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

OPERATING SYSTEMS (CO2017)

Assignment (Semester 221)

" Simple Operating System "

Lecturer: Lê Thanh Vân

Students: Nguyễn Tôn Minh - 2052600 - CC04
Đỗ Nguyễn Huy Hiệu - 2053003 - CC01
Bùi Ngọc Nam Anh - 2052833 - CC04

HO CHI MINH CITY, NOVEMBER 2022

Table of Contents

1 TEAM MEMBER

Student Name	**Student ID**
Nguyễn Tôn Minh	2052600
Đỗ Nguyễn Huy Hiệu	2053003
Bùi Ngọc Nam Anh	2052833

2 THEORY OF THE ASSIGNMENT

2.1 Schedulers, Dispatchers and their uses

Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types:

- Long-Term Scheduler
- Short-Term Scheduler
- Medium-Term Scheduler

Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Bảng 1: Comparison among Schedulers

Dispatcher

It is a special type of program that comes into play only after the scheduler. Once a scheduler completes selecting its processes, the dispatcher then takes that intended process to its desired queue/ state. A dispatcher is basically a module that provides a process with total control over the CPU (after the short-term scheduler finally selects it). This function requires the following:

- Context switching
- Switching to the user mode
- Jumping into the user program's proper location for restarting that given program

2.2 Synchronization

Definition:

An operating system is a software that manages all applications on a device and basically helps in the smooth functioning of our computer. Because of this reason, the operating system has to perform many tasks, and sometimes simultaneously. This isn't usually a problem unless these simultaneously occurring processes use a common resource.

Uses in OS:

Let us take a look at why exactly we need Process Synchronization. For example, If a processA is trying to read the data present in a memory location while another processB is trying to change the data present at the same location, there is a high chance that the data read by the processA will be incorrect.

Different elements/sections of a program:

- Entry Section: The entry Section decides the entry of a process.
- Critical Selection: Critical section allows and makes sure that only one process is modifying the shared data.
- Exit Section: The entry of other processes in the shared data after the execution of one process is handled by the Exit section.
- Remainder Section: The remaining part of the code which is not categorized as above is contained in the Remainder section.

2.3 Multilevel Feedback Queue

In computer science, a multilevel feedback queue is a scheduling algorithm. Scheduling algorithms are designed to have some process running at all times to keep the central processing unit (CPU) busy. The multilevel feedback queue extends standard algorithms with the following design requirements:

- Separate processes into multiple ready queues based on their need for the processor.
- Give preference to processes with short CPU bursts.
- Give preference to processes with high I/O bursts. (I/O bound processes will sleep in the wait queue to give other processes CPU time.)

2.4 Memory Management

In a multiprogramming computer, the operating system resides in a part of memory and the rest is used by multiple processes. The task of subdividing the memory among different processes is called memory management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

Memory Management is required by:

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.

3 SCHEDULER

3.1 Implementation

3.1.1 queue.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;
  // find correct position for proc
  int pos;
  for (pos=0; pos < q->size; pos++){
    if (proc->prio <= q->proc[pos]->prio) break;
  }
    // Shift other proc , leave space for new proc
  for (int i = q->size; i > pos; i--) {
    q->proc[i] = q->proc[i-1];
  }

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

3.1.2 sched.c

```
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 = MAX_PRIO - 1; i >= 0; i--) {
    if (mlq_ready_queue[i].size != 0)
    {
      proc = dequeue(&mlq_ready_queue[i]);
      break;
    }
  }
    // If no proc in mlq_read_queue
  if(i == -1)
  {
        // Put back proc from run_queue to mlq_ready_queue if have any
    for(int j=0; j < run_queue.size; j++){
      enqueue(&mlq_ready_queue[run_queue.proc[j]->prio],run_queue.proc[j]);
    }
    run_queue.size = 0;

        // dequque proc after put back if have any
    for(int k = MAX_PRIO - 1; k >= 0; k--)
    {
      if(mlq_ready_queue[k].size != 0)
      {
        proc = dequeue(&mlq_ready_queue[k]);
        break;
      }
    }
  }
  pthread_mutex_unlock(&queue_lock);
  return proc;
}
```

3.2 Result

Here is our output of sched.c

```
------ 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
Time slot   4
  Loaded a process at input/proc/s1, PID: 2 PRIO: 3
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  2
Time slot  10
Time slot  11
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot  12
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  1
Time slot  13
Time slot  14
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  15
Time slot  16
  CPU 0: Put process  1 to run queue
  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
NOTE: Read file output/sched_0 to verify your result
```

```
------ SCHEDULING TEST 1 -------------------------------------------
./os sched_1
Time slot   0
```

PROCESS		p1				p2							p1										
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

Hình 1:

```
  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
Time slot   4
  Loaded a process at input/proc/s1, PID: 2 PRIO: 2
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: 2
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: 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  4
Time slot  12
Time slot  13
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  14
Time slot  15
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  16
Time slot  17
  CPU 0: Put process  4 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  4
Time slot  20
  CPU 0: Processed  4 has finished
  CPU 0: Dispatched process  2
Time slot  21
Time slot  22
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
Time slot  23
Time slot  24
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  25
Time slot  26
```

```
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
Time slot  27
Time slot  28
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  29
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  3
Time slot  30
Time slot  31
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot  32
Time slot  33
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot  34
Time slot  35
  CPU 0: Processed  3 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
NOTE: Read file output/sched_1 to verify your result
```

PROCESS		p1				p2		p3		p4											p2		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

PROCESS		p2		p3		p2	p3						p1										
TIME SLOT	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45

Hình 2:

3.3 Question

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

In multilevel feedback scheduling, the processes are allowed to move in between the queues, the idea behind is to separate the processes with different CPU – burst characteristics. If any process uses too much CPU then it'll be moved to the lower priority queue and if a process waiting too much for the CPU is moved to the high priority queue this prevents starvation. Multilevel feedback queue is the most general scheme and also the most complex.
Advantages of MLFQ compare to another scheduling algorithms:

- MLFQ allows different processes to move between different queues while MLQ processes are assigned permanently into queues.
- It prevents starvation.
- In MLQ priority is fixed while in MLFQ, priority can be changed during runtime as process are allowed to move between queues. There for, lower the scheduling overhead and more flexible.

4 Memory Management

4.1 Implementation

4.1.1 Mapping from virtual address to physical address

```
#define ADDRESS_SIZE  20
#define OFFSET_LEN    10
#define FIRST_LV_LEN
#define SECOND_LV_LEN 5
#define SEGMENT_LEN      FIRST_LV_LEN
#define PAGE_LEN         SECOND_LV_LEN

#define NUM_PAGES (1 << (ADDRESS_SIZE - OFFSET_LEN))
#define PAGE_SIZE (1 << OFFSET_LEN)
```

As can be seen that , we define that 20 bits are used to represent the address, the first 5 bits encode the segment length and mean the next 5 bits represent page length and the last 10 bits denote the offset.
Furthermore, we will complete 2 function get_trans_table() and translate() to make the transition from the virtual address of a process to the address.

get_trans_table(): Finding a translate table with a segment index for a process.

```
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;
}
```

translate(): use get_trans_table to convert from virtual address to physical address.

```
static int translate(
  addr_t virtual_addr,   // Given virtual address
  addr_t *physical_addr, // Physical addresqs 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
    ;
      return 1;
    }
  }
  return 0;
}
```

4.1.2 Allocate and free memory

The idea is that we must check if there is enough free memory in both physical and virtual memory space, by comparing the number of free pages with the number of needed pages, and ensure that the size of virtual memory does not expand beyond the maximum size of the RAM. First, we will loop through every page in RAM to find available pages to allocate. To do that we need some new variables v_i, p_i, $prev\_v\_i$ and $num\_allocated\_pages$ to keep track of the current physical index, the current virtual index, the physical index of the last allocated page, and the total number of allocated pages. If finding a free pages, update the *proc*, index and the next of the previous allocated page.Then we update the corresponding page table by adding a new row. If the table is not found, we simply creating a new one. And after a successful allocation, we increment v_i to allocate the next page until $num\_allocated\_pages$ is equal to the desired number.

For the $free\_memory()$ we found the physical page index corresponding to the given virtual address by using translate() function and shift right operator, and store it in index variable and this is the index of the first page that we need to free. To delete, we set $\_mem\_stat[index].proc = 0$ and delete page from the page table. If the table is empty, we also delete it. After finishing deleting the page, we go to the next

4.1.2.a Memory allocation

alloc_mem(): We allocate memory to a process by updating the '_mem_stat' array, and updating the page table of the process

```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).
   * */
  uint32_t num_free_pages = 0;
  for (int i = 0; i < NUM_PAGES; i++)
  {
    if (_mem_stat[i].proc == 0)
    {
      num_free_pages++;
    }
  }
  if (num_free_pages >= num_pages && proc->bp + num_pages * PAGE_SIZE < RAM_SIZE)
```

```c
    mem_avail = 1;

  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_allocated_pages = 0;
    for (int p_i = 0, v_i = 0, prev_p_i = 0; p_i < NUM_PAGES; p_i++)
    {
      if (_mem_stat[p_i].proc == 0)
      {
        _mem_stat[p_i].proc = proc->pid;
        _mem_stat[p_i].index = v_i;
        if (v_i != 0)
          _mem_stat[prev_p_i].next = p_i;

        addr_t physical_addr = p_i << OFFSET_LEN;
        addr_t seg_idx = get_first_lv(ret_mem + v_i * PAGE_SIZE);
        addr_t page_idx = get_second_lv(ret_mem + v_i * PAGE_SIZE);

        struct trans_table_t *table = get_trans_table(seg_idx, proc->page_table);
        if (table)
        {
          table->table[table->size].v_index = page_idx;
          table->table[table->size].p_index = physical_addr >> OFFSET_LEN;
          table->size++;
        }
        else
        {
          /* This is the code that allocates a new page table for a new segment. */
          struct page_table_t *t = proc->page_table;
          int n = t->size;
          t->size++;
          t->table[n].next_lv = (struct trans_table_t *)malloc(sizeof(struct trans_table_t));
          t->table[n].next_lv->size++;
          t->table[n].v_index = seg_idx;
          t->table[n].next_lv->table[0].v_index = page_idx;
          t->table[n].next_lv->table[0].p_index = physical_addr >> OFFSET_LEN;
        }

        prev_p_i = p_i;
        v_i++;
        num_allocated_pages++;
        if (num_allocated_pages == num_pages)
        {
          _mem_stat[prev_p_i].next = -1;
          break;
        }
      }
    }
  }
```

```
  pthread_mutex_unlock(&mem_lock);
  // printf("-----Allocated-----------------------------------------\n");
  // dump();
  return ret_mem;
}
```

4.1.2.b Free memory

free_mem(): Free up the allocated memory area

```
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 phys_addr;
  if (translate(address, &phys_addr, proc))
  {
    int v_i = 0;
    int index = phys_addr >> OFFSET_LEN;
    struct page_table_t *t = proc->page_table;
    while (index != -1)
    {
      _mem_stat[index].proc = 0;
      addr_t seg_idx = get_first_lv(address + v_i * PAGE_SIZE);
      addr_t page_idx = get_second_lv(address + v_i * PAGE_SIZE);
      for (int n = 0; n < t->size; n++)
      {
        if (t->table[n].v_index == seg_idx)
        {
          for (int m = 0; m < t->table[n].next_lv->size; m++)
          {
            if (t->table[n].next_lv->table[m].v_index == page_idx)
            {
              int k = 0;
              for (k = m; k < t->table[n].next_lv->size - 1; k++)
              {
                t->table[n].next_lv->table[k].v_index = t->table[n].next_lv->
  table[k + 1].v_index;
                t->table[n].next_lv->table[k].p_index = t->table[n].next_lv->
  table[k + 1].p_index;
              }
              t->table[n].next_lv->table[k].v_index = 0;
              t->table[n].next_lv->table[k].p_index = 0;
              t->table[n].next_lv->size--;
              break;
            }
          }
          if (t->table[n].next_lv->size == 0)
          {
            free(t->table[n].next_lv);
            int m = 0;
            for (m = n; m < t->size - 1; m++)
            {
              t->table[m].v_index = t->table[m + 1].v_index;
```

```
              t->table[m].next_lv = t->table[m + 1].next_lv;
            }
            t->table[m].v_index = 0;
            t->table[m].next_lv = NULL;
            t->size--;
          }
          break;
        }
      }
      v_i++;
      index = _mem_stat[index].next;
    }
  }
  pthread_mutex_unlock(&mem_lock);
  // printf("-----Freed---------------------------------------------\n");
  // dump();
  return 0;
}
```

4.2 Result

Here is our output of mem.c

```
------ 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)
```

To show the status of RAM after each memory allocation and deallocation function as it call in the requirement, we call dump() before exiting alloc_mem() and free_mem(). So this is the output.

```
------ MEMORY MANAGEMENT TEST 0 ------------------------------------
./mem input/proc/m0
-----Allocated-----------------------------------------
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)
-----Allocated-----------------------------------------
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)
-----Freed---------------------------------------------
```

```
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
-----Allocated---------------------------------------
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)
-----Allocated---------------------------------------
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
  003e8: 15
001: 00400-007ff - PID: 01 (idx 001, nxt: -01)
002: 00800-00bff - PID: 01 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)
004: 01000-013ff - PID: 01 (idx 002, nxt: 005)
005: 01400-017ff - PID: 01 (idx 003, nxt: 006)
006: 01800-01bff - PID: 01 (idx 004, nxt: -01)
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
  03814: 66
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
NOTE: Read file output/m0 to verify your result
------ MEMORY MANAGEMENT TEST 1 -----------------------------------
./mem input/proc/m1
-----Allocated---------------------------------------
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)
-----Allocated---------------------------------------
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)
-----Freed------------------------------------------
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
-----Allocated--------------------------------------
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)
-----Allocated--------------------------------------
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)
-----Freed------------------------------------------
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)
-----Freed------------------------------------------
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
-----Freed------------------------------------------
NOTE: Read file output/m1 to verify your result (your implementation should print nothing)
```

4.3 Question

What is the advantage and disadvantage of segmentation with paging?
To reduce the size of the page table in RAM, we use Segmented Paging that combines both
A logical address contains:

- Segment index
- Page table index
- Offset

A physical address includes:

- Physical address
- Offset

We will have segment table and page table. CPU request memory in logical address form. MMU get the page table index based on segment index. When have a table index, MMU lookup on this table and find the actual index in physical memory
Advantages of segmentation with paging:

- It need less memory usage.
- The segment size limited page table size.
- Segment table has only one entry corresponding to one actual segment. (more higher security for the systems)
- No more External Fragmentation.
- It simplifies memory allocation.
- Reduce memory usage compared to the paging technique. Because it can reduce the size of the page table which is store in RAM. A program can be divided into segments and its segments have their page table. Therefore, the page table size will be reduced depending on the number of segments we divided
- 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

Disadvantages of segmentation with paging:

- There is a chance that it will cause Internal Fragmentation..
- The complexity leve is much more higher than paging
- There is internal fragmentation. This problem will occur when the memory required is not divisible by the page size. That is, a frame in RAM will not be used totally. For example, the page size is 5 byte and the program wants to allocate 11 bytes, there will be 3 frames allocated in the RAM and there is a frame that just needs 1 byte of it
- More complex than the paging technique because it requires the implementation of a page table and segment table

- Increase the memory access time because it has to access the segment table and the page table to reach the actual memory it needs
- The page table must be contiguous in the memory because we access the frame in RAM through the offset of the base address of the page table. This may lead to external fragmentation

5 PUT IT ALL TOGETHER

Compiling the whole source code:

```
------ 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)
------ SCHEDULING TEST 0 -----------------------------------------
./os sched_0
Time slot   0
  Loaded a process at input/proc/s0, PID: 1 PRIO: 0
  CPU 0: Dispatched process  1
Time slot   1
Time slot   2
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   3
  Loaded a process at input/proc/s1, PID: 2 PRIO: 3
Time slot   4
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  2
Time slot   5
Time slot   6
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot   7
Time slot   8
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot   9
Time slot  10
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  2
Time slot  11
  CPU 0: Processed  2 has finished
  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
Time slot  17
```

```
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  18
Time slot  19
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  20
Time slot  21
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot  22
  CPU 0: Processed  1 has finished
  CPU 0 stopped
NOTE: Read file output/sched_0 to verify your result
------ SCHEDULING TEST 1 -------------------------------------------
./os sched_1
  Loaded a process at input/proc/s0, PID: 1 PRIO: 1
Time slot   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
Time slot   4
  Loaded a process at input/proc/s1, PID: 2 PRIO: 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 PRIO: 2
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: 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  4
Time slot  12
Time slot  13
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  14
Time slot  15
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  16
Time slot  17
  CPU 0: Put process  4 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  4
Time slot  20
  CPU 0: Processed  4 has finished
  CPU 0: Dispatched process  2
```

```
Time slot  21
Time slot  22
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
Time slot  23
Time slot  24
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  25
Time slot  26
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
Time slot  27
Time slot  28
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  29
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  3
Time slot  30
Time slot  31
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot  32
Time slot  33
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  3
Time slot  34
Time slot  35
  CPU 0: Processed  3 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
NOTE: Read file output/sched_1 to verify your result
----- OS TEST 0 ----------------------------------------------------
./os os_0
  Loaded a process at input/proc/p0, PID: 1 PRIO: 2
Time slot   0
  CPU 1: Dispatched process  1
Time slot   1
  Loaded a process at input/proc/p0, PID: 2 PRIO: 0
```

```
Time slot   2
  CPU 0: Dispatched process  2
  Loaded a process at input/proc/p0, PID: 3 PRIO: 0
Time slot   3
  Loaded a process at input/proc/p0, PID: 4 PRIO: 0
Time slot   4
Time slot   5
Time slot   6
  CPU 1: Put process  1 to run queue
  CPU 1: Dispatched process  1
Time slot   7
Time slot   8
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  3
Time slot   9
Time slot  10
  CPU 1: Processed  1 has finished
  CPU 1: Dispatched process  4
Time slot  11
Time slot  12
Time slot  13
Time slot  14
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  15
Time slot  16
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  3
Time slot  17
Time slot  18
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  4
Time slot  19
Time slot  20
  CPU 1: Processed  3 has finished
  CPU 1 stopped
Time slot  21
Time slot  22
  CPU 0: Processed  4 has finished
  CPU 0 stopped
NOTE: Read file output/os_0 to verify your result
----- OS TEST 1 ----------------------------------------------------
./os os_1
Time slot   0
  Loaded a process at input/proc/p0, PID: 1 PRIO: 2
Time slot   1
  CPU 2: Dispatched process  1
Time slot   2
  Loaded a process at input/proc/p0, PID: 2 PRIO: 0
Time slot   3
  CPU 2: Put process  1 to run queue
  CPU 2: Dispatched process  1
  CPU 1: Dispatched process  2
  Loaded a process at input/proc/p0, PID: 3 PRIO: 0
  CPU 0: Dispatched process  3
Time slot   4
  Loaded a process at input/proc/p0, PID: 4 PRIO: 0
  CPU 3: Dispatched process  4
Time slot   5
  CPU 2: Put process  1 to run queue
  CPU 1: Put process  2 to run queue
  CPU 0: Put process  3 to run queue
```

```
  CPU 2: Dispatched process  1
  CPU 0: Dispatched process  3
  CPU 1: Dispatched process  2
  Loaded a process at input/proc/p0, PID: 5 PRIO: 0
Time slot   6
  Loaded a process at input/proc/p0, PID: 6 PRIO: 0
  CPU 3: Put process  4 to run queue
Time slot   7
  CPU 3: Dispatched process  5
  CPU 2: Put process  1 to run queue
  CPU 1: Put process  2 to run queue
  CPU 0: Put process  3 to run queue
  CPU 2: Dispatched process  1
  CPU 0: Dispatched process  6
  CPU 1: Dispatched process  4
  Loaded a process at input/proc/p0, PID: 7 PRIO: 0
Time slot   8
  Loaded a process at input/proc/p0, PID: 8 PRIO: 0
  CPU 3: Put process  5 to run queue
Time slot   9
  CPU 3: Dispatched process  3
  CPU 2: Put process  1 to run queue
  CPU 1: Put process  4 to run queue
  CPU 0: Put process  6 to run queue
  CPU 2: Dispatched process  1
  CPU 1: Dispatched process  2
  CPU 0: Dispatched process  7
Time slot  10
  CPU 3: Put process  3 to run queue
Time slot  11
  CPU 3: Dispatched process  8
  CPU 2: Processed  1 has finished
  CPU 1: Put process  2 to run queue
  CPU 2: Dispatched process  5
  CPU 0: Put process  7 to run queue
  CPU 1: Dispatched process  4
  CPU 0: Dispatched process  6
Time slot  12
  CPU 3: Put process  8 to run queue
Time slot  13
  CPU 3: Dispatched process  3
  CPU 1: Put process  4 to run queue
  CPU 2: Put process  5 to run queue
  CPU 0: Put process  6 to run queue
  CPU 1: Dispatched process  2
  CPU 2: Dispatched process  7
  CPU 0: Dispatched process  8
Time slot  14
  CPU 3: Put process  3 to run queue
Time slot  15
  CPU 3: Dispatched process  4
  CPU 2: Put process  7 to run queue
  CPU 1: Put process  2 to run queue
  CPU 0: Put process  8 to run queue
  CPU 2: Dispatched process  5
  CPU 0: Dispatched process  6
  CPU 1: Dispatched process  3
Time slot  16
  CPU 3: Put process  4 to run queue
Time slot  17
  CPU 3: Dispatched process  7
  CPU 1: Processed  3 has finished
```

```
  CPU 2: Put process  5 to run queue
  CPU 0: Put process  6 to run queue
  CPU 1: Dispatched process  8
  CPU 2: Dispatched process  2
  CPU 0: Dispatched process  4
Time slot  18
  CPU 3: Put process  7 to run queue
  CPU 3: Dispatched process  6
  CPU 2: Processed  2 has finished
  CPU 1: Put process  8 to run queue
Time slot  19
  CPU 0: Processed  4 has finished
  CPU 1: Dispatched process  5
  CPU 0: Dispatched process  7
  CPU 2: Dispatched process  8
Time slot  20
  CPU 3: Put process  6 to run queue
  CPU 3: Dispatched process  6
  CPU 2: Put process  8 to run queue
Time slot  21
  CPU 2: Dispatched process  8
  CPU 1: Put process  5 to run queue
  CPU 1: Dispatched process  5
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  22
  CPU 3: Processed  6 has finished
  CPU 3 stopped
  CPU 2: Processed  8 has finished
Time slot  23
  CPU 2 stopped
  CPU 1: Processed  5 has finished
  CPU 1 stopped
  CPU 0: Processed  7 has finished
  CPU 0 stopped
NOTE: Read file output/os_1 to verify your result
```

And here is our make os output

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

```
  CPU 1: Dispatched process  4
Time slot  11
Time slot  12
Time slot  13
Time slot  14
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  2
Time slot  15
Time slot  16
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  3
Time slot  17
Time slot  18
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  4
Time slot  19
Time slot  20
  CPU 1: Processed  3 has finished
  CPU 1 stopped
Time slot  21
Time slot  22
  CPU 0: Processed  4 has finished
  CPU 0 stopped
NOTE: Read file output/os_0 to verify your result
----- OS TEST 1 ----------------------------------------------------
./os os_1
Time slot   0
Time slot   1
  Loaded a process at input/proc/p0, PID: 1 PRIO: 2
  CPU 0: Dispatched process  1
Time slot   2
  Loaded a process at input/proc/p0, PID: 2 PRIO: 0
  CPU 2: Dispatched process  2
Time slot   3
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
  Loaded a process at input/proc/p0, PID: 3 PRIO: 0
Time slot   4
  CPU 3: Dispatched process  3
  Loaded a process at input/proc/p0, PID: 4 PRIO: 0
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  4
Time slot   5
  CPU 1: Dispatched process  2
  CPU 0: Put process  1 to run queue
  Loaded a process at input/proc/p0, PID: 5 PRIO: 0
  CPU 0: Dispatched process  1
  CPU 3: Put process  3 to run queue
  CPU 3: Dispatched process  5
Time slot   6
  Loaded a process at input/proc/p0, PID: 6 PRIO: 0
  CPU 2: Put process  4 to run queue
  CPU 2: Dispatched process  3
Time slot   7
  CPU 1: Put process  2 to run queue
  CPU 0: Put process  1 to run queue
  CPU 1: Dispatched process  6
  CPU 0: Dispatched process  1
  Loaded a process at input/proc/p0, PID: 7 PRIO: 0
Time slot   8
  CPU 3: Put process  5 to run queue
  CPU 3: Dispatched process  4
```

```
  CPU 2: Put process  3 to run queue
  CPU 2: Dispatched process  2
  Loaded a process at input/proc/p0, PID: 8 PRIO: 0
Time slot   9
  CPU 1: Put process  6 to run queue
  CPU 1: Dispatched process  7
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
  CPU 3: Put process  4 to run queue
Time slot  10
  CPU 3: Dispatched process  5
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  3
  CPU 1: Put process  7 to run queue
Time slot  11
  CPU 1: Dispatched process  8
  CPU 0: Processed  1 has finished
  CPU 0: Dispatched process  6
  CPU 2: Put process  3 to run queue
Time slot  12
  CPU 2: Dispatched process  4
  CPU 3: Put process  5 to run queue
  CPU 3: Dispatched process  2
Time slot  13
  CPU 1: Put process  8 to run queue
  CPU 0: Put process  6 to run queue
  CPU 1: Dispatched process  7
  CPU 0: Dispatched process  3
  CPU 2: Put process  4 to run queue
  CPU 2: Dispatched process  5
Time slot  14
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  8
  CPU 1: Put process  7 to run queue
Time slot  15
  CPU 1: Dispatched process  6
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  4
  CPU 2: Put process  5 to run queue
  CPU 2: Dispatched process  2
Time slot  16
  CPU 3: Put process  8 to run queue
  CPU 3: Dispatched process  7
Time slot  17
  CPU 1: Put process  6 to run queue
  CPU 0: Put process  4 to run queue
  CPU 1: Dispatched process  3
  CPU 0: Dispatched process  5
  CPU 2: Processed  2 has finished
Time slot  18
  CPU 2: Dispatched process  8
  CPU 3: Put process  7 to run queue
  CPU 3: Dispatched process  6
  CPU 1: Processed  3 has finished
Time slot  19
  CPU 1: Dispatched process  4
  CPU 0: Put process  5 to run queue
  CPU 0: Dispatched process  7
  CPU 2: Put process  8 to run queue
  CPU 2: Dispatched process  5
Time slot  20
  CPU 3: Put process  6 to run queue
```

```
  CPU 3: Dispatched process  8
Time slot  21
  CPU 1: Processed  4 has finished
  CPU 1: Dispatched process  6
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
  CPU 2: Processed  5 has finished
  CPU 2 stopped
Time slot  22
  CPU 3: Put process  8 to run queue
  CPU 3: Dispatched process  8
  CPU 1: Processed  6 has finished
Time slot  23
  CPU 1 stopped
  CPU 0: Processed  7 has finished
  CPU 0 stopped
Time slot  24
  CPU 3: Processed  8 has finished
  CPU 3 stopped
NOTE: Read file output/os_1 to verify your result
```

5.1 Question

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.

Synchronization is important in every operating system. It helps coordinates the execution of processes so that not two concurrent process can access the same shared data. Especially in the multi-process environment when several processes are activated at once and attempt to access the same shared data.
One of the particular example is the **Critical Section problem**. The problem occur when a shared data is accessed and modified by different process or thread in same time, result in data inconsistency, raise error during execution.

For specific, our OS is running under multi-process environment using the **multilevel feedback queue** scheduler system. While many processes are run concurrently, the scheduler use only one **same queues system** to determine which processes is allowed to run on which CPU. Therefore, task for synchronization is to protect the consistent of queues. For example, the queue size only contains 1 left slot while there are two or more processes need to be put in. Without synchronization, they first check for the queue size at the same time and both processes is valid to be put in. However the queue can contain one more process, so the second processes could be missed or error. But with the help of synchronization, no two processes can access queue at the same time, they have to wait for the other to finish their actions (check for queue size and enqueue) so that ensure the thread safe.

Secondly, a problem may occur within the memory. There will be conflicts in getting the allocated memory.

Suppose that two different threads try to allocate the same page to itself, the first thread may find that the page at location 0x00 has not been used and so does the second thread. Then, both threads decided to take that page for allocation. The result is that two processes will record the same memory frame as theirs. The _mem_stat table will only record one of the two threads as the true owner, causing one process to not have enough space for its memory. There may also be a problem in deallocating memory. For example, thread A tries to set the flag to indicate that the page is free. Then, thread B sees that page is free, so it gets that page to be allocated. However, thread A continues its job, which is to remove the unused entries in the page table and segment table of that page. Therefore, thread B will fail when it tries to reference that page.

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

6 CONCLUSION

This assignment provides us with knowledge about how a simple operating system works by implementing Scheduler and Virtual Memory Engine(VME) and enhancing our teamwork skills. The combination of Scheduler and VME makes fundamental concepts of scheduling, synchronization, and memory management, allowing the CPU to run Multilevel Queue (MLQ) CPU Scheduling.

References

[1] *Operating System Concepts*, Abraham Silberschatz, Greg Gagne, Peter B. Galvin, 10th Edition, John Wiley & Sons, 2018. ISBN1119439256, 9781119439257, 976 pages.

[2] *Operating Systems: Three Easy Pieces*, Remzi H. Arpaci.Dusseau, Andrea C. Arpaci-Dusseau, CreateSpace Independent Publishing Platform, 2018. ISBN198508659X, 9781985086593, 714 pages.