VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Operating System (CO2017)

Operating Systems Assignment - Simple Operating System

Instructor: Le Thanh Van
Group members: Trinh Minh Trung – 1852825
Tran Cao Duy Truong –2052299
Pham Minh Duc – 1752177

Ho Chi Minh City, April 26, 2022

Contents

1 Introduction

In this assignment, we must carry out the simulation of a simple operating system. The goal of this project is to know how scheduler work, synchronization and the related between physical memory and virtual memory. All the works will be implement with C-language and run test by terminal on LINUX OS.

2 Scheduling

2.1 Priority feedback queue

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

Answer: In Priority Scheduling, each process is given a priority, and higher priority methods are executed first. Priority can be defined in both internal and external. With internal, priorities are assigned by technical quantities such as memory usage, and file I/O operations. In external, priorities are assigned by politics, commerce, or user preference, such as importance and amount being paid for process access (the latter usually being for mainframes).

The PFQ algorithms uses 2 queues ready_queue and run_queue with the meanings:
+ Ready_queue: it contains processes that are arranged in order of their priorities. When the CPU moves to the next slot, the process with highest priority will jump out to be executed.
+ Run_queue: This queue contains processes that are waiting to continue executing after its slot has not completed its process. Processes in this queue can only continue to the next slot when ready_queue is free and the remaining works of unfinished processes will be move to the back of Ready_queue.
The advantages of PFQ:

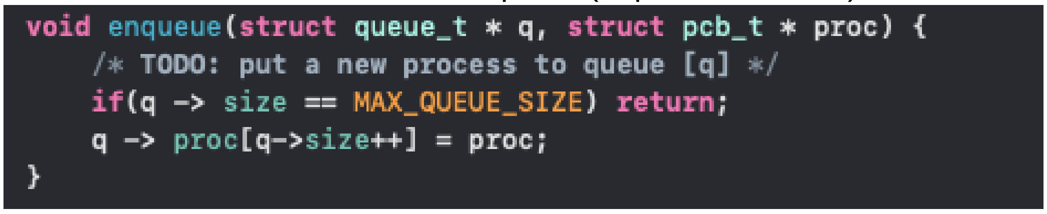
Compared to SJF, SRTF, FCFS or Round Robin with FCFS, it’s more flexible, this algorithm can allow important works done without long waiting time.

Priority feedback queue supports preemption, which is switching the process. This also prevents the indefinite blocking (starvation). The different between Priority scheduling and PFQ, if high-priority processes keep coming to the queue frequently, the low-priority processes have no chance to run. With Ready_queue and Run_queue, PFQ solve this problem.

2.2 Implementation

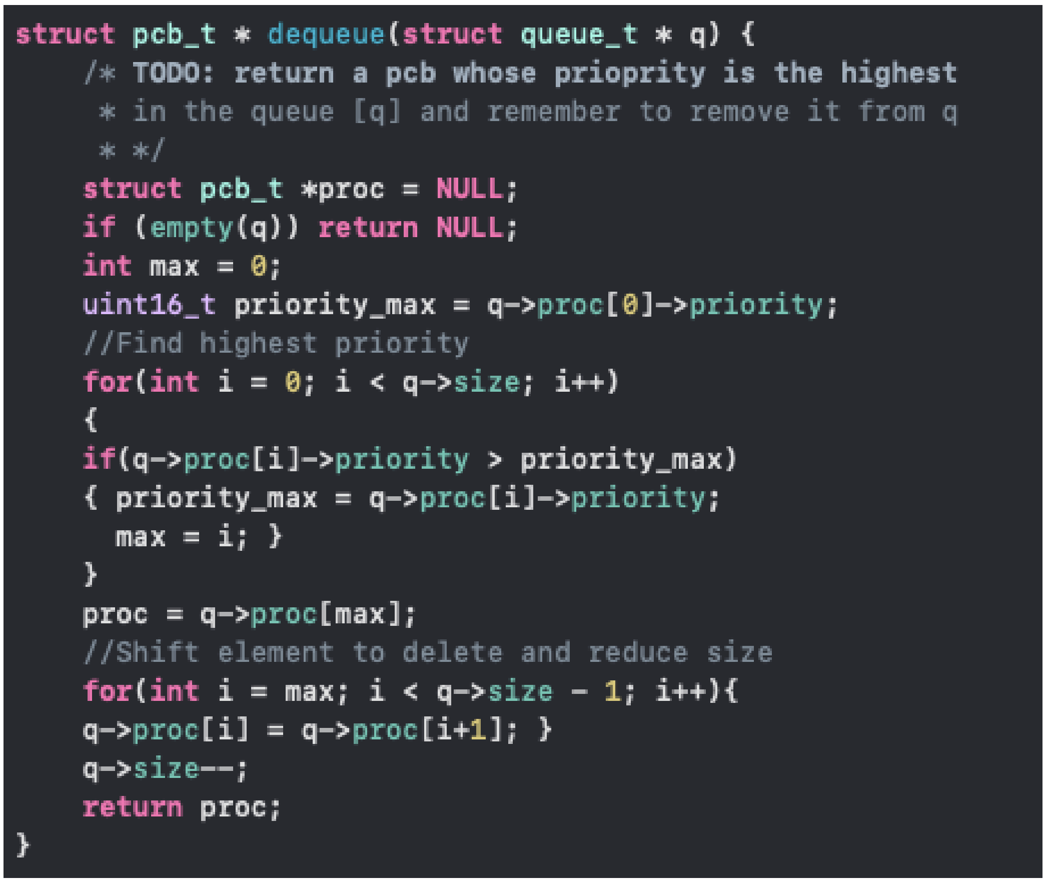
2.2.1 Queue

Enqueue add element to the back of the queue (if queue is not full)



```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;
}
```

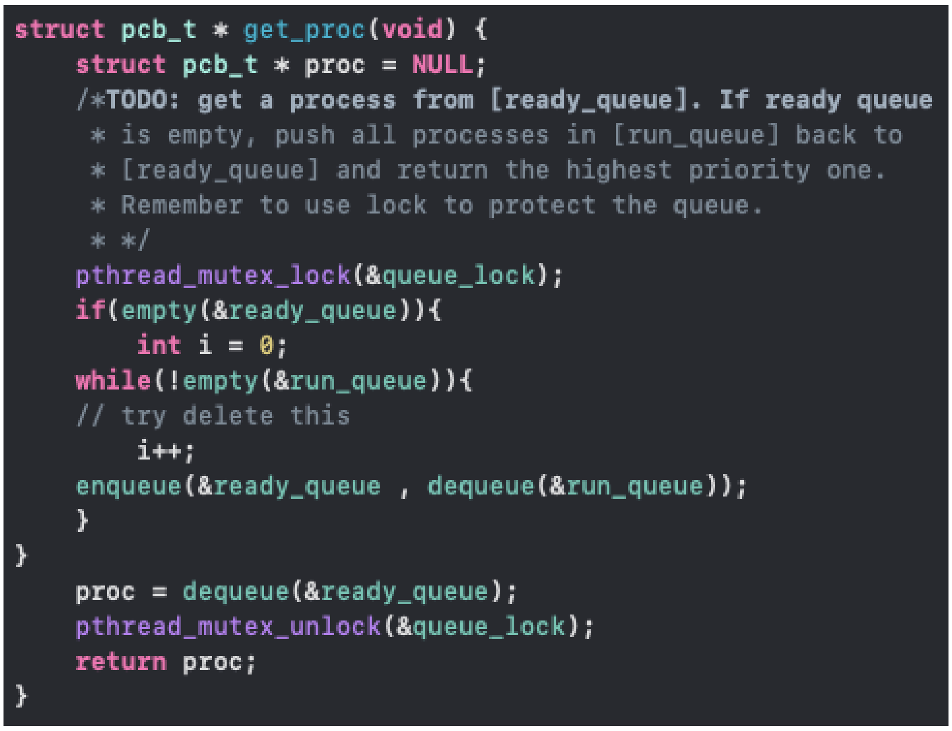
Dequeue find the process which has highest priority, store the information of that process and finally, update the queue after removing the process.



```c
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
     * */
    struct pcb_t *proc = NULL;
    if (empty(q)) return NULL;
    int max = 0;
    uint16_t priority_max = q->proc[0]->priority;
    //Find highest priority
    for(int i = 0; i < q->size; i++)
    {
    if(q->proc[i]->priority > priority_max)
    { priority_max = q->proc[i]->priority;
      max = i; }
    }
    proc = q->proc[max];
    //Shift element to delete and reduce size
    for(int i = max; i < q->size - 1; i++){
    q->proc[i] = q->proc[i+1]; }
    q->size--;
    return proc;
}
```

2.2.2 Make schedule

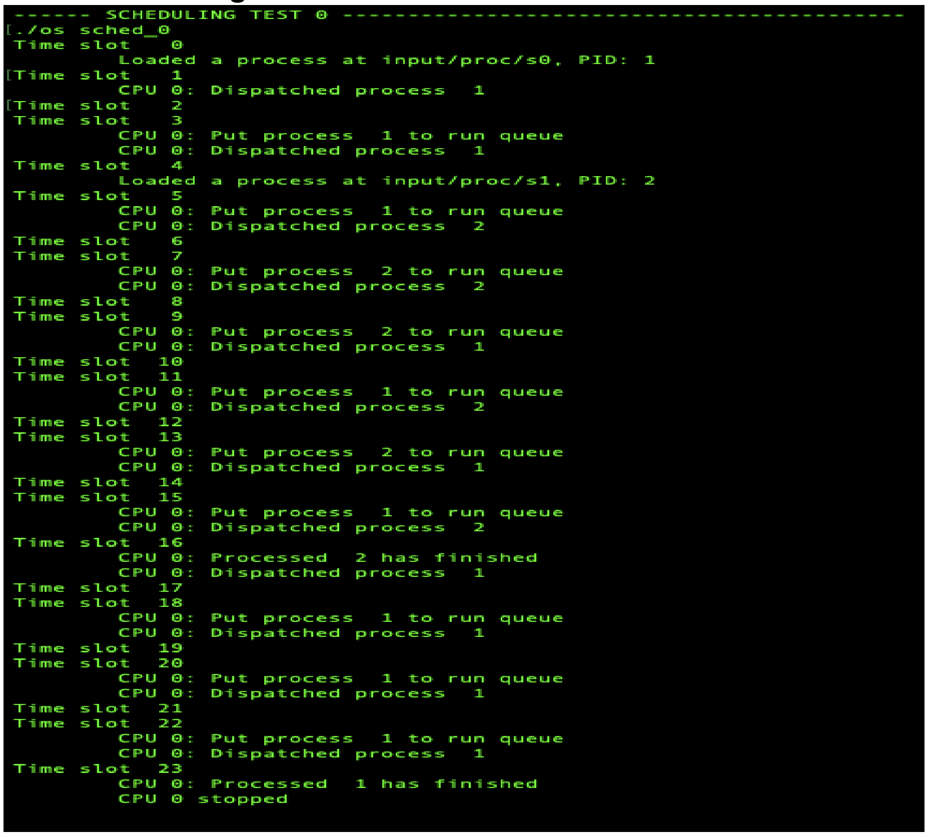
We implement get_proc() in scheh.c to get the PCB of a process in ready_queue. If Ready_queue is empty, feedback all PCB in Run_queue and return the highest priority one



```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)){
        int i = 0;
    while(!empty(&run_queue)){
    // try delete this
        i++;
    enqueue(&ready_queue , dequeue(&run_queue));
    }
}
    proc = dequeue(&ready_queue);
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

2.3 Results

Result running from SCHEDULING TEST 0.



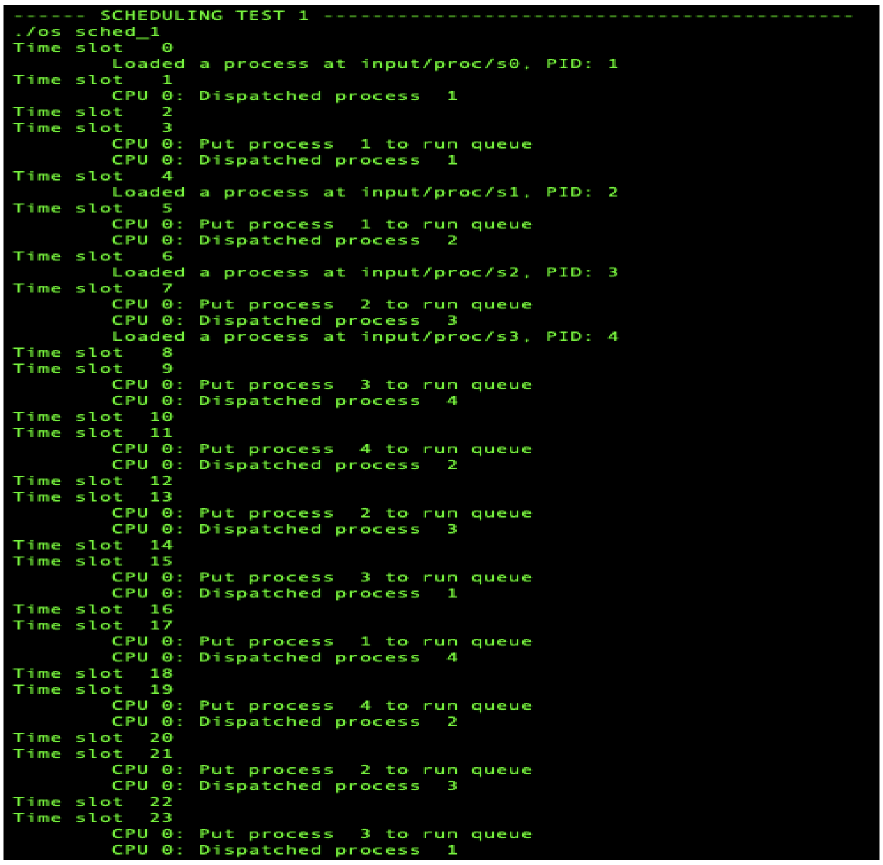
```
------ SCHEDULING TEST 0 -------------------------------------------
[./os sched_0
 Time slot   0
        Loaded a process at input/proc/s0, PID: 1
[Time slot   1
        CPU 0: Dispatched process  1
[Time slot   2
 Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched 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
 Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
 Time slot   8
 Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
 Time slot  10
 Time slot  11
        CPU 0: Put process  1 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  1
 Time slot  14
 Time slot  15
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
 Time slot  16
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
 Time slot  17
 Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
 Time slot  19
 Time slot  20
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
 Time slot  21
 Time slot  22
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
 Time slot  23
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Process		P1				P2				P1		P2		P1		P2	P1							
Timeslot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

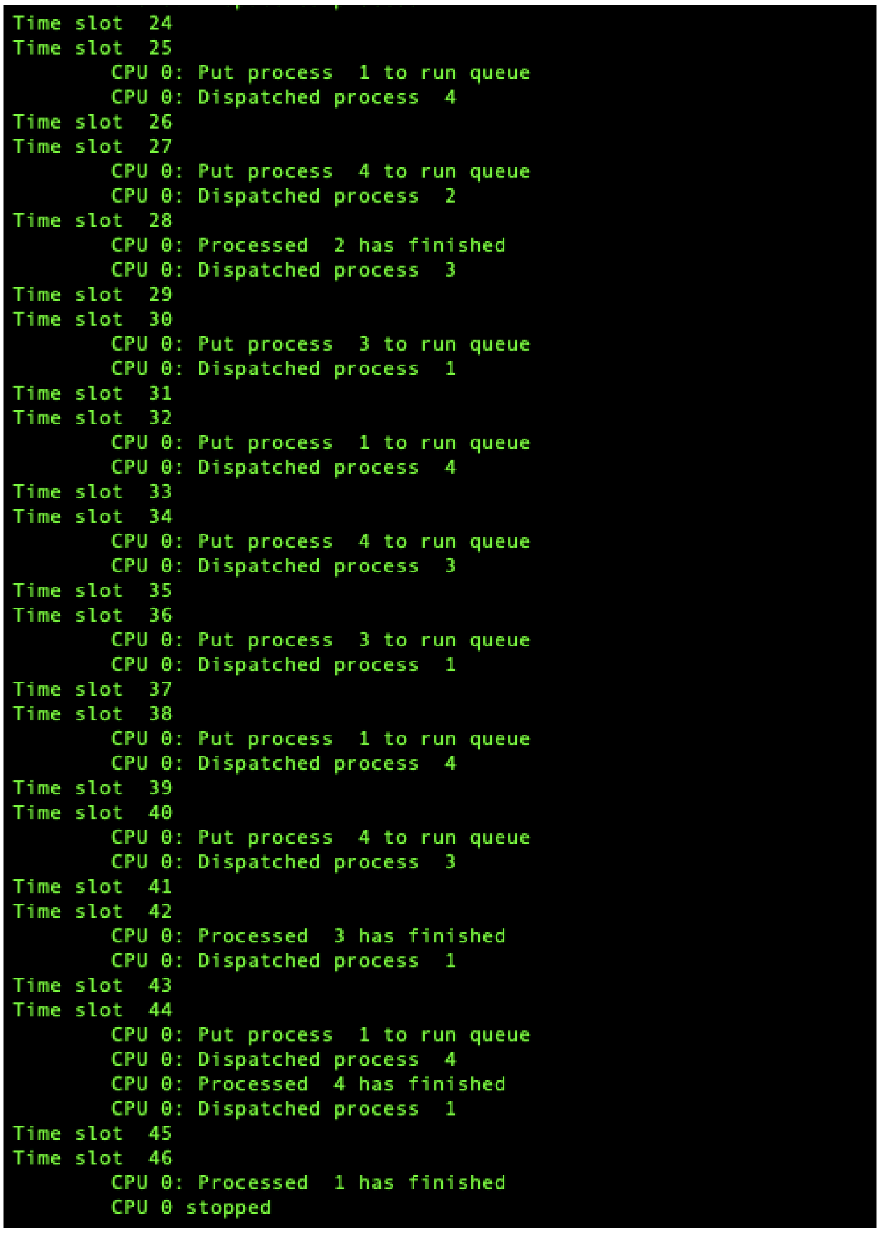
Average waiting time = 1

Average turnaround time = 17

Result running from SCHEDULING TEST 1



```
------ SCHEDULING TEST 1 -------------------------------------------
./os 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
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  45
Time slot  46
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Process		P1				P2		P3		P4		P2		P3		P1		P4		P2		P3	
Timeslot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Process	P1		P4		P2	P3		P1		P4		P3		P1		P4		P3		P1		P4	P1	
Timeslot	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46

Average waiting time: 1.25

Average turnaround time: 35.25

3 Memory

3.1 Applications, Advantages and Disadvantages

Question: In which system is segmentation with paging used (give an example of at least one system)? Explain clearly the advantage and disadvantage of segmentation with paging.

3.1.1 System

IA-32 systems (short for Intel Architecture, 32-bit) use segmentation with paging. Its segment is 4GB large.

3.1.2 Advantages

Reduce memory usage compared to the paging technique. The page table size is reduced as pages are present for data segments instead of storing all pages in RAM.

Solve the external fragmentation problem because it uses a fix-sized page for the memory block. Every hole between allocated memory can be assigned to processes. It can be discontiguous, but it has a table for managing the pages.

Because the memories in RAM does not need to be contiguous, so allocating memory into the frame is become less complicated.

3.1.3 Disadvantages

Internal fragmentation remains a problem. When the memory required is not divisible by the page size lead to this problem. For example, the program need to allocate 23 bytes, the size of page is 5 bytes, there will be 5 frames allocated in the RAM and there will exist a frame will be empty some bytes of it.

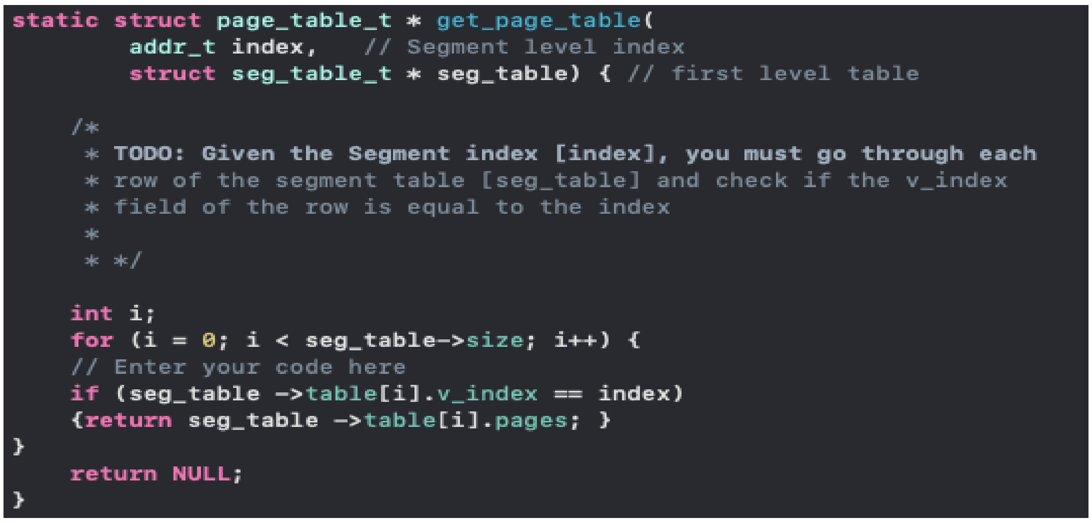
The complexity can be higher because it requires the implementation of a page table and segment table.

It has to access the segment table and the page table to get exact memory. This can make longer time access.

3.2 Implementation

3.2.1 Get page table from segment

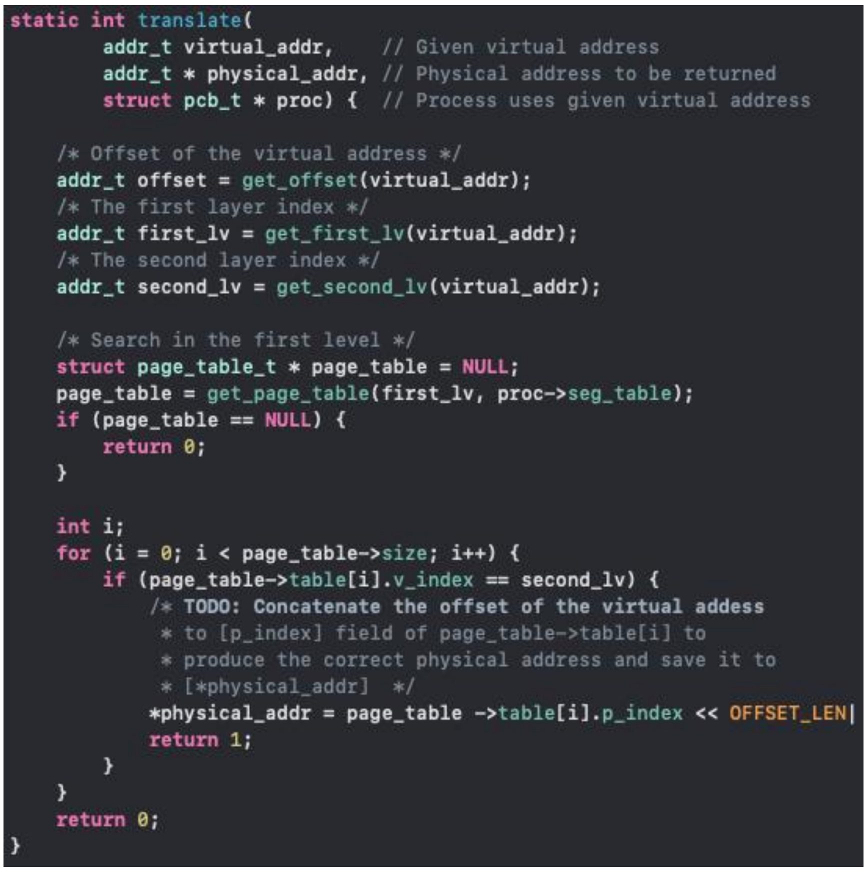
Get exactly paging table with a segment index.



```
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;
}
```

3.2.2 **Translate**

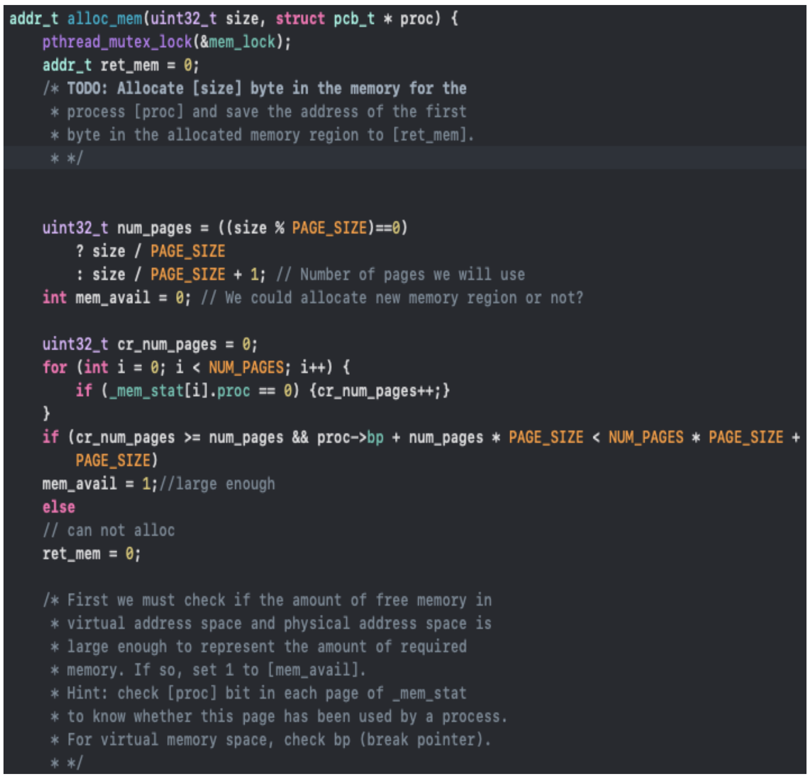
Convert from virtual address to physical address.



```
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 << OFFSET_LEN|
            return 1;
        }
    }
    return 0;
}
```

3.2.3 **Allocate**

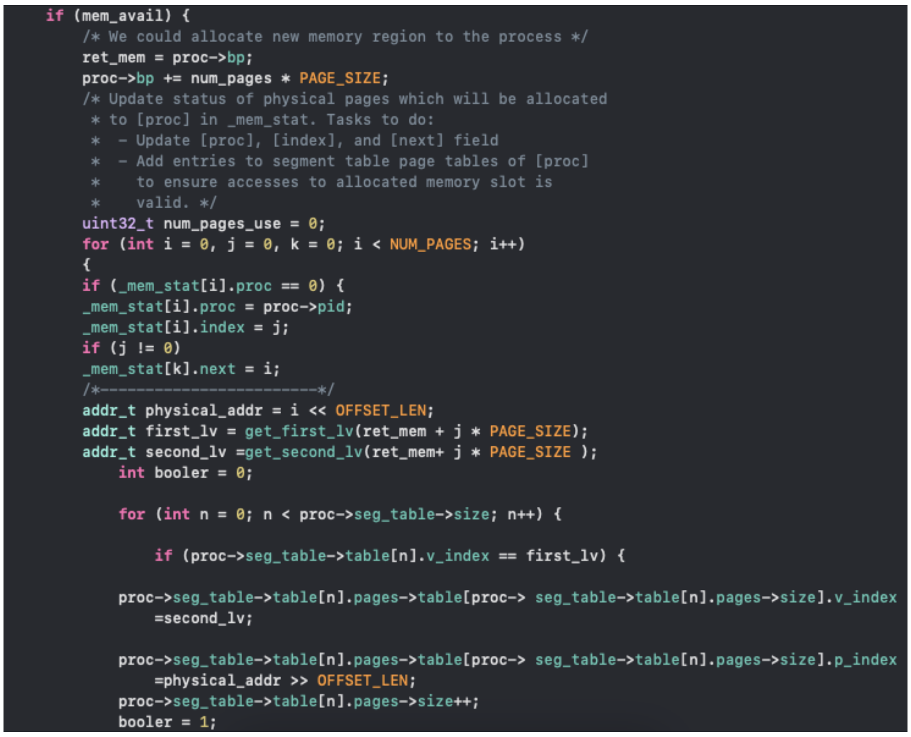
Check if memory is available to allocate:



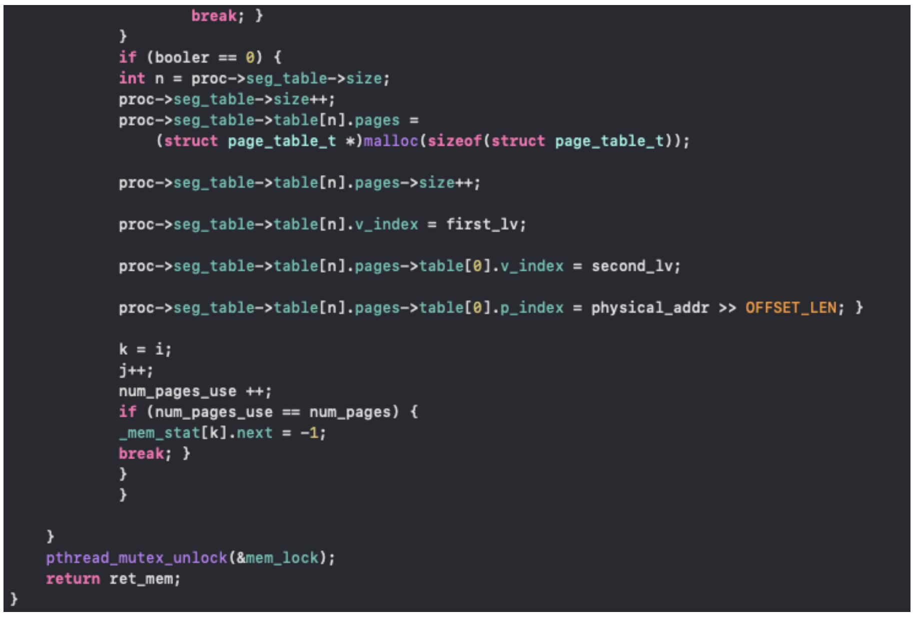
```c
addr_t alloc_mem(uint32_t size, struct pcb_t * proc) {
    pthread_mutex_lock(&mem_lock);
    addr_t ret_mem = 0;
    /* TODO: Allocate [size] byte in the memory for the
     * process [proc] and save the address of the first
     * byte in the allocated memory region to [ret_mem].
     * */


    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?

    uint32_t cr_num_pages = 0;
    for (int i = 0; i < NUM_PAGES; i++) {
        if (_mem_stat[i].proc == 0) {cr_num_pages++;}
    }
    if (cr_num_pages >= num_pages && proc->bp + num_pages * PAGE_SIZE < NUM_PAGES * PAGE_SIZE +
        PAGE_SIZE)
    mem_avail = 1;//large enough
    else
    // can not alloc
    ret_mem = 0;

    /* 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).
     * */
```

Allocate:

+ Iterate through physical memory, find free pages, assign that process has been used on these pages
+ Update [id], [index], and [next] field
+ Add entries if not exist to make it valid

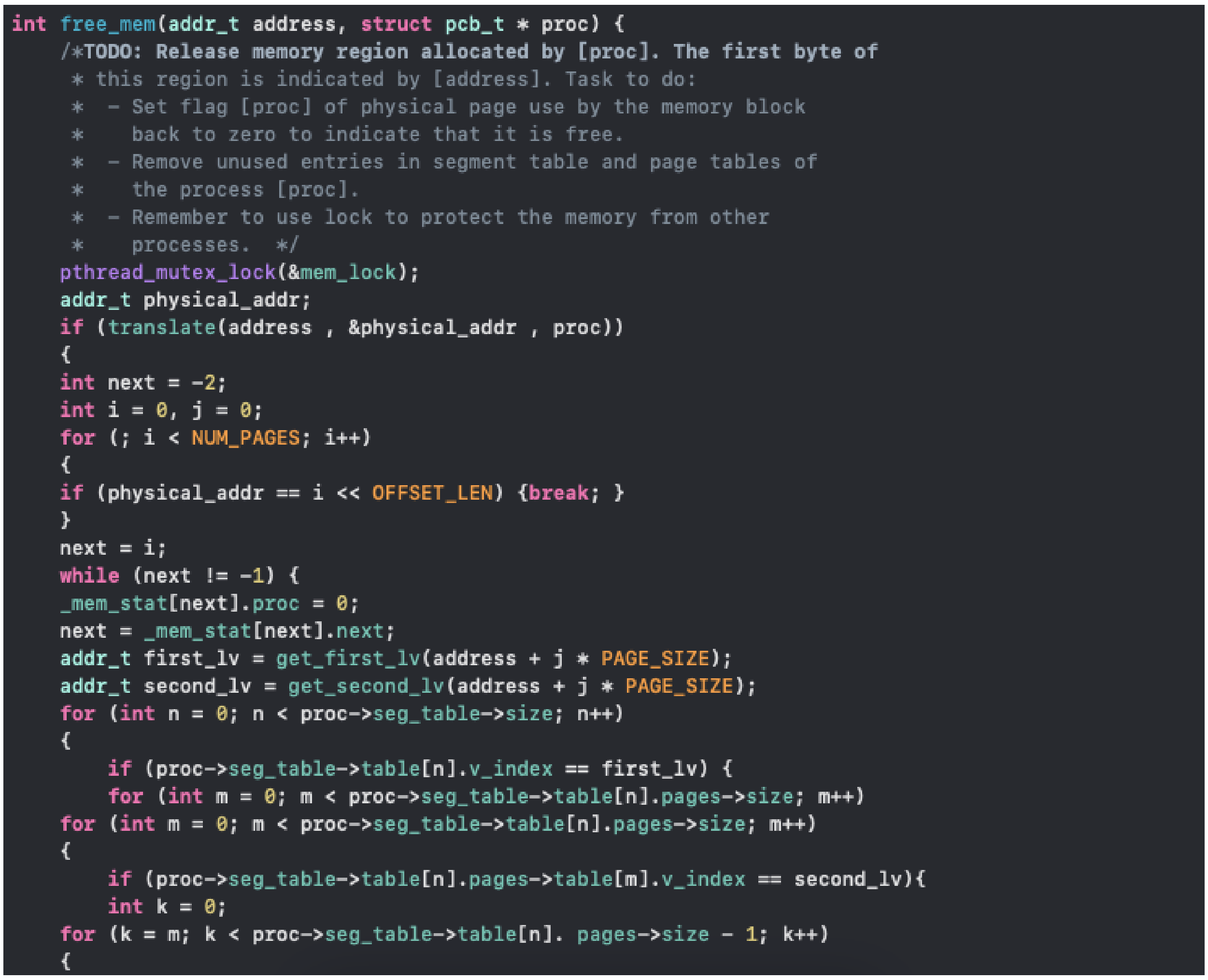


```c
    if (mem_avail) {
        /* We could allocate new memory region to the process */
        ret_mem = proc->bp;
        proc->bp += num_pages * PAGE_SIZE;
        /* Update status of physical pages which will be allocated
         * to [proc] in _mem_stat. Tasks to do:
         *  - Update [proc], [index], and [next] field
         *  - Add entries to segment table page tables of [proc]
         *    to ensure accesses to allocated memory slot is
         *    valid. */
        uint32_t num_pages_use = 0;
        for (int i = 0, j = 0, k = 0; i < NUM_PAGES; i++)
        {
        if (_mem_stat[i].proc == 0) {
        _mem_stat[i].proc = proc->pid;
        _mem_stat[i].index = j;
        if (j != 0)
        _mem_stat[k].next = i;
        /*-----------------------*/
        addr_t physical_addr = i << OFFSET_LEN;
        addr_t first_lv = get_first_lv(ret_mem + j * PAGE_SIZE);
        addr_t second_lv =get_second_lv(ret_mem+ j * PAGE_SIZE );
            int booler = 0;

            for (int n = 0; n < proc->seg_table->size; n++) {

                if (proc->seg_table->table[n].v_index == first_lv) {

            proc->seg_table->table[n].pages->table[proc-> seg_table->table[n].pages->size].v_index
                =second_lv;

            proc->seg_table->table[n].pages->table[proc-> seg_table->table[n].pages->size].p_index
                =physical_addr >> OFFSET_LEN;
            proc->seg_table->table[n].pages->size++;
            booler = 1;
```



```
                    break; }
            }
            if (booler == 0) {
            int n = proc->seg_table->size;
            proc->seg_table->size++;
            proc->seg_table->table[n].pages =
                (struct page_table_t *)malloc(sizeof(struct page_table_t));

            proc->seg_table->table[n].pages->size++;

            proc->seg_table->table[n].v_index = first_lv;

            proc->seg_table->table[n].pages->table[0].v_index = second_lv;

            proc->seg_table->table[n].pages->table[0].p_index = physical_addr >> OFFSET_LEN; }

            k = i;
            j++;
            num_pages_use ++;
            if (num_pages_use == num_pages) {
            _mem_stat[k].next = -1;
            break; }
            }
            }

    }
    pthread_mutex_unlock(&mem_lock);
    return ret_mem;
}
```

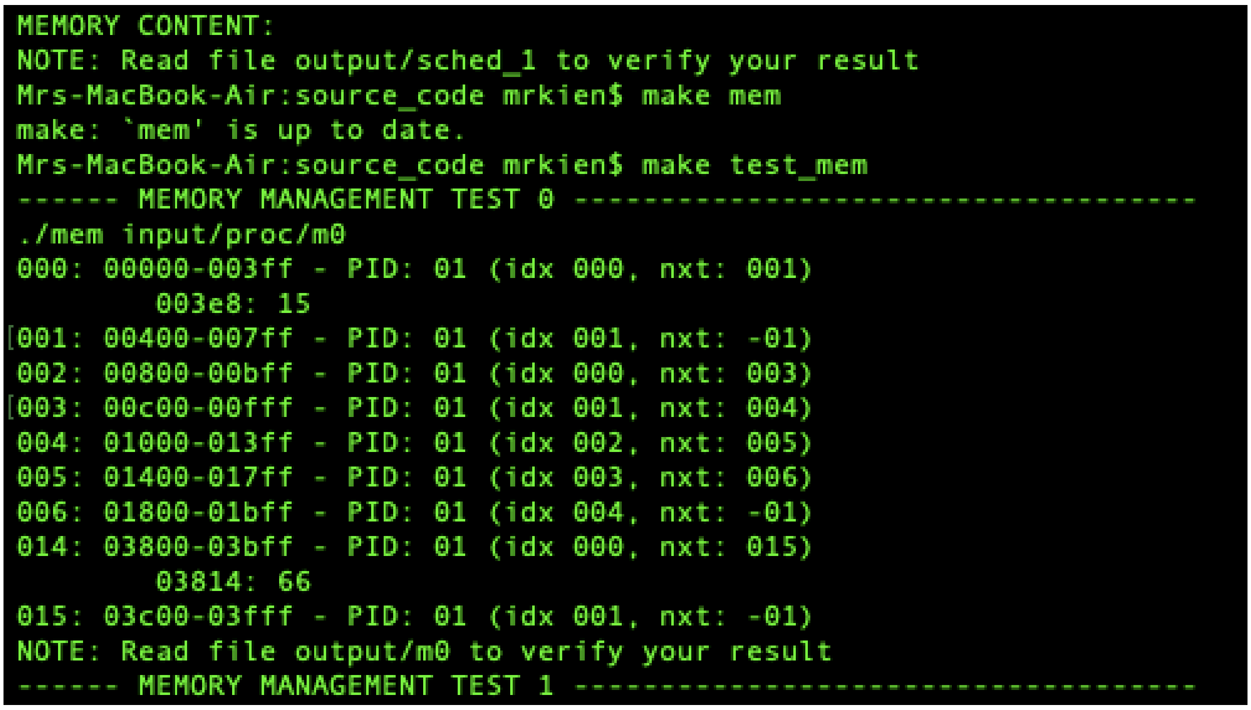
3.2.3 **Deallocate**



```
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);
    addr_t physical_addr;
    if (translate(address , &physical_addr , proc))
    {
    int next = -2;
    int i = 0, j = 0;
    for (; i < NUM_PAGES; i++)
    {
    if (physical_addr == i << OFFSET_LEN) {break; }
    }
    next = i;
    while (next != -1) {
    _mem_stat[next].proc = 0;
    next = _mem_stat[next].next;
    addr_t first_lv = get_first_lv(address + j * PAGE_SIZE);
    addr_t second_lv = get_second_lv(address + j * PAGE_SIZE);
    for (int n = 0; n < proc->seg_table->size; n++)
    {
        if (proc->seg_table->table[n].v_index == first_lv) {
        for (int m = 0; m < proc->seg_table->table[n].pages->size; m++)
    for (int m = 0; m < proc->seg_table->table[n].pages->size; m++)
    {
        if (proc->seg_table->table[n].pages->table[m].v_index == second_lv){
        int k = 0;
    for (k = m; k < proc->seg_table->table[n]. pages->size - 1; k++)
    {
```

```
    proc->seg_table->table[n].pages->table[k].v_index =
        proc->seg_table->table[n].pages->table[k+1].v_index;

        proc->seg_table->table[n].pages->table[k].p_index =
            proc->seg_table->table[n].pages->table[k+1].p_index;}

    proc->seg_table->table[n].pages->table[k].v_index = 0;

    proc->seg_table->table[n].pages->table[k].p_index = 0;

    proc->seg_table->table[n].pages->size--;
    /*----------------*/
    break;
            }
    }
    if (proc->seg_table->table[n].pages->size == 0)
    {
    free(proc->seg_table->table[n].pages);

    int m = 0;

    for (m = n; m < proc->seg_table->size - 1; m++) {
    proc->seg_table->table[m].v_index = proc-> seg_table->table[m + 1].v_index;
    proc->seg_table->table[m].pages = proc-> seg_table ->table[m + 1].pages;
    }
        proc->seg_table->table[m].v_index = 0;
        proc->seg_table->table[m].pages = NULL;
        proc->seg_table->size--;
        }
    break;

         }
     }
    j++;
     }
}
    pthread_mutex_unlock(&mem_lock);
    return 0;
}
```

3.3 Result



```
MEMORY CONTENT:
NOTE: Read file output/sched_1 to verify your result
Mrs-MacBook-Air:source_code mrkien$ make mem
make: `mem' is up to date.
Mrs-MacBook-Air:source_code mrkien$ make test_mem
------ MEMORY MANAGEMENT TEST 0 ------------------------------------
./mem input/proc/m0
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
------ MEMORY MANAGEMENT TEST 1 ------------------------------------
```

4 Put it all together

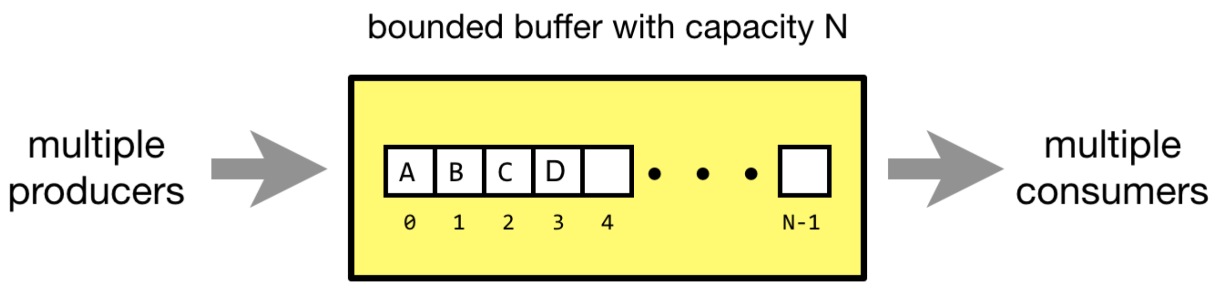
Question: What will happen if the synchronization is not handled in your system?

Illustrate the problem by example if you have any.

First common problem that we can encounter is race condition. For example, if the queue remaining 1 slot to enqueue. The program needs to enqueue 2 slots at the same time, the system will occur a race condition, like this.

- Both threads check the queue at the same time when the queue just have 1 slot remaining. Therefore, it returns a result that not enough slot to enqueue.
- The program failed to add 2 threads into the queue.

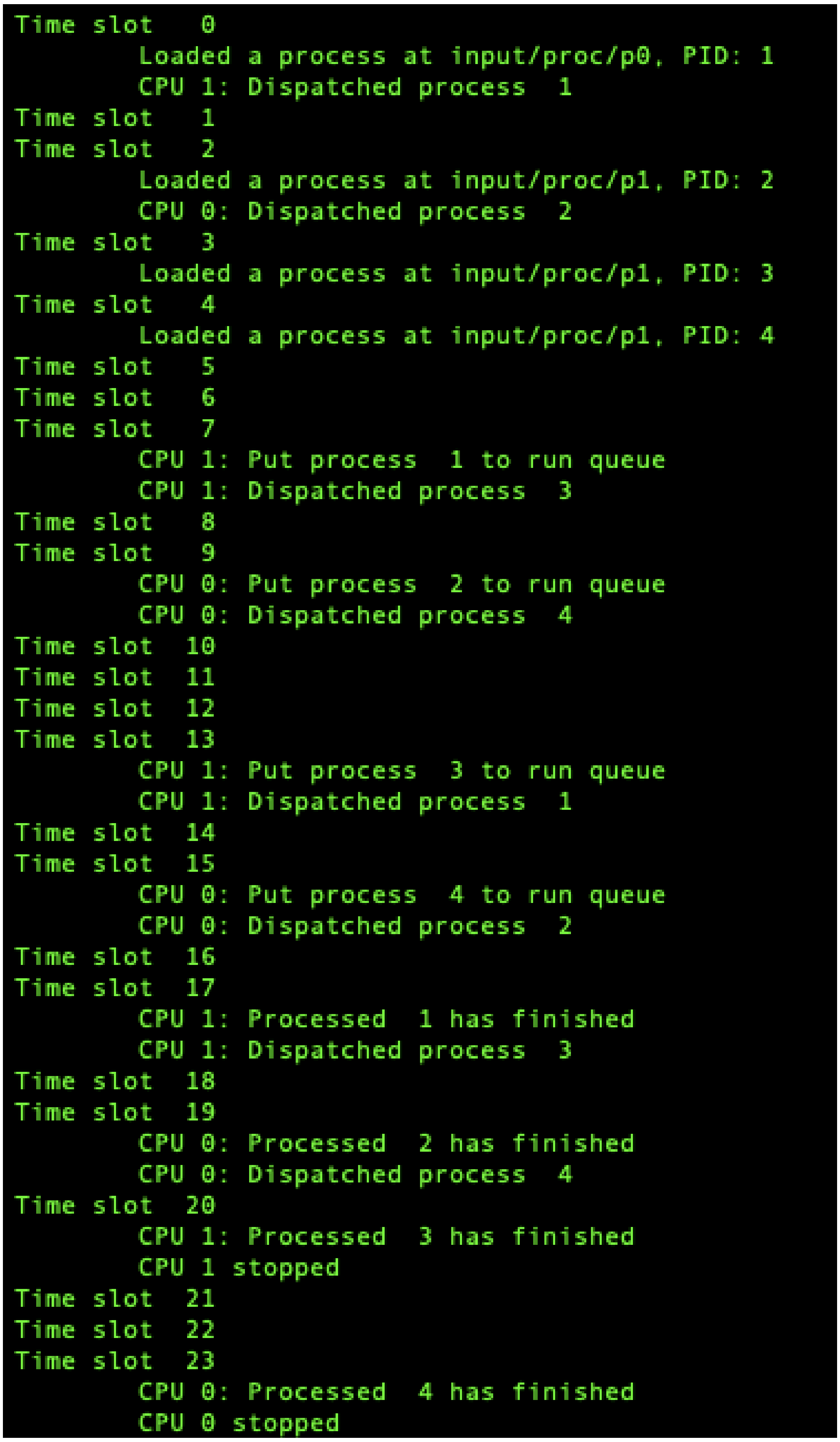
Second problem is a classic problem called bounded buffer problem also known as "The producer-consumer problem". Imagine that the allocate work as the producer and the deallocate work as the consumer. Then the problem occurs when both processes execute concurrently will lead to unexpected output.



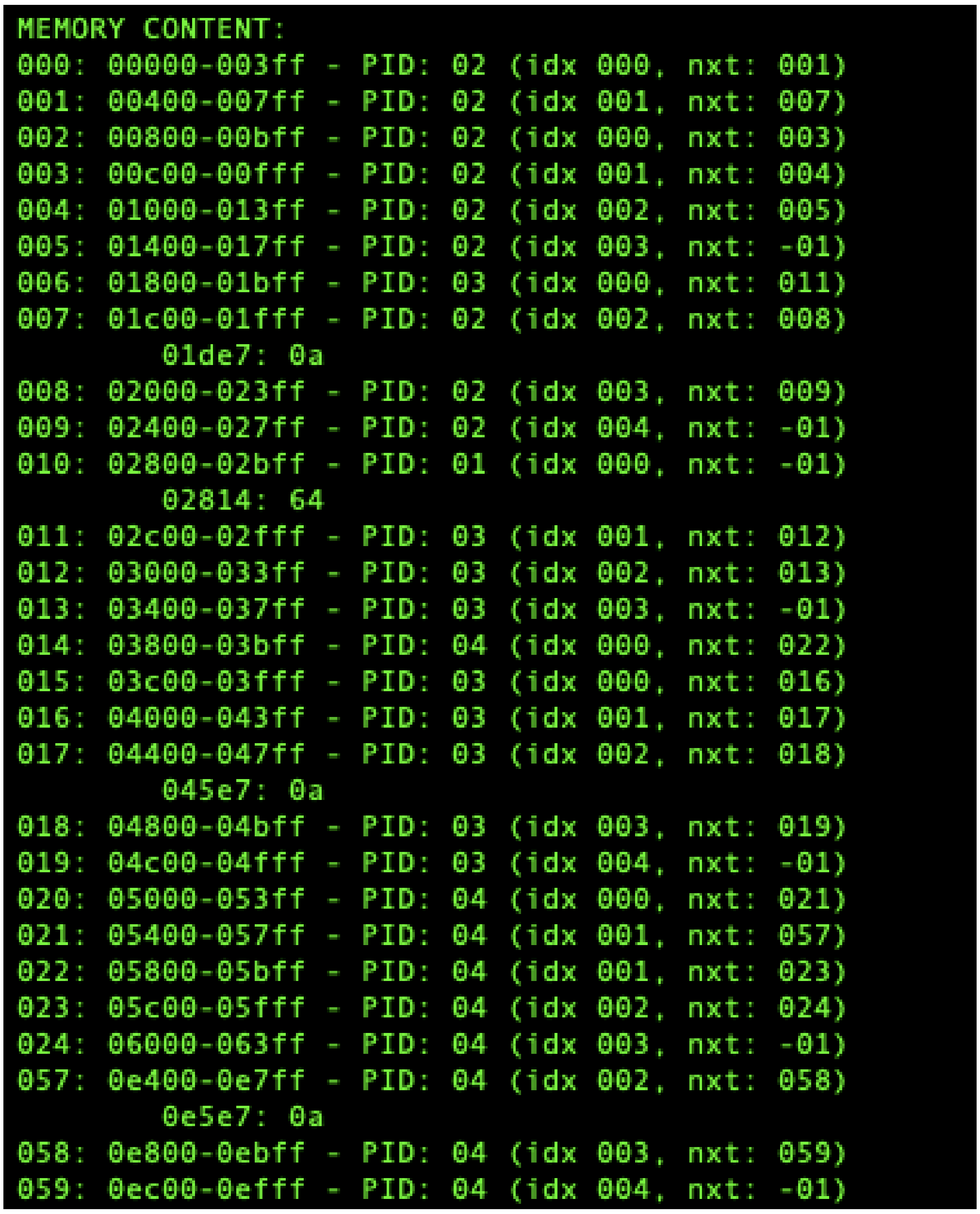
We used mutex lock for this assignment to protect critical section when we get process from ready_queue or push from run_queue to ready_queue and manage the memory.

In overview, we lock queue_lock when get process and lock mem_lock when allocate and de-allocate the memory.

And finally this is our result after running os_0

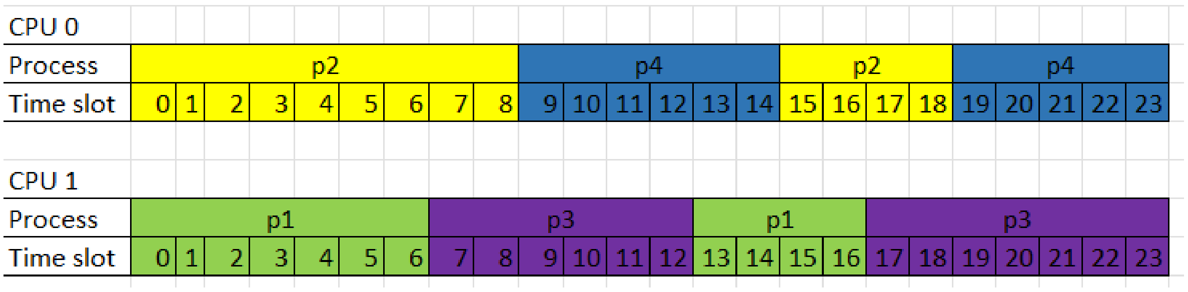


```
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
        CPU 1: Dispatched process  1
Time slot   1
Time slot   2
        Loaded a process at input/proc/p1, PID: 2
        CPU 0: Dispatched process  2
Time slot   3
        Loaded a process at input/proc/p1, PID: 3
Time slot   4
        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  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
        CPU 1: Processed  3 has finished
        CPU 1 stopped
Time slot  21
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: 022)
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: 057)
022: 05800-05bff - PID: 04 (idx 001, nxt: 023)
023: 05c00-05fff - PID: 04 (idx 002, nxt: 024)
024: 06000-063ff - PID: 04 (idx 003, nxt: -01)
057: 0e400-0e7ff - PID: 04 (idx 002, nxt: 058)
        0e5e7: 0a
058: 0e800-0ebff - PID: 04 (idx 003, nxt: 059)
059: 0ec00-0efff - PID: 04 (idx 004, nxt: -01)
```

In Gantt chart for the os_0 we divided the chart into CPU 0 and CPU 1 to see clearly how these CPU work.

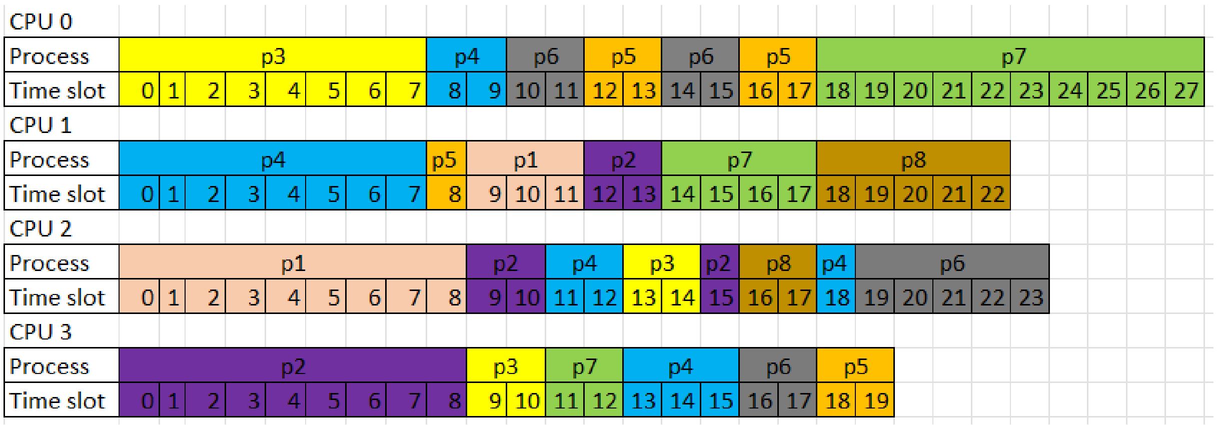


CPU 0																								
Process	p2									p4						p2				p4				
Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

CPU 1																								
Process	p1							p3						p1				p3						
Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

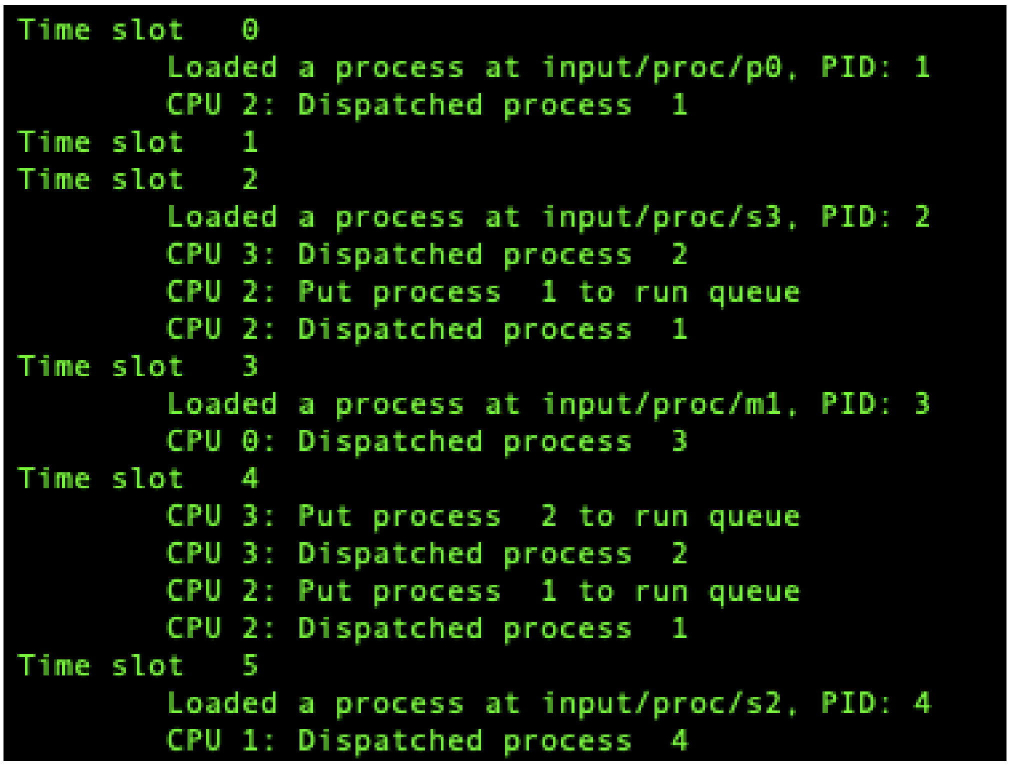
And os_1.

As we do so with os_1 with the quantities of CPU up to 4

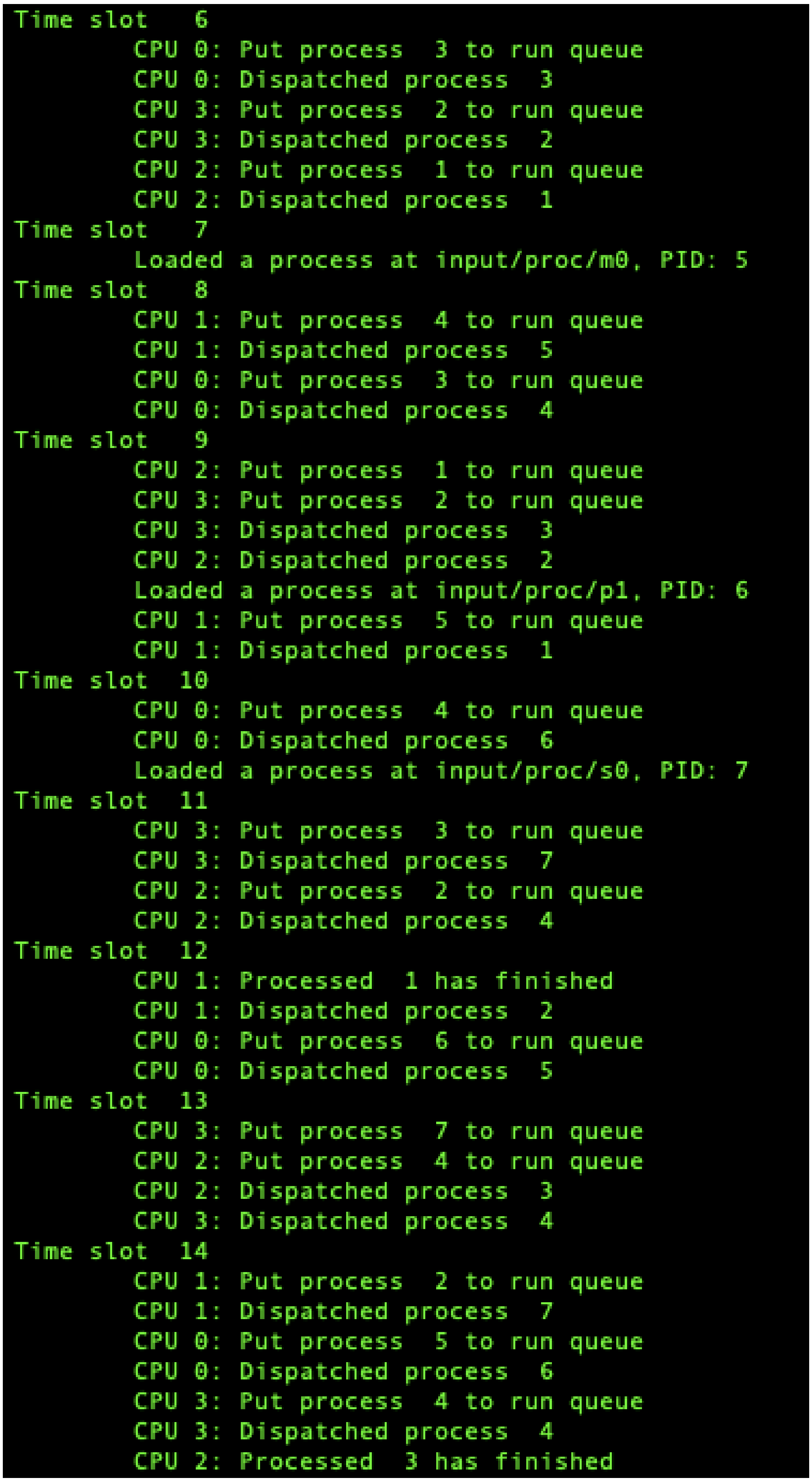


CPU 0																												
Process	p3								p4		p6		p5		p6		p5		p7									
Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
CPU 1																												
Process	p4								p5	p1			p2		p7				p8									
Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22					
CPU 2																												
Process	p1									p2		p4		p3		p2	p8		p4	p6								
Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23				
CPU 3																												
Process	p2									p3		p7		p4			p6		p5									
Time slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19								

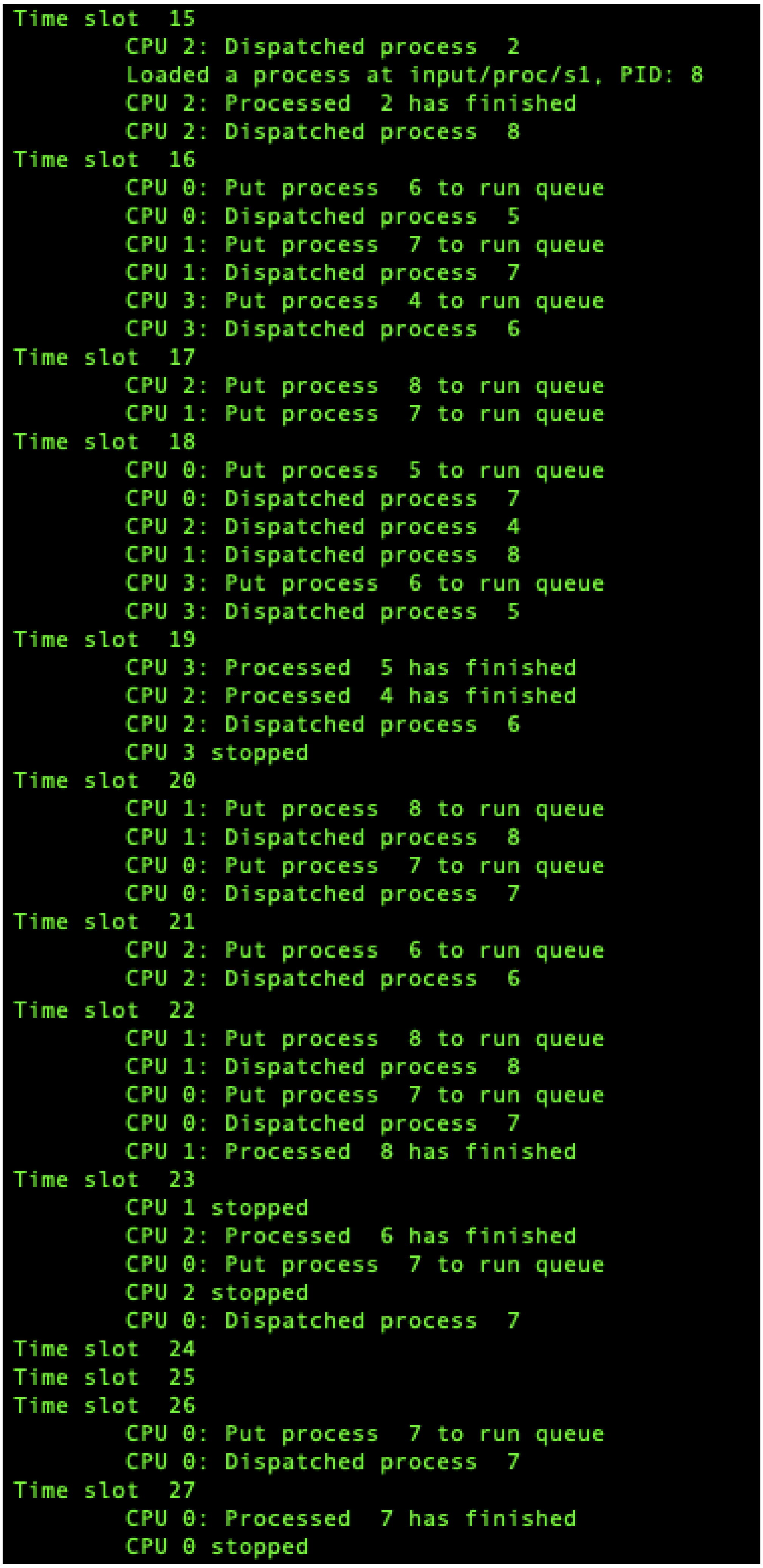
And this is the result of running os_1.



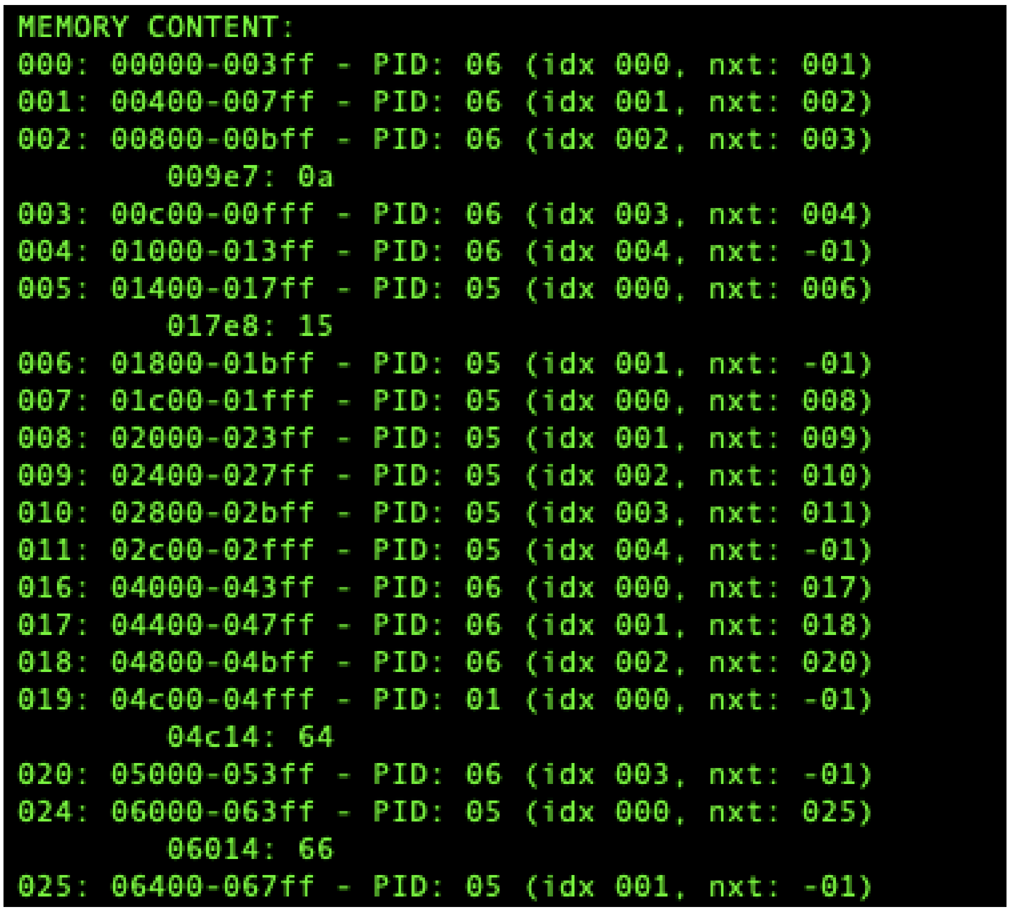
```
Time slot   0
        Loaded a process at input/proc/p0, PID: 1
        CPU 2: Dispatched process  1
Time slot   1
Time slot   2
        Loaded a process at input/proc/s3, PID: 2
        CPU 3: Dispatched process  2
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
Time slot   3
        Loaded a process at input/proc/m1, PID: 3
        CPU 0: Dispatched process  3
Time slot   4
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
Time slot   5
        Loaded a process at input/proc/s2, PID: 4
        CPU 1: Dispatched process  4
```



```
Time slot   6
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
Time slot   7
        Loaded a process at input/proc/m0, PID: 5
Time slot   8
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  5
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot   9
        CPU 2: Put process  1 to run queue
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  3
        CPU 2: Dispatched process  2
        Loaded a process at input/proc/p1, PID: 6
        CPU 1: Put process  5 to run queue
        CPU 1: Dispatched process  1
Time slot  10
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  6
        Loaded a process at input/proc/s0, PID: 7
Time slot  11
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  7
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  4
Time slot  12
        CPU 1: Processed  1 has finished
        CPU 1: Dispatched process  2
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  5
Time slot  13
        CPU 3: Put process  7 to run queue
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  3
        CPU 3: Dispatched process  4
Time slot  14
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  7
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  6
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  4
        CPU 2: Processed  3 has finished
```



```
Time slot  15
        CPU 2: Dispatched process  2
        Loaded a process at input/proc/s1, PID: 8
        CPU 2: Processed  2 has finished
        CPU 2: Dispatched process  8
Time slot  16
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  5
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  6
Time slot  17
        CPU 2: Put process  8 to run queue
        CPU 1: Put process  7 to run queue
Time slot  18
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  7
        CPU 2: Dispatched process  4
        CPU 1: Dispatched process  8
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  5
Time slot  19
        CPU 3: Processed  5 has finished
        CPU 2: Processed  4 has finished
        CPU 2: Dispatched process  6
        CPU 3 stopped
Time slot  20
        CPU 1: Put process  8 to run queue
        CPU 1: Dispatched process  8
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
Time slot  21
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  6
Time slot  22
        CPU 1: Put process  8 to run queue
        CPU 1: Dispatched process  8
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
        CPU 1: Processed  8 has finished
Time slot  23
        CPU 1 stopped
        CPU 2: Processed  6 has finished
        CPU 0: Put process  7 to run queue
        CPU 2 stopped
        CPU 0: Dispatched process  7
Time slot  24
Time slot  25
Time slot  26
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
Time slot  27
        CPU 0: Processed  7 has finished
        CPU 0 stopped
```



```
MEMORY CONTENT:
000: 00000-003ff - PID: 06 (idx 000, nxt: 001)
001: 00400-007ff - PID: 06 (idx 001, nxt: 002)
002: 00800-00bff - PID: 06 (idx 002, nxt: 003)
        009e7: 0a
003: 00c00-00fff - PID: 06 (idx 003, nxt: 004)
004: 01000-013ff - PID: 06 (idx 004, nxt: -01)
005: 01400-017ff - PID: 05 (idx 000, nxt: 006)
        017e8: 15
006: 01800-01bff - PID: 05 (idx 001, 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: 020)
019: 04c00-04fff - PID: 01 (idx 000, nxt: -01)
        04c14: 64
020: 05000-053ff - PID: 06 (idx 003, nxt: -01)
024: 06000-063ff - PID: 05 (idx 000, nxt: 025)
        06014: 66
025: 06400-067ff - PID: 05 (idx 001, nxt: -01)
```

5 Conclusion

In this assignment, we gain knownledge about:

A simple operation system implement by simulation of scheduler using priority feedback queue.

How memory manage in operation system. The relation between physical memory and virtual memory.

Benefits and Harmful of some common scheduling algorithm.