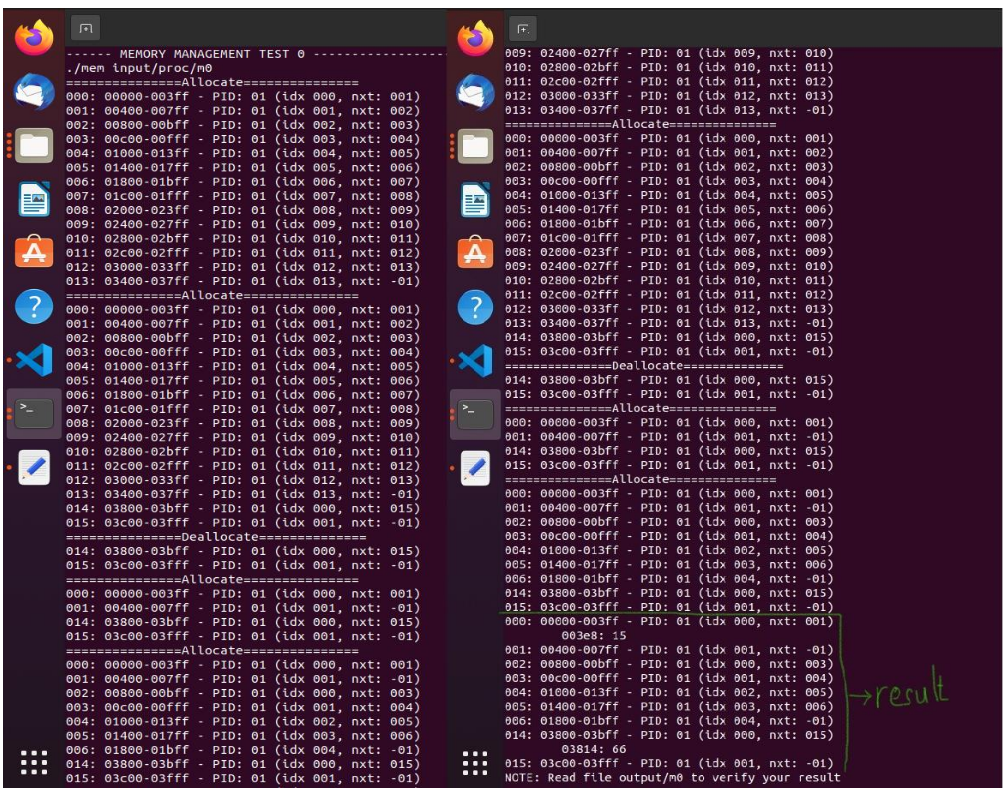
- Save memory, use memory effectively.
- Bring the advantages of pagination algorithm: simple memory allocation, fix external fragmentation.
- Solve the external fragmentation of segmentation algorithm by paging in each segment.

Disadvantages :

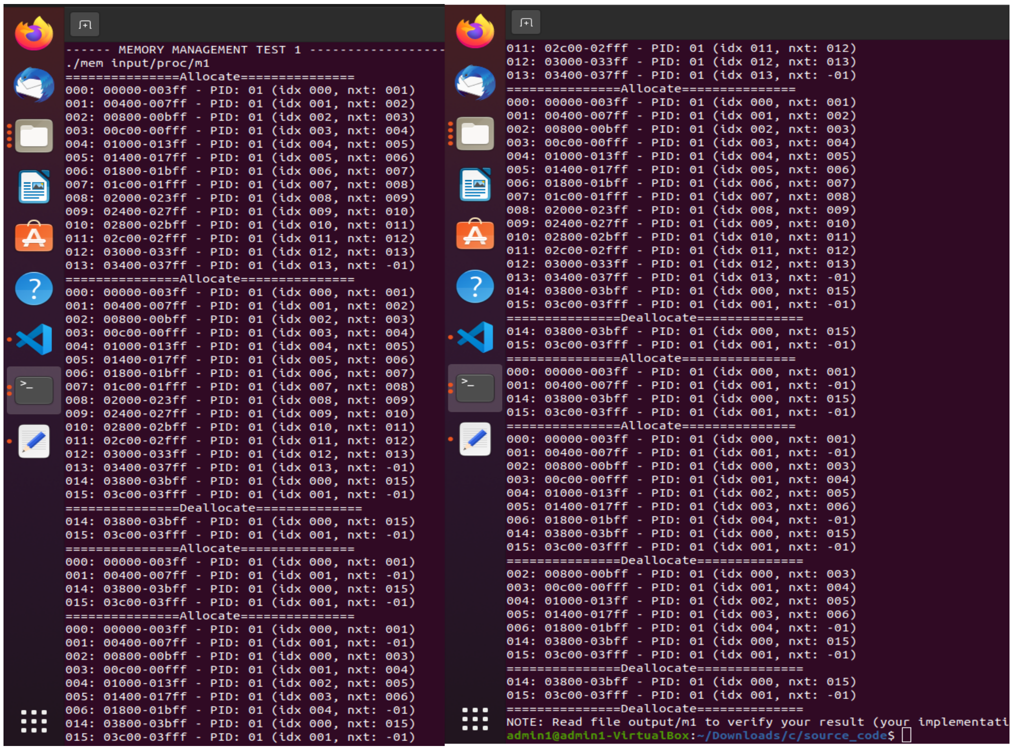
- The internal fragmentation of the paging algorithm remains.

2.2. Result

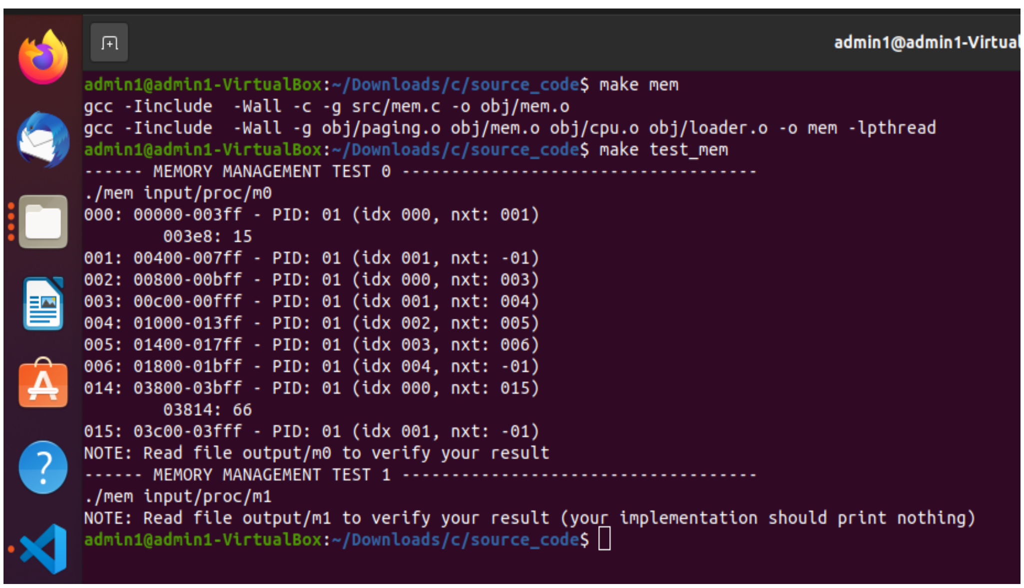
Show the status of RAM after each memory allocation and deallocation function call.
Test 0:



Test 1:



And the result of test 0 and test 1 when run make test_mem :



2.3. Implementation

2.1.1. Find the paging table from the segment

In this assignment, each address is represented by 20 bits, where the first 5 bits are segments, the next 5 bits are page, and the last 10 bits are offset.This function accepts 5 bit segment index and seg_table segment table, find the res paging table of corresponding segment in the above segment table. Since the seg_table table of segments is a structured list of u elements (v_index, page_table_t), where v_index is the 5 bits segment of the element u and page_table_t is the corresponding page section table of that segment. So to find res, we just need to browse on this fractional table, any element u has v_inde equal to the index we need to find, we return the corresponding 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

        /*
         * 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(index==seg_table->table[i].v_index)
                {
                        return seg_table->table[i].pages;
                }
        }
        return NULL;
}
```

2.3.2. Translate virtual address to physical address

Since each address consists of 20 bits with the organization described above, to create a physical address, we take the first 10 bits (segment and page) and connect to the last 10 bits (offset). Each page_table_t stores elements with p_index of 10 that first bit. So to create the physical address, we just need to move those 10 bits left by 10 bits offset then or (|) two this string.

```
/* 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 < 1<<PAGE_LEN; 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. Allocate memory

2.3.3.1. Check memory ready

This step we check if memory is available on the physical memory

logic and logical memory or not. On the physical area, we check the number of empty pages, not used by any process, if there are enough pages to allocate, the physical area is ready. In addition to optimizing the search time when there is not enough memory, we can organize _mem_stat in the form of a list, including size management, free memory, ... to access necessary information quickly. On logical memory, we check based on the process break point, do not exceed the allowed memory.

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

2.3.3.2. Allocate memory

Browse on physical memory, find free pages, assign this page to be used by the process.Create variable last_allocated_page_index to update next value easier. On logical memory, based on the allocated address, from the starting address and the page-allocated position, we find its segments and pages. From there update the paging tables, corresponding segments.

```
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 number_alloc_pages = 0;
		int prev_index;		// index of the previous page in the list
		addr_t curr_virtual_addr;
		int seg_idx,page_idx;
		for(i = 0; i < NUM_PAGES; i++){
			if(_mem_stat[i].proc == 0){
				_mem_stat[i].proc = proc->pid;
				_mem_stat[i].index = number_alloc_pages;

				if(_mem_stat[i].index != 0)
					_mem_stat[prev_index].next = i;
				prev_index = i;

				int found = 0;
				struct seg_table_t * seg_table = proc->seg_table;
				if(seg_table->table[0].pages == NULL)
					seg_table->size = 0;

				curr_virtual_addr =ret_mem + (number_alloc_pages << OFFSET_LEN) ;

				seg_idx=get_first_lv(curr_virtual_addr);
				page_idx=get_second_lv(curr_virtual_addr);
				int j;
				for(j = 0; j < seg_table->size; j++){
					if( seg_table->table[j].v_index == seg_idx ){
						struct page_table_t * curr_page_table = seg_table->table[j].pages;

						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++;

						found = 1;
						break;
					}
				}
```

```
                if(!found){//If not found, add new row into table
                        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++;
                }

                number_alloc_pages++;
                if(number_alloc_pages == num_pages){
                        _mem_stat[i].next = -1;
                        break;
                }
            }
        }
}
/*if(True){
        puts("==============Allocate==============");
        dump();
}*/     //only use if user want to show Deallocate
pthread_mutex_unlock(&mem_lock);
return ret_mem;
```

2.3.4. Free memory

Free the physical address: convert the logical address from the process to physical, then based on the next value of mem, we update the corresponding address string.

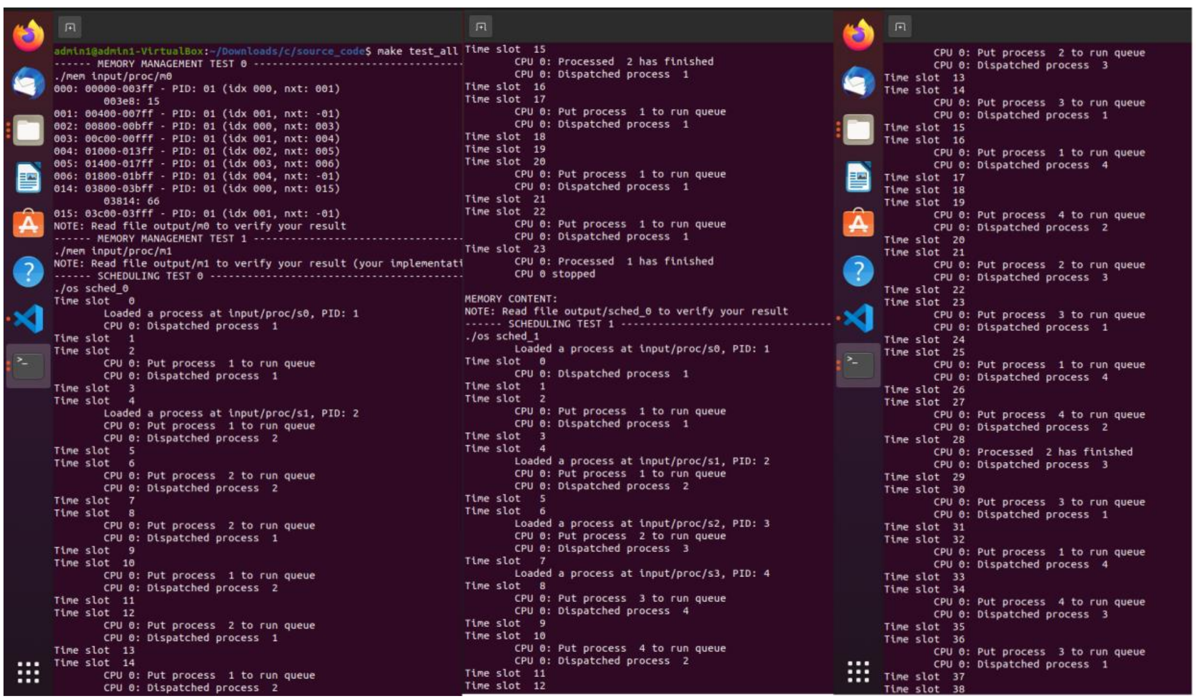
Update logical address: based on the number of pages deleted on the block of the physical address, we look for pages on the logical address in turn, based on the address, we find the corresponding segment and page. Then update again the paging table, after the update, if the table is empty then delete this table in the segment.

```
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;
                                        int number_free_pages = 0;
                                        addr_t curr_virtual_addr = (number_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(curr_virtual_addr);
                                                page_idx=get_second_lv(curr_virtual_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;
                                                                                for(m = l; m < proc->seg_table->table[k].pages->size - 1; m++)//Rearrange page table
                                                                                        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(proc->seg_table->table[k].pages->size == 0){//If page empty
                                                                                        free(proc->seg_table->table[k].pages);
                                                                                        for(m = k; m < proc->seg_table->size - 1; m++)//Rearrange segment table
                                                                                                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;
                                            number_free_pages++;
                                    }
                                    while(p_index != -1);
                                    valid = 1;
                            }
                            break;
                    }
            }
    }
    /*if(True){
            puts("===============Deallocate==============");
            dump();
    }*/  //only use if user want to show Deallocate

    pthread_mutex_unlock(&mem_lock);

    if(!valid)
            return 1;
    else
            return 0;
}
```

3. Put it all together

After combining both scheduling and memory, we did make all and got the results as log files in the directory.



```
        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

MEMORY CONTENT:
NOTE: Read file output/sched_1 to verify your result
----- OS TEST 0 ------------------------------------------------------------
./os os_0
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
Time slot   1
        CPU 1: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/p1, PID: 2
Time slot   3
        CPU 0: 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 1: Put process  1 to run queue
        CPU 1: Dispatched process  3
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  10
Time slot  11
Time slot  12
Time slot  13
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  1
Time slot  14
```

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

MEMORY CONTENT:
000: 00000-003ff - PID: 02 (idx 000, nxt: 001)
001: 00400-007ff - PID: 02 (idx 001, nxt: 007)
002: 00800-00bff - PID: 02 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 02 (idx 001, nxt: 004)
004: 01000-013ff - PID: 02 (idx 002, nxt: 005)
005: 01400-017ff - PID: 02 (idx 003, nxt: -01)
006: 01800-01bff - PID: 03 (idx 000, nxt: 011)
007: 01c00-01fff - PID: 02 (idx 002, nxt: 008)
        01de7: 0a
008: 02000-023ff - PID: 02 (idx 003, nxt: 009)
009: 02400-027ff - PID: 02 (idx 004, nxt: -01)
010: 02800-02bff - PID: 01 (idx 000, nxt: -01)
        02814: 64
011: 02c00-02fff - PID: 03 (idx 001, nxt: 012)
012: 03000-033ff - PID: 03 (idx 002, nxt: 013)
013: 03400-037ff - PID: 03 (idx 003, nxt: -01)
014: 03800-03bff - PID: 04 (idx 000, nxt: 025)
015: 03c00-03fff - PID: 03 (idx 000, nxt: 016)
016: 04000-043ff - PID: 03 (idx 001, nxt: 017)
017: 04400-047ff - PID: 03 (idx 002, nxt: 018)
        045e7: 0a
018: 04800-04bff - PID: 03 (idx 003, nxt: 019)
019: 04c00-04fff - PID: 03 (idx 004, nxt: -01)
020: 05000-053ff - PID: 04 (idx 000, nxt: 021)
021: 05400-057ff - PID: 04 (idx 001, nxt: 022)
022: 05800-05bff - PID: 04 (idx 002, nxt: 023)
        059e7: 0a
023: 05c00-05fff - PID: 04 (idx 003, nxt: 024)
024: 06000-063ff - PID: 04 (idx 004, nxt: -01)
025: 06400-067ff - PID: 04 (idx 001, nxt: 026)
```

```
026: 06800-06bff - PID: 04 (idx 002, nxt: 027)
027: 06c00-06fff - PID: 04 (idx 003, nxt: -01)
NOTE: Read file output/os_0 to verify your result
----- OS TEST 1 ------------------------------------------------------------
./os os_1
Time slot   0
Time slot   1
        Loaded a process at input/proc/p0, PID: 1
        CPU 0: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/s3, PID: 2
Time slot   3
        CPU 2: Dispatched process  2
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/m1, PID: 3
Time slot   5
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 1: Dispatched process  3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   6
        Loaded a process at input/proc/s2, PID: 4
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  4
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  2
        Loaded a process at input/proc/m0, PID: 5
Time slot   7
        CPU 3: Dispatched process  3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  5
Time slot   8
Time slot   9
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  1
        Loaded a process at input/proc/p1, PID: 6
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  3
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  4
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  6
        Loaded a process at input/proc/s0, PID: 7
        CPU 2: Put process  3 to run queue
        CPU 2: Dispatched process  5
```

```
        CPU 2: Dispatched process  5
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  1
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  7
Time slot  12
Time slot  13
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  3
        CPU 1: Processed  1 has finished
        CPU 1: Dispatched process  4
        CPU 2: Put process  5 to run queue
        CPU 2: Dispatched process  2
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  6
Time slot  14
Time slot  15
        CPU 3: Processed  3 has finished
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  7
        CPU 3: Dispatched process  5
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  2
Time slot  16
        Loaded a process at input/proc/s1, PID: 8
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  8
Time slot  17
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  6
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
        CPU 3: Put process  5 to run queue
        CPU 3: Dispatched process  7
Time slot  18
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
Time slot  19
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  5
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  7
Time slot  20
        CPU 3: Processed  5 has finished
        CPU 3: Dispatched process  6
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
Time slot  21
```

```
Time slot  21
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
        CPU 1: Processed  4 has finished
        CPU 1 stopped
Time slot  22
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
Time slot  23
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
        CPU 0: Processed  8 has finished
        CPU 0 stopped
Time slot  24
        CPU 3: Processed  6 has finished
        CPU 3 stopped
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

MEMORY CONTENT:
000: 00000-003ff - PID: 05 (idx 000, nxt: 001)
        003e8: 15
001: 00400-007ff - PID: 05 (idx 001, nxt: -01)
002: 00800-00bff - PID: 06 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 06 (idx 001, nxt: 004)
004: 01000-013ff - PID: 06 (idx 002, nxt: 005)
        011e7: 0a
005: 01400-017ff - PID: 06 (idx 003, nxt: 006)
006: 01800-01bff - PID: 06 (idx 004, nxt: -01)
007: 01c00-01fff - PID: 05 (idx 000, nxt: 008)
008: 02000-023ff - PID: 05 (idx 001, nxt: 009)
009: 02400-027ff - PID: 05 (idx 002, nxt: 010)
010: 02800-02bff - PID: 05 (idx 003, nxt: 011)
011: 02c00-02fff - PID: 05 (idx 004, nxt: -01)
016: 04000-043ff - PID: 06 (idx 000, nxt: 017)
017: 04400-047ff - PID: 06 (idx 001, nxt: 018)
018: 04800-04bff - PID: 06 (idx 002, nxt: 019)
019: 04c00-04fff - 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
```