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

OPERATING SYSTEMS (CO2018)

Assignment

Simple Operating System

Advisor: Nguyễn Lê Duy Lai

Students: Lương Duy Hưng (Leader) (CC04) - 1952747
Trần Nguyễn Phước Nhân (CC04) - 1952893
Trần Quốc Việt (CC04) - 1953097

HO CHI MINH CITY, MAY 2021

Contents

1 Member list & Workload

No.	Fullname	Student ID	Problems	% of work
1	Lương Duy Hưng (Leader)	1952747	- Scheduling - Memory Management - Result verification	33.33%
2	Trần Nguyễn Phước Nhân	1952893	- Memory Management - Question answering - Result verification	33.33%
3	Trần Quốc Việt	1953097	- LaTeX editing - Memory Managementt - Question answering	33.33%

2 Scheduler

2.1 Priority Feedback Queue

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

Advantages of Priority Feedback Queue

- Inherits the idea of Round-Robin algorithm with a quantum time, prevent CPU wastage and deadlock.
- Using 2 queues : run_queue and ready_queue intimately to create flexibility between many processes.
- Apart from process' priority, the fairness of every processes is guaranteed. For example at a specific time step t, the execution of a process only based on its priority. If there is an upcoming process with eventually lower priority, current process is not allowed to interfere and will be pushed to wait in run_queue until the ready_queue is free.

2.2 Gantt diagram of outputs

Question : Draw Gantt diagram describing how processes are executed by the CPU Below are the Gantt diagram visualizing the result of 2 sample test cases for the scheduling section from output/sched_0 and output/sched_1.

Sched 0 :

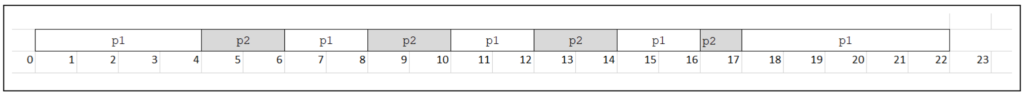

Figure 1: Gantt diagram showing execution of processes by CPU - sched_0

Sched 1 :

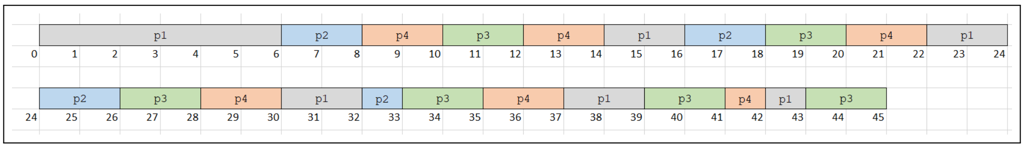

Figure 2: Gantt diagram showing execution of processes by CPU - sched_1

2.3 Implementation

To implement the scheduler, we will to implement 3 functions in the 2 following files *src/queue.c* and *src/sched.c*.

2.3.1 Priority Queue

In *queue.c*, for *enqueue()*, we pushed the new process at the back of the array (if possible). For *dequeue()*, we must search for and take the process with highest priority and earliest out from the queue; additionally, we have to update the content of the queue after removing its element.

In our work, we use heap data structure to represent the priority queue as there is a prominent advantage in time complexity compare to normal array data structure :

- Heap : O(logN) (Enqueue : O(logN), Dequeue : O(logN).
- Array : O(N) (Enqueue : O(1), Dequeue : O(N)).

Below is the implementation.

```c
void push(struct queue_t * q, int idx){
  for( int i = idx+1 ; i< q->size - 1; i++){
    q->proc[i-1] = q->proc[i];
  }
}


void swap(struct pcb_t  * a, struct pcb_t * b) {
  struct pcb_t  temp  = *b;
  *b = *a;
  *a = temp;
}

void heapify(struct queue_t * q, int sizee, int i) {
  if (sizee == 1) {
    //printf("Single element in the heap");
  } else {
    // Find the largest among root, left child and right child
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < sizee
        && q->proc[l]->priority < q->proc[largest]->priority)
      largest = l;
    if (r < sizee && q->proc[r]->priority < q->proc[largest]->priority)
      largest = r;

    // Swap and continue heapifying if root is not largest
    if (largest != i) {
      swap(q->proc[i], q->proc[largest]);
      heapify(q, sizee, largest);
    }
  }
}

void enqueue(struct queue_t * q, struct pcb_t * proc) {
  if (q->size == MAX_QUEUE_SIZE) return;
  q->proc[q->size++] = proc;

}

struct pcb_t * dequeue(struct queue_t * q) {
  if (q->size == 0) return NULL;
  int i = 0, j;
  for (j = 1; j < q->size; j++) {
```

```
    if (q->proc[j]->priority < q->proc[i]->priority) {
      i = j;
    }
  }
  struct pcb_t * res = q->proc[i];
  for (j = i+1; j < q->size; j++) {
    q->proc[j-1] = q->proc[j];
  }
  q->size--;

  return res;
}
```

2.3.2 Scheduler

Scheduler's work is to manage ready queue and run queue. In the file *sched.c*, we only have to implement the function *get_proc()* to return a process from the ready _queue. If the ready_queue is empty, we have to move all processes from the run_queue back to the ready_queue. Otherwise, return the process with highest priority using the aforementioned *dequeue()* function.

Below is the implementation of *get_proc()* for scheduler.

```
struct pcb_t * get_proc(void) {
  struct pcb_t * proc = NULL;
  /*TODO: get a process from [ready_queue]. If [ready queue]
   * is empty, push all processes in [run_queue] back to
   * [ready_queue] and return the highest priority one.
   * Remember to use lock to protect the queue.
     * */
  pthread_mutex_lock(&queue_lock);
  if(empty(&ready_queue)){
    while(!empty(&run_queue)){
      enqueue(&ready_queue, dequeue(&run_queue));
    }
  }
  proc = dequeue(&ready_queue);
  pthread_mutex_unlock(&queue_lock);
  return proc;
}
```

3 Memory Management

3.1 Segmentation with Paging

Question : What is the advantage and disadvantage of segmentation with paging.

Advantages of Segmentation with Paging

- Greatly utilizing memory arrangement.
- Improve the disadvantages of Paging : External Fragmentation.

Disadvantages of Segmentation with Paging

- Internal fragmentation still exists.

3.2 Status of RAM

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

Below is the result from the output log file from 2 sample tests showing content of RAM after memory allocation and deallocation. **Note :** During the work, we found out that the input file m0 for Memory test did not give out the correct result, so we have modified the *m0* file, below is the modified content.

m0 :

```
1 7
alloc 13535 0
alloc 1568 1
free 0
alloc 1386 2
alloc 4564 4
write 100 1 20
write 20 2 1000
```

Test 0 :

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

```
================ Allocation ================
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)
================ Allocation ================
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)

================ Final Result ================
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)
```

Test 1 :

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

================ Final Result ================
```

3.3 Implementation

3.3.1 Get page table

In this assignment, each address is represented by 20 bits, first 5 bits is segment index, next 5 bits is page index and the last 10 bits is offset. This function receive 5 bits of segment *index* and the segment table *seg_table* and return the corresponding *page_table*.

```c
static struct page_table_t * get_page_table(
    addr_t index,    // Segment level index
    struct seg_table_t * seg_table) { // first level table

  if (seg_table == NULL) return NULL;
  int i;
  for (i = 0; i < seg_table->size; i++) {
    if(seg_table->table[i].v_index == index)
      return seg_table->table[i].pages;
  }
  return NULL;
}
```

3.3.2 Address translation

As each address is 20 bits mentioned above, to create physical address, we need to concatenate 10 first bits with 10 offset bits. Each *page_table_t* contains that first 10 bits, so we only need to shift left that 10 bits and do the or operator (|) these bits together.

```
static int translate(
    addr_t virtual_addr,   // Given virtual address
    addr_t * physical_addr, // Physical address to be returned
    struct pcb_t * proc) {  // Process uses given virtual address

  /* Offset of the virtual address */
  addr_t offset = get_offset(virtual_addr);
  /* The first layer index */
  addr_t first_lv = get_first_lv(virtual_addr);
  /* The second layer index */
  addr_t second_lv = get_second_lv(virtual_addr);

  /* Search in the first level */
  struct page_table_t * page_table = NULL;
  page_table = get_page_table(first_lv, proc->seg_table);
  if(page_table == NULL) {
    return 0;
  }

  int i;
  for (i = 0; i < page_table->size; i++) {
    if (page_table->table[i].v_index == second_lv) {
      /* TODO: Concatenate the offset of the virtual address
       * to [p_index] field of page_table->table[i] to
       * produce the correct physical address and save it to
       * [*physical_addr]  */
      addr_t p_index = page_table->table[i].p_index; // physical page index
      * physical_addr = (p_index << OFFSET_LEN) | (offset);
      return 1;
    }
  }
  return 0;
}
```

3.3.3 Memory allocation

3.3.3.a Check if memory is available

- We must check if memory is available to allocate in this step in both physical and logical one.
- In logical region, we check whether the break pointer exceed the permitted limit.
- In physical region, we check available pages, if it is enough then the allocation step is ready.

```
int is_any_available(int num_pages, struct pcb_t * proc) {
  /**
   * 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.
```

```
   * Set 1 to [mem_avail] if possible.
   */

  // Check physical space
  int i = 0;
  int free_pages = 0; // count free pages
  for (i = 0; i < NUM_PAGES; i++) {
    if (_mem_stat[i].proc == 0) {
      free_pages++;
      if (free_pages >= num_pages) break;
    }
  }

  if (free_pages < num_pages) return 0;

  // Check virtual space
  if (proc->bp + num_pages*PAGE_SIZE >= RAM_SIZE){
    printf("procbp >= \n");
    return 0;
  }

  return 1;
}
```

3.3.3.b Alloc memory

Once the above steps are adequately checked, we allocate memory for the process.

```
void allocate_available_memory(int ret_mem, int num_pages, struct pcb_t * proc) {
  /**
   * Update status of physical pages which will be allocated to [proc] in _mem_stat.
   * Tasks to do:
   *  + Update [proc], [index], and [next] field in _mem_stat
   *  + Add entries to segment table page tables of [proc]
   *    to ensure accesses to allocated memory slot is valid.
   */

  int used_pages = 0; // count allocated pages
  int index_of_last_used_page = -1; // use for update field [next] of last allocated page
  int i;
  for (i = 0; i < NUM_PAGES; i++) {
    if (!_mem_stat[i].proc) { // page is unused
      // assign proc
      _mem_stat[i].proc = proc->pid; // the page is used by process [proc]
      _mem_stat[i].index = used_pages; // index in list of allocated pages

      //set index
      if (index_of_last_used_page >= 0) { // not initial page, update last page
        _mem_stat[index_of_last_used_page].next = i;
      }
      index_of_last_used_page = i; // update last page

      //Find or Create virtual page table
      addr_t v_address = ret_mem + used_pages * PAGE_SIZE; // virtual address of this page
      addr_t v_segment = get_first_lv(v_address);
```

```
      struct page_table_t * v_page_table = get_page_table(v_segment, proc->seg_table);
      // first level set up
      if (v_page_table == NULL) {
        int idx = proc->seg_table->size;
        proc->seg_table->table[idx].v_index = v_segment;// setting up v_segment
        v_page_table= proc->seg_table->table[idx].pages = (struct page_table_t*)malloc(sizeof(struct page_table_t));// malloc new mem
        proc->seg_table->size++;
      }

      // second level set up
      int idx = v_page_table->size++;
      v_page_table->table[idx].v_index = get_second_lv(v_address);
      v_page_table->table[idx].p_index = i; // format of i is 10 bit segment and page in address
      used_pages++;
      if (used_pages == num_pages) {
        _mem_stat[i].next = -1; // check if last page in list
        break;
      }
    }
  }
}
```

3.3.4 Memory deallocation

3.3.4.a Deallocate and update logical memory

In this step, we translate the logical address from process to physical one, then base on *next* value of mem to update the sequence of corresponding address. Finally, base on number of deleted pages in physical address region, we search for respective segment and page to update page table. If this page table is empty then we delete it.

```
void free_mem_break_point(struct pcb_t * proc) {
  while (proc->bp >= PAGE_SIZE) {
    addr_t last_addr = proc->bp - PAGE_SIZE;
    addr_t last_segment = get_first_lv(last_addr);
    struct page_table_t * page_table = get_page_table(last_segment, proc->seg_table);
    addr_t last_page = get_second_lv(last_addr);
    if (page_table == NULL) return;
    while (last_page >= 0) {
      int i;
      for (i = 0; i < page_table->size; i++) {
        if (page_table->table[i].v_index == last_page) {
          proc->bp -= PAGE_SIZE;
          last_page--;
          break;
        }
      }
      if (i == page_table->size) break;
    }
    if (last_page >= 0) break;
  }
}

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);
 int num_pages = 0;
 addr_t physical_addr = 0;
 addr_t virtual_adrr = address;
 int i;
 if (translate(address, &physical_addr, proc)) //*check validity of address
 {
   addr_t physical_page = physical_addr >> OFFSET_LEN; //physical

   while (physical_page != -1) //physical page not null
   {
     _mem_stat[physical_page].proc = 0;                    // empty proc in physical
     addr_t segIndex = get_first_lv(virtual_adrr);             // to segtable
     for (i = 0; i < proc->seg_table->table[segIndex].pages->size; i++) // go through all segtable to find identical page table
     {
       if (proc->seg_table->table[segIndex].pages->table[i].p_index == physical_page)
       {
         proc->seg_table->table[segIndex].pages->table[i].v_index = 0; // free virtual addr
         proc->seg_table->table[segIndex].pages->table[i].p_index = 0; // free [seg] + [page]
       }
     }
     physical_page = _mem_stat[physical_page].next;
     virtual_adrr += PAGE_SIZE;
     num_pages += 1;
     proc->bp -= PAGE_SIZE;
   }

 }

 if (MEM_TEST) {
   printf("==========  Deallocation  ==========\n");
   dump();
 }


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

3.3.4.b Update break pointer

Once the last block in logical region is deleted, we start update the bp, then reverse iterate until we reach the using region.

```
void free_mem_break_point(struct pcb_t * proc) {
  while (proc->bp >= PAGE_SIZE) {
    addr_t last_addr = proc->bp - PAGE_SIZE;
    addr_t last_segment = get_first_lv(last_addr);
    struct page_table_t * page_table = get_page_table(last_segment, proc->
        seg_table);
    addr_t last_page = get_second_lv(last_addr);
    if (page_table == NULL) return;
    while (last_page >= 0) {
      int i;
      for (i = 0; i < page_table->size; i++) {
        if (page_table->table[i].v_index == last_page) {
          proc->bp -= PAGE_SIZE;
          last_page--;
          break;
        }
      }
      if (i == page_table->size) break;
    }
    if (last_page >= 0) break;
  }
}
```

4 Overall Operating System simulation

After combining the content of scheduler and memory management, we perform *make all* and come to the final result of a simple operating system. Below is a comprehensive look on the content of scheduler and memory from the tests. Due to the nature of scheduler, result of memory content would be affected as well.

os_0 :

```
------ OS TEST 0 ------------------------------------------------------------
./os os_0
Time slot   0
  Loaded a process at input/proc/p0, PID: 1
  CPU 0: Dispatched process  1
Time slot   1
Time slot   2
  Loaded a process at input/proc/p1, PID: 2
Time slot   3
  CPU 1: Dispatched process  2
  Loaded a process at input/proc/p1, PID: 3
Time slot   4
  Loaded a process at input/proc/p1, PID: 4
Time slot   5
Time slot   6
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  3
Time slot   7
Time slot   8
Time slot   9
  CPU 1: Put process  2 to run queue
  CPU 1: Dispatched process  4
Time slot  10
Time slot  11
```

```
Time slot  12
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  1
Time slot  13
Time slot  14
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  2
Time slot  15
Time slot  16
  CPU 0: Processed  1 has finished
  CPU 0: Dispatched process  3
Time slot  17
Time slot  18
  CPU 1: Processed  2 has finished
  CPU 1: Dispatched process  4
Time slot  19
Time slot  20
Time slot  21
  CPU 0: Processed  3 has finished
  CPU 0 stopped
Time slot  22
Time slot  23
  CPU 1: Processed  4 has finished
  CPU 1 stopped

MEMORY CONTENT:
000: 00000-003ff - PID: 02 (idx 000, nxt: 001)
001: 00400-007ff - PID: 02 (idx 001, nxt: 007)
002: 00800-00bff - PID: 03 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 03 (idx 001, nxt: 004)
004: 01000-013ff - PID: 03 (idx 002, nxt: 005)
005: 01400-017ff - PID: 03 (idx 003, nxt: -01)
  01414: 64
006: 01800-01bff - PID: 04 (idx 000, nxt: 028)
007: 01c00-01fff - PID: 02 (idx 002, nxt: 008)
  01de7: 0a
008: 02000-023ff - PID: 02 (idx 003, nxt: 009)
009: 02400-027ff - PID: 02 (idx 004, nxt: -01)
010: 02800-02bff - PID: 01 (idx 000, nxt: -01)
011: 02c00-02fff - PID: 02 (idx 000, nxt: 012)
012: 03000-033ff - PID: 02 (idx 001, nxt: 013)
013: 03400-037ff - PID: 02 (idx 002, nxt: 014)
014: 03800-03bff - PID: 02 (idx 003, nxt: -01)
015: 03c00-03fff - PID: 03 (idx 000, nxt: 016)
016: 04000-043ff - PID: 03 (idx 001, nxt: 017)
017: 04400-047ff - PID: 03 (idx 002, nxt: 018)
  045e7: 0a
018: 04800-04bff - PID: 03 (idx 003, nxt: 019)
019: 04c00-04fff - PID: 03 (idx 004, nxt: -01)
020: 05000-053ff - PID: 04 (idx 000, nxt: 021)
021: 05400-057ff - PID: 04 (idx 001, nxt: 022)
022: 05800-05bff - PID: 04 (idx 002, nxt: 023)
023: 05c00-05fff - PID: 04 (idx 003, nxt: -01)
024: 06000-063ff - PID: 02 (idx 000, nxt: 025)
025: 06400-067ff - PID: 02 (idx 001, nxt: 026)
026: 06800-06bff - PID: 02 (idx 002, nxt: 027)
027: 06c00-06fff - PID: 02 (idx 003, nxt: -01)
028: 07000-073ff - PID: 04 (idx 001, nxt: 029)
029: 07400-077ff - PID: 04 (idx 002, nxt: 030)
030: 07800-07bff - PID: 04 (idx 003, nxt: -01)
057: 0e400-0e7ff - PID: 04 (idx 000, nxt: 058)
058: 0e800-0ebff - PID: 04 (idx 001, nxt: 059)
```

```
059: 0ec00-0efff - PID: 04 (idx 002, nxt: 060)
  0ede7: 0a
060: 0f000-0f3ff - PID: 04 (idx 003, nxt: 061)
061: 0f400-0f7ff - PID: 04 (idx 004, nxt: -01)
062: 0f800-0fbff - PID: 03 (idx 000, nxt: 063)
063: 0fc00-0ffff - PID: 03 (idx 001, nxt: 064)
064: 10000-103ff - PID: 03 (idx 002, nxt: 065)
065: 10400-107ff - PID: 03 (idx 003, nxt: -01)
```

os_1 :

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

```
Time slot  11
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  7
  CPU 2: Put process  6 to run queue
  CPU 2: Dispatched process  4
  CPU 1: Put process  5 to run queue
  CPU 1: Dispatched process  3
  CPU 0: Processed  1 has finished
  CPU 0: Dispatched process  6
Time slot  12
Time slot  13
  CPU 3: Put process  7 to run queue
  CPU 3: Dispatched process  5
  CPU 2: Put process  4 to run queue
  CPU 2: Dispatched process  2
  CPU 1: Processed  3 has finished
  CPU 1: Dispatched process  7
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  4
Time slot  14
Time slot  15
  Loaded a process at input/proc/s1, PID: 8
  CPU 3: Put process  5 to run queue
  CPU 3: Dispatched process  8
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  6
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  5
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  2
Time slot  16
  CPU 1: Processed  5 has finished
  CPU 1: Dispatched process  7
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  4
Time slot  17
  CPU 3: Put process  8 to run queue
  CPU 3: Dispatched process  8
  CPU 2: Put process  6 to run queue
  CPU 2: Dispatched process  6
Time slot  18
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  4
Time slot  19
  CPU 3: Put process  8 to run queue
  CPU 3: Dispatched process  8
  CPU 2: Put process  6 to run queue
  CPU 2: Dispatched process  6
Time slot  20
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
  CPU 0: Processed  4 has finished
  CPU 0 stopped
Time slot  21
  CPU 3: Put process  8 to run queue
  CPU 3: Dispatched process  8
  CPU 2: Processed  6 has finished
  CPU 2 stopped
Time slot  22
  CPU 3: Processed  8 has finished
```

```
  CPU 3 stopped
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
Time slot  23
Time slot  24
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
Time slot  25
Time slot  26
  CPU 1: Put process  7 to run queue
  CPU 1: Dispatched process  7
Time slot  27
  CPU 1: Processed  7 has finished
  CPU 1 stopped
Time slot  28

MEMORY CONTENT:
000: 00000-003ff - PID: 05 (idx 000, nxt: 001)
001: 00400-007ff - PID: 05 (idx 001, nxt: 009)
002: 00800-00bff - PID: 06 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 06 (idx 001, nxt: 004)
004: 01000-013ff - PID: 06 (idx 002, nxt: 005)
  011e7: 0a
005: 01400-017ff - PID: 06 (idx 003, nxt: 006)
006: 01800-01bff - PID: 06 (idx 004, nxt: -01)
007: 01c00-01fff - PID: 05 (idx 000, nxt: 008)
  01fe8: 15
008: 02000-023ff - PID: 05 (idx 001, nxt: -01)
009: 02400-027ff - PID: 05 (idx 002, nxt: 010)
010: 02800-02bff - PID: 05 (idx 003, nxt: 011)
011: 02c00-02fff - PID: 05 (idx 004, nxt: -01)
012: 03000-033ff - PID: 06 (idx 000, nxt: 013)
013: 03400-037ff - PID: 06 (idx 001, nxt: 014)
014: 03800-03bff - PID: 06 (idx 002, nxt: 015)
015: 03c00-03fff - PID: 06 (idx 003, nxt: -01)
016: 04000-043ff - PID: 06 (idx 000, nxt: 017)
017: 04400-047ff - PID: 06 (idx 001, nxt: 018)
018: 04800-04bff - PID: 06 (idx 002, nxt: 019)
019: 04c00-04fff - PID: 06 (idx 003, nxt: -01)
  04c14: 64
021: 05400-057ff - PID: 01 (idx 000, nxt: -01)
029: 07400-077ff - PID: 05 (idx 000, nxt: 030)
  07414: 66
030: 07800-07bff - PID: 05 (idx 001, nxt: -01)
```

5 Conclusion

In this assignment, our group has successfully carried out the implementation of simulating a simple operating system, visualizing the work of scheduling of processes, demonstrating the content of memory and so far have a good understanding on what we have done.