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

OPERATING SYSTEMS

Report for Assignment #02

Simple Operating System

Advisor: Tran Viet Toan
Students: Vo Minh Khoa - 1812670
Tran Long Vi - 1814804

HO CHI MINH CITY, DECEMBER 2020

Contents

1 Scheduling

1.1 Questions

Question 1: What is the advantage of using priority feedback queue in comparison with other scheduling algorithms you have learned?
The priority feedback queue algorithm: The Priority feedback queue is based on the multilevel feedback queue algorithm used on the Linux kernel.
The multilevel feedback queue (MFQ) algorithm: In the Multilevel Feedback Queue system, the scheduler can move processes between queues according to its observed properties, changing the process's priority.
we can apply MFQ to the priority feedback queue algorithm that applied MFQ by using two queues with priority: ready queue and run queue.

- Ready queue is a priority queue that inherits a portion of Priority Scheduling (PS), every time the CPU receives the process with the highest priority from the ready queue.

- Run queue contains processes that are waiting to continue to execute after their slots have expired but their process has not yet been completed.

Ready queue has a higher priority than the run queue so it is executed first by the CPU. When the CPU moves to the next slot, it looks for the process in the ready queue.
When the ready queue is empty, the processes on the run queue switch to the ready queue to consider the next slot. In this situation, we promote Process.
When the process at the ready queue runs out of the given quantum time it goes down to the run queue. Both queues are the priority queue, the priority level is based on the priority level of the process in the queue. In this case, we downgrade Process.
Disadvantages of other schedulings are:

- First Come First Served (FCFS):
  For this algorithm, due to using the nonpreemptive strategy in the process, if a process starts, the CPU executes the process until it finishes.Therefore, the processes behind (possibly the processes with a short CPU burst) in the queue have to wait longer.

- Shortest Job First (SJF):
  + It is necessary to estimate the amount of time it takes for the next process of the CPU to first while this is often not possible in practice.
  + Process with long CPU burst has more waiting time or indefinitely delay when there are many processes with short CPU burst in queue.

- Round Robin (RR):
  + Throughput is often large depending on the time quantum. If time quantum is too large, RR will like FCFS.
  + If the time quantum is too small, the CPU context switch will increase a lot causing OS overhead and reduce CPU utilization.

- Priority Scheduling (PS):
  + There must be a scheduler with processes with equal priorities.
  + Processes with lower priorities may not have a chance to be executed.

The priority feedback queue (PFQ) algorithm can overcome the disadvantages of other scheduling with the following points:

- This algorithm is based on the Multilevel Feedback Queue, which uses two different queues and moves the processes back and forth between queues until the process is completed. So respond time can be shorter.
- Turnaround time is optimized: PFQ runs the process according to the time quantum then changes the process's priority, so it learns from the process's past behavior and then predicts its future behavior. In this way, the PFQ tries to run a short process first, so to optimize the turnaround time.
- It's also more flexible than the scheduling algorithms have been learned.
- Ensure fairness for processes by applying Round Robin style.
- Avoid starvation: Since PFQ also needs to prioritize processes with high priority, if there are no different queues, the process with low priority will not take its turn, it will still be executed before the processes with higher priority after the slot has been completed.

1.2 Implementation

1.2.1 Priority queue

We use *enqueue()* and *dequeue()* in *queue.c* to implement priority queue in this project.

- *enqueue()*: Put a new process into queue if queue is empty.

```
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++;
    }
}
```

Figure 1: *enqueue() implementation*

- *dequeue()*: Retrieve process with highest priority in queue, update queue status when retrieving element.

```c
struct pcb_t * dequeue(struct queue_t * q) {
    if(q->size!=0)
    {
        struct pcb_t* temp_proc=NULL;
        int max_priority=0;
        int max_idx=0;
        int i;
        for(i=0;i<q->size;++i)
        {
            if(q->proc[i]->priority>max_priority)
            {
                temp_proc=q->proc[i];
                max_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;
}
```

Figure 2: *dequeue() implementation*

1.2.2 Scheduler

The scheduler is used to manage updating the processes that will be executed for the CPU. We use *get_proc()* in *sched.c* to implement scheduler in this project.
get_proc() will return the process at queue "ready". If queue is empty at the time function is called, update queue again with processes that are waiting for next slot in queue "run". On the contrary, we find the process with high priority from this queue.

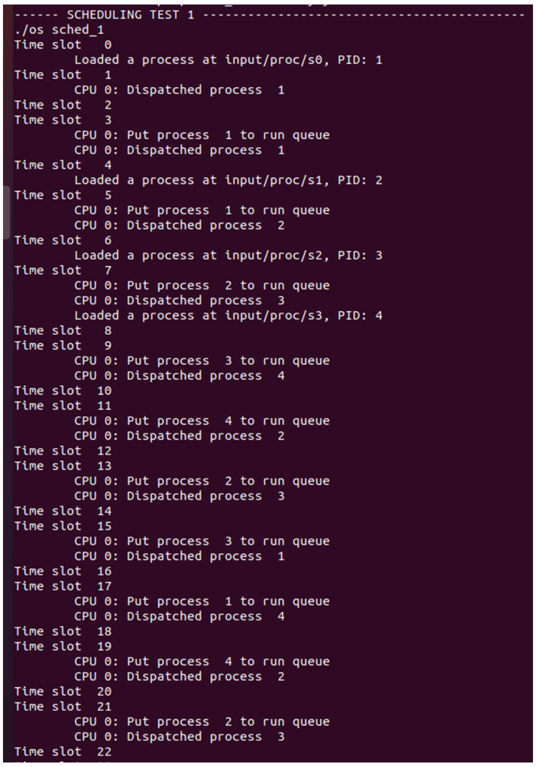
```c
struct pcb_t * get_proc(void) {
    struct pcb_t * proc = NULL;
    pthread_mutex_lock(&queue_lock);
    if(ready_queue.size==0)
    {
        int i = 0
        while(i <= MAX_QUEUE_SIZE)
        {
            i++;
            ready_queue.proc[i]=run_queue.proc[i];
            run_queue.proc[i]=NULL;
        }
        ready_queue.size=run_queue.size;
        run_queue.size=0;
    }
    proc=dequeue(&ready_queue);
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

Figure 3: *get_proc() implementation*

After run command *make test_sched* by terminal, we can see result like this:

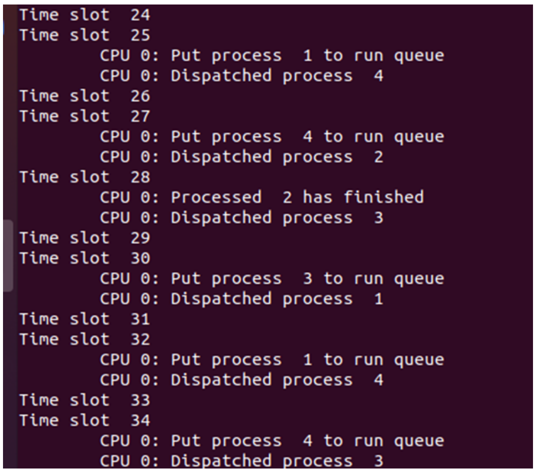
```
------ SCHEDULING TEST 0 -----------------------------------------
./os sched_0
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  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
```

Figure 4: *Scheduling Test 1 result*



```
------ 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
```

Figure 5: *Scheduling Test 2 result (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
```

Figure 6: *Scheduling Test 2 result (2)*

```
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
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
```

Figure 7: *Scheduling Test 2 result (3)*

1.3 Gantt Diagrams

In this Assignment, my team draw Gantt Diagrams to describe how processes are executed by the CPU in 2 situations.

1.3.1 Test 1

In this situation, CPU executes 2 processes p1 and p2 in 23 time slots.
Gantt Diagrams:

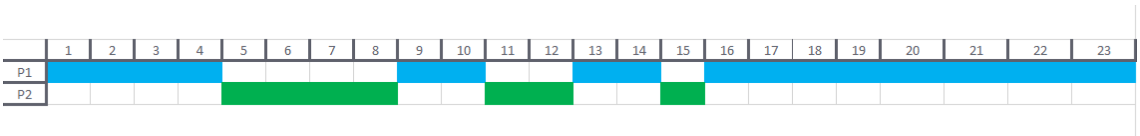
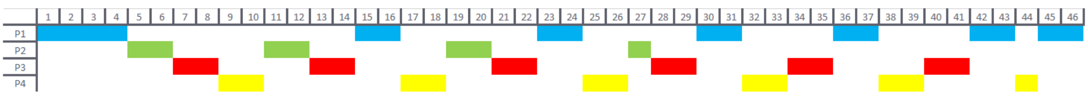


Figure 8: *Gantt diagram in test 1*

1.3.2 Test 2

In this situation, CPU executes 4 processes p1, p2, p3 and p4 in 48 time slots.
Gantt Diagrams:



2 Memory management

2.1 Questions

Question 2: What is the advantage and disadvantage of segmentation with paging?
Segmentation with paging is a mechanism Virtual Memory Engine (VME) used to manage memory.
The segmentation technique satisfies the program's need to demonstrate the logical structure of the program, but it leads to the situation of having to allocate memory blocks of different sizes for segments in the physical memory. Therefore, if we do Paging the segmentation, the problem will be better resolved.
Segmentation with paging:

- Address space is a set of segments, each segment is divided into many pages.
- When a process is put into OS, it will allocate to the process necessary pages to contain all segments of the process.
- Cuz the process uses virtual addresses to access RAM, we should set the virtual addresses of the allocated pages to be adjacent.

Advantage of Segmentation with paging:

- Saving memory, using memory effectively.
- Allocating intermittent memory in a simpler way.
- Taking advantages of Paging and Segmentation and minimize disadvantages of each other.
- Sharing data between processes is more flexible
- Fixing the size of the page too large by paging in each segment.
- There is no foreign fragmentation.

Disadvantage of Segmentation with paging:

- Size of the process is limited by the size of physical memory.
- It is difficult to maintain multiple processes at the same time in memory.
- Page tables need a lot of memory space, so it is not good for systems with small RAM.

2.2 Implement and show the status of RAM

We use *alloc_mem()* and *free_mem()* in *mem.c* to implement the status of RAM in this project.
alloc_mem(): Allocation memory for the process and save the address of the first byte in the allocated memory region.

```c
addr_t alloc_mem(uint32_t size, struct pcb_t * proc) {
    pthread_mutex_lock(&mem_lock);
    addr_t ret_mem = 0;
    uint32_t num_pages = ((size % PAGE_SIZE) == 0) ? size / PAGE_SIZE :
        size / PAGE_SIZE + 1;
    int mem_avail = 0;
    int i;
    int num_avail_pages = 0;
    for(i = 0; i < NUM_PAGES; i++){
        if(_mem_stat[i].proc == 0){
            num_avail_pages++;
            if(num_avail_pages == num_pages && proc->bp + num_pages * PAGE_SIZE <= RAM_SIZE){
                mem_avail = 1;
                break;
            }
        }
    }

    if (mem_avail) {
        ret_mem = proc->bp;
        proc->bp += num_pages * PAGE_SIZE;int num_alloc_pages = 0;
        int pre_index;
        addr_t cur_vir_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 = num_alloc_pages;

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

Figure 9: *alloc_mem() implementation(1)*

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

                cur_vir_addr =ret_mem + (num_alloc_pages << OFFSET_LEN) ;

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

                        cur_page_table->table[cur_page_table->size].v_index = page_idx;
                        cur_page_table->table[cur_page_table->size].p_index = i;

                        cur_page_table->size++;

                        found = 1;
                        break;
                    }
                }

                if(!found){
                    seg_table->table[seg_table->size].v_index = seg_idx;
                    seg_table->table[seg_table->size].pages = (struct page_table_t *)malloc(sizeo

                    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++;
                }
```

Figure 10: *alloc_mem() implementation(2)*

```c
                num_alloc_pages++;
                if(num_alloc_pages == num_pages){
                    _mem_stat[i].next = -1;
                    break;
                }
            }
        }
    }

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

Figure 11: *alloc_mem() implementation(3)*

free_mem(): Release memory region allocated.

```c
int free_mem(addr_t address, struct pcb_t * proc) {
    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 num_free_pages = 0;
                    addr_t cur_vir_addr = (num_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(cur_vir_addr);
                        page_idx=get_second_lv(cur_vir_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_i
                                        int m;
                                        for(m = l; m < proc->seg_table->table[k].pages->size - 1;
                                            proc->seg_table->table[k].pages->table[m]= proc->seg_

                                        proc->seg_table->table[k].pages->size--;
```

Figure 12: *free_mem() implementation(1)*

```c
                        if(proc->seg_table->table[k].pages->size == 0){
                            free(proc->seg_table->table[k].pages);
                            for(m = k; m < proc->seg_table->size - 1; m++)
                                proc->seg_table->table[m]= proc->seg_table->table
                            proc->seg_table->size--;
                        }
                        found = 1;
                        break;
                    }
                }
            }
        }

        p_index = _mem_stat[p_index].next;
        num_free_pages++;
    }
    while(p_index != -1);
    valid = 1;
    }
    break;
    }
  }
}

pthread_mutex_unlock(&mem_lock);

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

Figure 13: *free_mem() implementation(2)*

After run command *make test_mem* by terminal, we can see result like this:

```
khoa@ubuntu:~/Desktop/Assignment2-OS/source_code$ make mem
gcc -Iinclude  -Wall -c -g src/paging.c -o obj/paging.o
gcc -Iinclude  -Wall -g obj/paging.o obj/mem.o obj/cpu.o obj/loader.o -o mem -lpthread
khoa@ubuntu:~/Desktop/Assignment2-OS/source_code$ 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 -----------------------------------
./mem input/proc/m1
NOTE: Read file output/m1 to verify your result (your implementation should print nothing)
khoa@ubuntu:~/Desktop/Assignment2-OS/source_code$
```

Figure 14: *test_mem result*

2.3 Explain file input m0

```
1 7
alloc 13535 0
alloc 1568 1
free 0
alloc 1386 2
alloc 4564 4
write 102 1 20
write 21 2 1000
```

- In first line, “1” is the priority of process m0, and “7” is the instruction number of the input file.
- In 2nd line, comment “alloc 13535” will allocate 14 _mem_stat pages (from 000 to 013) and store the address of the first allocated byte in register 0.
- In 3rd line, comment “alloc 1568” will allocate 2 pages (014 and 015) and store the address of the first allocated byte in register number 1.
- In 4th line, comment “free 0” will release the allocated memory from the alloc command in register number 0, which means there are only pages 014 and 015 at this time.
- In 5th line, comment “alloc 1386” will allocate 2 pages (000 and 001), then store the address of the first allocated byte in register number 2.
- In 6th line, comment “alloc 4564” will allocate 5 pages (from 002 to 006) and store the address of the first allocated byte in register number 4.
- In 7th line, comment “write 102 1 20” will write the value 102 to the place where the address is equal to register address 1 plus an offset of 20 that will output 0x03814.
  The physical address can be calculated by $Physical\ address = Base\ address + Offset$, in which, $Base\ address = 0x03800$ is the first byte and $offset = 20$.
- In final line, similarly, the result would be 003e8.

3 Overall

In file *mem.c*, in addition to the *alloc_mem()* and *free_mem()*, we also implement *get_page_table()* and *translate()*. Besides, we fixed *read_mem()* and *write_mem()*, too.

get_page_table() is used to find the paging table from segment:

```c
static struct page_table_t *get_page_table(
    addr_t index,
    struct seg_table_t *seg_table)
{
    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;
}
```

Figure 15: *get_page_table() implementation*

translate() is used to map virtual addresses to physical addresses:

```c
static int translate(
    addr_t virtual_addr,   // Given virtual address
    addr_t *physical_addr, // Physical address to be returned
    struct pcb_t *proc)
    addr_t offset = get_offset(virtual_addr);
    addr_t first_lv = get_first_lv(virtual_addr);
    addr_t second_lv = get_second_lv(virtual_addr);
    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)
        {
            *physical_addr = page_table->table[i].p_index * PAGE_SIZE + offset;
            return 1;
        }
    }
    return 0;
```

Figure 16: *translate() implementation*

read_mem() and *write_mem()* is added mutex to avoid asynchronous situation when they access to [ram] at the same time.

```c
pthread_mutex_t mutex;
int read_mem(addr_t address, struct pcb_t * proc, BYTE * data) {
    addr_t physical_addr;
    if (translate(address, &physical_addr, proc)) {
        pthread_mutex_lock(&mutex);
        *data = _ram[physical_addr];
        pthread_mutex_unlock(&mutex);
        return 0;
    }else{
        return 1;
    }
}

int write_mem(addr_t address, struct pcb_t * proc, BYTE data) {
    addr_t physical_addr;
    if (translate(address, &physical_addr, proc)) {
        pthread_mutex_lock(&mutex);
        _ram[physical_addr] = data;
        pthread_mutex_unlock(&mutex);
        return 0;
    }else{
        return 1;
    }
}
```

Figure 17: *read_mem() and write_mem() after added mutex*

4 Simulation

After run command *"make test_all"* by Terminal, we can see result like below:

```
----- OS TEST 1 ---------------------------------------------------
./os os_1
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/s3, PID: 2
        CPU 3: Dispatched process  2
Time slot   3
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/m1, PID: 3
Time slot   4
        CPU 0: Dispatched process  3
Time slot   5
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
        Loaded a process at input/proc/s2, PID: 4
Time slot   6
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
Time slot   7
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  3
        Loaded a process at input/proc/m0, PID: 5
        CPU 2: Dispatched process  1
Time slot   8
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
        Loaded a process at input/proc/p1, PID: 6
Time slot   9
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  6
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  4
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  3
Time slot  10
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s0, PID: 7
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  7
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  4
Time slot  11
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  2
```

Figure 18: *Final result (1)*

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

Figure 19: *Final result (2)*

```
Time slot  21
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  6
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  7
Time slot  22
        CPU 2: Processed  4 has finished
        CPU 2 stopped
Time slot  23
        CPU 0: Put process  8 to run queue
        CPU 0: Dispatched process  8
        CPU 1: Processed  6 has finished
        CPU 1 stopped
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  7
Time slot  24
        CPU 0: Processed  8 has finished
        CPU 0 stopped
Time slot  25
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  7
Time slot  26
        CPU 3: Processed  7 has finished
        CPU 3 stopped
```

Figure 20: *Final result (3)*

```
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: 05 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 05 (idx 001, nxt: 004)
004: 01000-013ff - PID: 05 (idx 002, nxt: 005)
005: 01400-017ff - PID: 05 (idx 003, nxt: 006)
006: 01800-01bff - PID: 05 (idx 004, nxt: -01)
011: 02c00-02fff - PID: 06 (idx 000, nxt: 012)
012: 03000-033ff - PID: 06 (idx 001, nxt: 013)
013: 03400-037ff - PID: 06 (idx 002, nxt: 014)
014: 03800-03bff - PID: 06 (idx 003, nxt: -01)
021: 05400-057ff - PID: 01 (idx 000, nxt: -01)
        05414: 64
022: 05800-05bff - PID: 06 (idx 000, nxt: 023)
023: 05c00-05fff - PID: 06 (idx 001, nxt: 031)
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 002, nxt: 032)
        07de7: 0a
032: 08000-083ff - PID: 06 (idx 003, nxt: 033)
033: 08400-087ff - PID: 06 (idx 004, nxt: -01)
NOTE: Read file output/os_1 to verify your result
khoa@ubuntu:~/Desktop/Assignment2-OS/source_code$
```

Figure 21: *Final result (4)*