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

OPERATING SYSTEM (CO2017)

Assignment

Simple Operating System

Advisor: Le Thanh Van
Students: Nguyen Thi Ngoc Nhi - 2052632.
Bui Anh Kent - 2053087.
Do Huu Thanh Thien - 2053453

HO CHI MINH CITY, DECEMBER 2022

Contents

1 Member list & Workload

No.	Fullname	Student ID	Problems	Percentage of work
1	Nguyen Thi Ngoc Nhi	2052632	- Implement Memory Management - Writing report	100%
2	Bui Anh Kent	2053087	- Writting Report - Answearing Question	100%
2	Do Huu Thanh Thien	2053453	- Implement Queue And Scheduler	100%

2 Question

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

2.1.1 Introduction

First, you need to understand how the scheduling method works so that you can understand the priority feedback method. How the queue is organized and the settings are selected. We will have many processes ruining at the same time, but for instance, if our CPU only has only 1 processor then it will only be able to process 1 instruction at a time.

With that in mind, we want to arrange and choose which instruction to process first in order to have the most efficiency with our scheduling criteria (For example **CPU utilization, Throughput, Turnaround-time, Waiting time, Average turn-around time, Respond time...**)

But What is A Priority Queue?

A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher-priority elements are served first. However, if elements with the same priority occur, they are served according to their order in the queue.

It's also based on a Multilevel Feedback Queue (**MFQ**) that is being used on the Linux Kernel.

And What is A MFQ?

In the Multilevel Feedback Queue system, a scheduler can freely move processes between queues based on their attributes and their priority.

***Example*:**

If a process uses it entire time slice, it is moved to a lower priority queue. Processes that block before the time slice ends are moved to a higher priority queue.

How does Priority Feedback Queue uses MFQ?

It uses 2 queue that have the priority attribute one of which is the **ready queue** and the other is **run queue**.

- *ready queue:* This queue contains processes in the order of their priority values, with the one having the highest priority value will be chosen to be executed once the CPU is free to do so.

- *run queue:* This queue contains processes waiting for the next execution as their work is not completed. When the CPU is free and the ready queue is empty , the processes of this queue are moved back to ready queue.

How does a priority queue run?

- A process is pushed into ready queue and wait for CPU (**enqueue**)
- CPU will run process round-robin style
- Each process is allow to run in a certain period of time
- Then CPU will pause that process (if haven't finished) and push to run queue
- CPU continues to take process from ready queue and run **dequeue**.

2.1.2 What are the differences between Priority Queue Scheduling and other Scheduling Method?

First Come First Served (FCFS)

- It used non-preemptive scheduling so when a process start to execute, it will not pause.
- If a process has a long burst time. Other processes might have a longer waiting time.

Shortest Job First(SJF)

- Process that has a long burst time will lead to longer waiting time or indefinite blocking if many short burst time appear. It will end up in a **starvation** state
- The real difficulty with SJF is knowing the length of the next CPU request or burst.

Shortest Remaining Job First(SRJF)

- The preemptive version of SJF scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute.
- Both SJF and SRJF are practically not feasible as it is not possible to predict the burst time of the processes.
- Both SJF and SRJF may lead to process starvation as long processes may be held off indefinitely if short processes are continually added.

Round Robin (RR)

- This algorithm mainly depends on the time quantum. Very large time quantum makes RR same as the FCFS while a very small time quantum will lead to the overhead as the context switch will happen again and again after very small intervals.
- The major advantage of this algorithm is that all processes get executed one after the other which does not lead to starvation of processes or waiting by the process for a quite long time to get executed
- Usually has a high Throughput

Priority Scheduling

- Require a scheduler with processes that have similar priorities.
- The major problem is starvation or indefinite blocking. It may so happen that in the stream of processes, system keeps executing high-priority processes and low-priority processes never get executed.

2.1.3 Advantages of using PFQ

+ Shorter Respond Time
+ More optimized Turn Around Time. Unlike SJF we need to know about the running time of a process but still maintain a reliable turnaround time. PFQ executes projects according to time quantum and then change the priority of a process
+ PFQ runs a process depending on time quantum and changes the priority of a process.
+ Avoid starvation. Because PFQ priority process with a high priority

It is also more flexible than other scheduling methods

By using other scheduling data structures like

+ Priority Scheduling - each process has a priority to execute
+ Multilevel Queue - use many different level for processes
+ Round Robin - use time quantum

=> It's like MFQ take 1 "good part" of each method

2.1.4 Implementation

First, we need to implement the function enqueue and dequeue in queue.c

2.1.4.a enqueue():

Add a PCB to the waiting queue.

```c
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    /* TODO: put a new process to queue [q] */
    if (q->size == 0){
        q->proc[0] = proc;
        q->size++;
    }
    else{
        // enqueue in increasing order of prio
        int idx = 0;
        for (; idx<q->size; idx++){
            if (proc->prio < q->proc[idx]->prio)
                break;
        }

        for (int i=q->size-1; i>idx; i--){
            q->proc[i] = q->proc[i-1];
        }

        q->proc[idx] = proc;
        q->size++;
    }
}
```

2.1.4.b dequeue():

Return a PCB whose priority is the highest in the queue [q] and remember to remove it from q

```
struct pcb_t * dequeue(struct queue_t * q) {
    /* TODO: return a pcb whose prioprity is the highest
     * in the queue [q] and remember to remove it from q
     * */
    if (q->size == 0) return NULL;
    struct pcb_t* removeElement = q->proc[0];
    for (int i=0; i<q->size-1; i++){
        q->proc[i] = q->proc[i+1];
    }
    q->size--;
    return removeElement;
}
```

2.1.4.c Get_mlq_proc function()

Gets a process from PRIORITY [ready_queue]. Remember to use lock to protect the queue.

```
struct pcb_t * get_mlq_proc(void) {
    struct pcb_t * proc = NULL;
    /*TODO: get a process from PRIORITY [ready_queue].
     * Remember to use lock to protect the queue.
     * */
    pthread_mutex_lock(&queue_lock);
    int i;
    for (i = 0; i < MAX_PRIO; i++) {
        if (!empty(&mlq_ready_queue[i])) break;
    }
    if (i < MAX_PRIO) {
        proc = dequeue(&mlq_ready_queue[i]);
    }
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

Results

After compiling using **make sched** and **make test_sched**, we receive a result.

sched_0

Structure of sched_0 test file

```
2 1 2
0 s0 0
4 s1 3

```

Hình 1: In this test case, our CPU will handle 2 processes s0 and s1 with 23 timeslots

```
------ SCHEDULING TEST 0 -----------------------------------------
./os sched_0
Time slot   0
        Loaded a process at input/proc/s0, PID: 1 PRIO: 0
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
        Loaded a process at input/proc/s1, PID: 2 PRIO: 3
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   6
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   8
Time slot   9
        CPU 0: Put process  1 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  1
Time slot  12
Time slot  13
        CPU 0: Put process  1 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  1
Time slot  16
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
Time slot  17
Time slot  18
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  19
Time slot  20
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  21
Time slot  22
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  23
        CPU 0: Processed  2 has finished
        CPU 0 stopped
```

Hình 2: Result of the first scheduler

sched_1

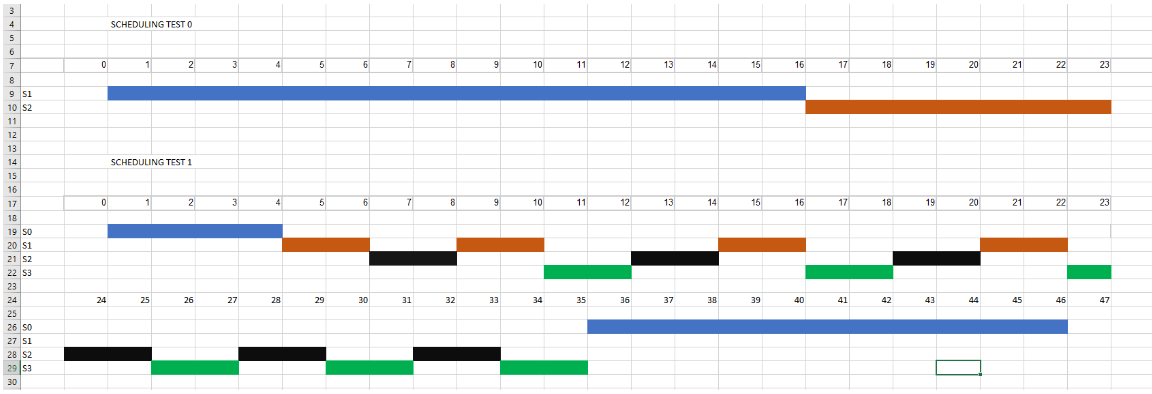
```
1   2 1 4
2   0 s0 1
3   4 s1 0
4   6 s2 0
5   7 s3 0
6
```

Hình 3: In this test case, our CPU will handle 4 processes s0, s1,s2 and s3 with 46 timeslots

```
----- SCHEDULING TEST 1 -------------------------------------------
./os sched_1
Time slot   0
        Loaded a process at input/proc/s0, PID: 1 PRIO: 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
        Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/s2, PID: 3 PRIO: 0
Time slot   6
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 PRIO: 0
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  12
Time slot  13
        CPU 0: Put process  4 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  2
Time slot  16
Time slot  17
        CPU 0: Put process  2 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  3
Time slot  20
Time slot  21
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  22
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  23
Time slot  24
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  25
Time slot  26
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
```

```
Time slot  20
Time slot  21
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  22
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  23
Time slot  24
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  25
Time slot  26
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  27
Time slot  28
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  29
Time slot  30
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  31
Time slot  32
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  33
Time slot  34
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  4
Time slot  35
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  36
Time slot  37
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  38
Time slot  39
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  40
Time slot  41
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  42
Time slot  43
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  44
Time slot  45
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  46
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

2.1.5 Gantt diagram



Hình 4: Gantt diagram for both sched_0 and sched_1

+ Average **Waiting** Time sched_0: 7.5s
+ Average **Turnaround** Time sched_0: 18.5s
+ Average **Waiting** Time sched_1: 22s
+ Average **Turnaround** Time sched_1: 33.25s

2.2 What is the advantage and disadvantage of segmentation with paging?

Why do we use Segmentation with Paging

The segmentation technique meets the needs of the program's logic, but the instructions are also allocate memory blocks of different sizes to segments in physical memory. This is more complicated than allocating pages with equal and fixed sizes. A good way to solve this is to combine **Segmentation** and **Paging**

The Idea of Segmentation with Paging

Address space is a collection of segments, each segment is divided into several pages. When a process is assigned to the OS, it will allocate to the process the pages that needed to be set. Even if a process required a small allocation of memory **(< page size)**, it will receive a full new page. Because the process used the virtual address to access RAM we might want to have the virtual address of the allocated pages next to each other, we don't need to do that for the physical address.

2.2.1 Advantages

- Optimized memory usage, use memory more efficiently

- Make allocating non-consecutive memory simpler
- Shared memory becomes more flexible through page sharing or segment table
- Overcome size overload by allocating fixed-size pages
- It doesn't cause external fragmentation

2.2.2 Disadvantages

- Internal Fragmentation still happens due to Paging algorithm
- Size of a process is limited by the size of the physical memory
- Hard to maintain lots of processes in memory $\Longrightarrow$ difficult to make our OS level of multi-programming
- If the segment table is not used then we don't need to save the **page table**. Page table needs a lot of memory space $\Longrightarrow$ Bad for systems with small RAM size

2.2.3 Implementation

In this section, we will implement 4 functions

- get_trans_table()
- translate()
- alloc_mem()
- free_mem()

2.2.3.a get_trans_table()

```c
/* Search for page table table from the a segment table */
static struct trans_table_t *
get_trans_table(addr_t index,                         // Segment level index
                struct page_table_t *page_table) { // first level table

  /*
   * TODO: Given the Segment index [index], you must go through each
   * row of the segment table [page_table] and check if the v_index
   * field of the row is equal to the index
   *
   * */

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

2.2.3.b translate()

```c
/* 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 trans_table_t *trans_table = NULL;
  trans_table = get_trans_table(first_lv, proc->page_table);
  if (trans_table == NULL) {
    return 0;
  }

  int i;
  for (i = 0; i < trans_table->size; i++) {
    if (trans_table->table[i].v_index == second_lv) {
      /* TODO: Concatenate the offset of the virtual addess
       * to [p_index] field of trans_table->table[i] to
       * produce the correct physical address and save it to
       * [*physical_addr]  */

      *physical_addr = (trans_table->table[i].p_index << OFFSET_LEN) | offset;

      // printf("1st: %05x | %05x = %05x\n", trans_table->table[i].p_index <<
      // OFFSET_LEN, offset, *physical_addr);

      return 1;
    }
  }
  return 0;
}
```

2.2.3.c alloc_mem()

```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?

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

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
   *    isValid. */

  int frame_index = 0;
  int prev = -1; // prev index
  for (int i = 0; i < NUM_PAGES; i++) {
    // update status pages in physical address
    if (_mem_stat[i].proc == 0) {
      // add/update [proc], [index], and [next] field
      _mem_stat[i].proc = proc->pid;
      _mem_stat[i].index = frame_index;
      if (prev != -1) { // not initial page, update last page
        _mem_stat[prev].next = i;
      }
      prev = i;

      /*Add entries to segment table page tables of [proc]
      to ensure accesses to allocated memory slot is
      valid.
      */
      struct page_table_t *page_table = proc->page_table;
      if (page_table->table[0].next_lv == NULL) {
        // if page table is null, set page table size = 0
        page_table->size = 0;
      }

      addr_t virtual_address = ret_mem + (frame_index << OFFSET_LEN);
      addr_t segment_idx =
          get_first_lv(virtual_address); // get the first layer index
      addr_t page_idx =
          get_second_lv(virtual_address); // get the second layer index
      struct trans_table_t *trans_table =
          get_trans_table(segment_idx, page_table);

      // if there is not trans_table in seg -> create new trans_table
      if (trans_table == NULL) {
        int idx = page_table->size;
        page_table->table[idx].v_index = segment_idx;
        trans_table =
```

```
              (struct trans_table_t *)malloc(sizeof(struct trans_table_t));
          page_table->table[idx].next_lv =
              (struct trans_table_t *)malloc(sizeof(struct trans_table_t));
          trans_table = page_table->table[idx].next_lv;
          page_table->size++;
        }

        // update mem_stat
        int idx = trans_table->size++;
        trans_table->table[idx].v_index = page_idx;
        trans_table->table[idx].p_index = i;
        if (frame_index == (num_pages - 1)) {
          // the last element's next field = -1
          _mem_stat[i].next = -1;
          break;
        }
        frame_index++; // update page index
      }
    }
  }
  pthread_mutex_unlock(&mem_lock);
  return ret_mem;
}
```

2.2.3.d free_mem()

```

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 virtual_address = address; // virtual address to free in process
  addr_t physical_addr = 0;         // physical address to free in memory
  // printf("translate(virtual_address, &physical_addr, proc):%d \n",
  // translate(virtual_address, &physical_addr, proc));
  if (translate(virtual_address, &physical_addr, proc) == 0) {
    pthread_mutex_unlock(&mem_lock);
    return 1;
  }

  // clear physical page in mem
  int num_pages = 0;
  int i = 0;

  // get physical index used by memory block
  for (i = physical_addr >> OFFSET_LEN; i != -1; i = _mem_stat[i].next) {
    num_pages++;

    // Set flag [proc] of physical page use by the memory block
    // back to zero to indicate that it is free.
    _mem_stat[i].proc = 0;
  }
  for (i = 0; i < num_pages; i++) {
    addr_t addr = virtual_address + i * PAGE_SIZE;
    addr_t first_lv = get_first_lv(addr);
    addr_t second_lv = get_second_lv(addr);
```

```
    struct trans_table_t *trans_table =
        get_trans_table(first_lv, proc->page_table);
    if (trans_table == NULL)
      continue;
    // remove
    for (int j = 0; j < trans_table->size; j++) {
      if (trans_table->table[j].v_index == second_lv) {
        trans_table->size--;
        trans_table->table[j] = trans_table->table[trans_table->size];
        break;
      }
    }
    // remove table
    if (trans_table->size == 0 && proc->page_table != NULL) {
      for (int k = 0; k < proc->page_table->size; k++) {
        if (proc->page_table->table[k].v_index == first_lv) {
          proc->page_table->size--;
          proc->page_table->table[k] =
              proc->page_table->table[proc->page_table->size];
          proc->page_table->table[proc->page_table->size].v_index = 0;
          free(proc->page_table->table[proc->page_table->size].next_lv);
        }
      }
    }
  }

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

2.2.4 RAM Status

2.2.4.a m0

Input of 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
```

Output of m0:

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
  003e8: 14
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: 64
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
```

Explanation:

- alloc 13535 0

  After alloc 13535 0, RAM of status will be:

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
```

  .In order to alloc 13535 bytes, we will need 13535/1024 = 14 page frames. Then the virtual address of the first byte, which is mapped to physical address (0x00000), will be stored in register 0

- alloc 1568 1

  After alloc 1568 0, RAM of status will be:

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
```

  In order to alloc 1568 bytes, we will need 1568/1024 = 2 page frames. Then the virtual address of the first byte, which is mapped to physical address (0x03800), will be stored in register 1 of the process

- free 0

  After free 0, RAM of status will be:

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

  The OS now deallocates the amount of allocated memory whose first byte is stored in register 0. In this case, register 0 stored the address of the first byte of an allocated memory of 14 page frames.

- alloc 1386 2

  After alloc 1386 2, RAM of status will be:

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

  In order to alloc 1386 bytes, we will need 1386/1024 = 2 page frames. Then the virtual address of the first byte, which is mapped to physical address (0x00000), will be stored in register 2

- alloc 4564 4

  After alloc 4564 4, RAM of status will be:

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
```

  In order to alloc 4564 bytes, we will need 4564/1024 = 5 page frames. Then the virtual address of the first byte, which is mapped to physical address (0x00800), will be stored in register 4

2.2.4.b m1

Input of m1:

```
1 8
alloc 13535 0
alloc 1568 1
free 0
alloc 1386 2
alloc 4564 4
free 2
free 4
free 1
```

Output of m1:

Explanation:

- alloc 13535 0

  After alloc 13535 0, RAM of status will be:

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
```

In order to alloc 13535 bytes, we will need 13535/1024 = 14 page frames. Then the virtual address of the first byte, which is mapped to physical address (0x00000), will be stored in register 0

- alloc 1568 1

After alloc 1568 0, RAM of status will be:

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
008: 02000-023ff - PID: 01 (idx 008, nxt: 009)
009: 02400-027ff - PID: 01 (idx 009, nxt: 010)
010: 02800-02bff - PID: 01 (idx 010, nxt: 011)
011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)
012: 03000-033ff - PID: 01 (idx 012, nxt: 013)
013: 03400-037ff - PID: 01 (idx 013, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
```

In order to alloc 1568 bytes, we will need 1568/1024 = 2 page frames. Then the virtual address of the first byte, which is mapped to physical address (0x03800), will be stored in register 1 of the process

- alloc 1386 2

After alloc 1386 2, RAM of status will be:

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

In order to alloc 1386 bytes, we will need 1386/1024 = 2 page frames. Then the virtual address of the first byte, which is mapped to physical address (0x00000), will be stored in register 2

- alloc 4564 4

After alloc 4564 4, RAM of status will be:

```
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
```

In order to alloc 4564 bytes, we will need $4564/1024 = 5$ page frames. Then the virtual address of the first byte, which is mapped to physical address (0x00800), will be stored in register 4

- free 2

After free 2, RAM of status will be:

```
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
```

The OS now deallocates the amount of allocated memory whose first byte is stored in register 2. In this case, register 2 stored the address of the first byte of an allocated memory of 2 page frames.

- free 4

After free 4, RAM of status will be:

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

The OS now deallocates the amount of allocated memory whose first byte is stored in register 4. In this case, register 4 stored the address of the first byte of an allocated memory of 5 page frames.

- free 1

After free 1, RAM of status will be:

```
* Empty memory *
```

The OS now deallocates the amount of allocated memory whose first byte is stored in register 1. In this case, register 1 stored the address of the first byte of an allocated memory of 2 page frames. Therefore, the memory is now completely

2.3 What will happen if the synchronization is not handled in your simple OS? Illustrate by example the problem of your simple OS if you have any

2.3.1 Results of Simulation.

First, there can be an issue with (run queue and ready queue). The system will experience an error if there are two threads attempting to enqueue while the ready queue has one slot available and is nearly full. The error may appear as follows:

- Thread 1 check the queue (not full, remain 1 slot)
- Thread 2 check the queue (not full, remain 1 slot)
- Thread 1 enqueue (valid)
- Thread 2 enqueue (error occurs because queue is full)

When there is only one process left in the queue and many CPUs attempts to dequeue, the same issue may arise.

Assuming that two distinct threads attempt to allocate the same page to themselves, both the first thread and the second thread may discover that the page at address 0x00 has not been used. The decision was made for both threads to allocate that page. As a result, two processes will both record the same memory frame. One process will run out of memory because the _mem_stat table will only list one of the two threads as the real owner. The deallocation of RAM could also be problematic.

Thread A, for example, tries to set the flag to signal that the page is free. Thread B notices that the page is free and requests that it be allocated. Thread A, on the other hand, continues its work, which is to remove the unused items from that page's page table and segment table. As a result, when thread B attempts to reference that page, it will fail.

Thirdly, if we do not use the mutex mechanism, there is no way the timer will work. The timeslot keeps incrementing despite unfinished tasks.

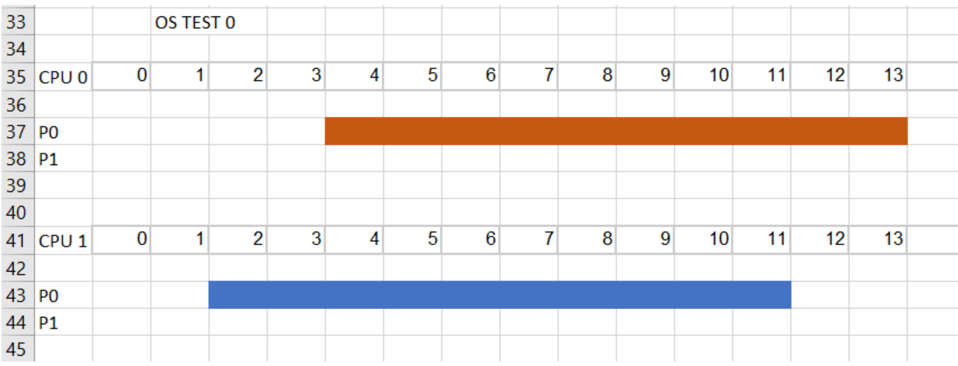
Finally, we run make all and make test_all to get the result

Memory States

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

Inside OS_0

```
----- OS TEST 0 ----------------------------------------------------
./os os_0
Time slot   0
        Loaded a process at input/proc/p0, PID: 1 PRIO: 0
Time slot   1
        CPU 1: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/p1, PID: 2 PRIO: 3
Time slot   3
        CPU 0: Dispatched process  2
Time slot   4
Time slot   5
Time slot   6
Time slot   7
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 1: Processed  1 has finished
        CPU 1 stopped
Time slot  12
Time slot  13
        CPU 0: Processed  2 has finished
        CPU 0 stopped
NOTE: Read file output/os_0 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)
```

Inside OS_1

```
    ----- OS TEST 1 -----------------------------------------------------
./os os_1
Time slot   0
        Loaded a process at input/proc/p0, PID: 1 PRIO: 3
Time slot   1
Time slot   2
        Loaded a process at input/proc/s3, PID: 2 PRIO: 2
        CPU 2: Dispatched process  1
Time slot   3
        CPU 3: Dispatched process  2
        Loaded a process at input/proc/m1, PID: 3 PRIO: 1
Time slot   4
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  3
Time slot   5
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/s2, PID: 4 PRIO: 0
Time slot   6
        CPU 2: Put process  3 to run queue
        CPU 2: Dispatched process  4
Time slot   7
        Loaded a process at input/proc/m0, PID: 5 PRIO: 0
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  5
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
        CPU 0: Dispatched process  3
Time slot   8
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
        Loaded a process at input/proc/p1, PID: 6 PRIO: 1
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  6
Time slot   9
        CPU 1: Put process  5 to run queue
        CPU 1: Dispatched process  5
Time slot  10
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
        Loaded a process at input/proc/s0, PID: 7 PRIO: 2
Time slot  11
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  3
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
        CPU 1: Put process  5 to run queue
        CPU 1: Dispatched process  5
Time slot  12
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  3
```

```
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
Time slot  13
        CPU 1: Put process  5 to run queue
        CPU 1: Dispatched process  5
Time slot  14
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
        CPU 1: Processed  5 has finished
        CPU 1: Dispatched process  7
Time slot  15
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
        CPU 3: Processed  3 has finished
        CPU 3: Dispatched process  2
        Loaded a process at input/proc/s1, PID: 8 PRIO: 3
Time slot  16
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  17
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
        CPU 2: Processed  4 has finished
        CPU 2: Dispatched process  1
Time slot  18
        CPU 3: Processed  2 has finished
        CPU 3: Dispatched process  8
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  19
        CPU 0: Processed  6 has finished
        CPU 0 stopped
Time slot  20
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  8
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  21
Time slot  22
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  8
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
Time slot  23
Time slot  24
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  8
        CPU 2: Processed  1 has finished
        CPU 2 stopped
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  25
        CPU 3: Processed  8 has finished
        CPU 3 stopped
```

```
Time slot  26
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  27
Time slot  28
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  29
        CPU 1: Processed  7 has finished
        CPU 1 stopped
NOTE: Read file output/os_1 to verify your result
```

Gantt Chart for the OS test