HO CHI MINH UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

- -

OPERATING SYSTEM

Assignment

Simple Operating System

Lecturer: Tran Viet Toan
Students: 1852619 - Nguyen Trung Nguyen
1852164 - Nguyen Hoang Long

Thanh pho Ho Chi Minh, 5/2021

Contents

List of Figures

1 Scheduler

1.1 Question - Priority Feedback Queue

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

The Priority Feedback Queue (PFQ) algorithm uses the ideas of a number of other algorithms including Priority Scheduling - each process has a priority to execute, Multilevel Queue algorithm - uses multi-level process queues, Round Robin algorithm - uses quantum time for processes to execute.

Specifically, the PFQ algorithm uses two queues, *ready_queue* and *run_queue* with the following meanings:

- *ready_queue*: This queue contains processes which the CPU searches for When it moves to the next slot.
- *run_queue*: This queue contains processes that are waiting to continue executing after running out of their slots without completing their process. Processes in this queue can only continue to the next slot when *ready_queue* is empty.
- Both queues are priority queues, the priority is based on the priority of the process in the queue.

preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

Advantages of using priority feedback queue

- With the idea of RR algorithm with a quantum time interval, creating fairness in execution time between processes, but it still execute in order of priority (using time for more importance process first, avoid waiting time for users).
- Bring the idea of MLQS algorithm but use quantum time to solve the problem of starvation or indefinite blocking.
- It just have 2 main queues, so it is lower scheduling overhead than Multilevel Feedback Queue Scheduling although they use similar algorithm.
- Compare with First Come First Serve (FCFS) and Shortest Job Next (SJN), it is preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

1.2 Result - Gantt Diagrams

REQUIREMENT: Draw Gantt diagram describing how processes are executed by the CPU.

Test 0:

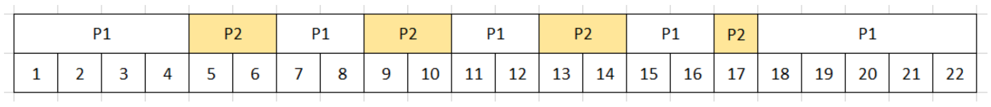

Figure 1: *Grantt diagram for test 0*

Test 1:

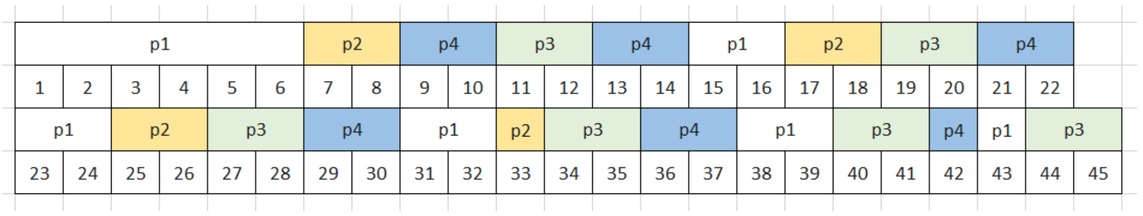

Figure 2: *Grantt diagram for test 1*

1.3 Implementation

1.3.1 Priority Queue

With the $enqueue()$ function, the Priority Queue in this case handles no more than MAX_QUEUE_SIZE processes, so we simply need to use the loop to handle the functionality a priority queue needs.
With the $dequeue()$ function, we find the process with the highest priority to return, and update the state of the queue when deleting an element.

```c
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;
}
```

1.3.2 Scheduler

With get_proc fnuction, we need to check whether ready queue empty or not. If it is empty we need to move all process PCBs from run queue to ready queue. Then, we return the highest priority PCB in ready queue.

```c
struct pcb_t * get_proc(void) {
  struct pcb_t * proc = NULL;
  pthread_mutex_lock(&queue_lock);
  if (empty(&ready_queue)) {
    // move all process is waiting in run_queue back to ready_queue
    while (!empty(&run_queue)) {
      enqueue(&ready_queue, dequeue(&run_queue));
    }
  }
  if (!empty(&ready_queue)) {
    proc = dequeue(&ready_queue);
  }
  pthread_mutex_unlock(&queue_lock);
  return proc;
}
```

2 Memory Management

2.1 Question - Segmentation with Paging

QUESTION:What is the advantage and disadvantage of segmentation with paging?

Advantages of the algorithm

- Save memory, use memory efficiently.
- Have the advantages of paging algorithm:
  Simplify memory allocation.
  Fixed foreign fragmentation.
- Solve the external fragmentation problem of the segmentation algorithm by paging within each segment.

Disadvantages of Algorithm

- Internal fragmentation of the paging algorithm remains.

2.2 Result - Status of RAM

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

In detail, the status of RAM in every steps:

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)
000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
============== Final - dump() ==============
  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)
```

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 - dump() ==============
```

2.3 Implementation

2.3.1 Find paging table from segment

In this assignment, each address is represented by 20 bits, where the first 5 bits are the segment, the next 5 bits are the page, and the last 10 bits are the offset.
This function takes in 5 bits segment *index* segment table *seg_table*, should find the paging table *res* of the corresponding segment in the aforementioned shard table.
Since the segment table *seg_table* is a list of u elements with the structure $(v\_index, page\_table\_t)$, in that *v_index* is 5 bits segment of u elements and *page_table_t* is the corresponding page part table for that segment.
So to find *res*, we just need to browse on this segment table, which u element has *v_index* equal to *index* to find, we return the corresponding *page_table*.
The implementation for the above function show below :

```c
static struct page_table_t * get_page_table(addr_t index, struct seg_table_t * ↩
    seg_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;
}
```

2.3.2 Mapping virtual addresses to physical addresses

Because each address consists of 20 bits with the above organization, so to create a physical address, we take the first 10 bits (segment and page) and connect the last 10 bits (offset). Each *page_table_t* stores elements where *p_index* is the first 10 bits. so to create a physical address, we just need to shift left those 10 bits by 10 bits offset and then or (—) these two strings.
The implementation for the above function show below :

```c
static int translate(addr_t virtual_addr, addr_t * physical_addr, struct pcb_t * proc) ↩
    {
  /* Offset of the virtual address */
  addr_t offset = get_offset(virtual_addr);
  /* The first layer index, find segment virtual */
  addr_t first_lv = get_first_lv(virtual_addr);
  /* The second layer index, find page virtual */
  addr_t second_lv = get_second_lv(virtual_addr);

  /* Search in the first level */
  struct page_table_t * page_table = get_page_table(first_lv, proc->seg_table);
  if (page_table == NULL) return false;
  int i;
  for (i = 0; i < page_table->size; i++) {
    if (page_table->table[i].v_index == second_lv) {
      addr_t p_index = page_table->table[i].p_index; // physical page index
      * physical_addr = (p_index << OFFSET_LEN) | (offset);
      return true;
    }
  }
  return false;
}
```

2.3.3 Memory allocation

2.3.3.1 Check the readiness of memory: In this step we check if the memory is available both on the physical memory and on the logical memory.
On the physical area, we check the number of empty pages, not yet used by any process, if enough pages need to be allocated, the physical area is ready. In addition, to optimize the search time when falling into the case of insufficient memory, we can organize *_mem_stat* as a list, including size management, free memory, ... for quick access to the required information.
On the logical memory area, we check based on the break point of the process, not exceeding the allowed memory area.

```c
int memmory_available_to_allocate(int num_pages, struct pcb_t * proc) {
  // Check physical space
  int i = 0;
  int cnt_pages = 0; // count free pages
  for (i = 0; i < NUM_PAGES; i++) {
    if (_mem_stat[i].proc == 0) {
      if (++cnt_pages == num_pages) break;
    }
  }
  if (cnt_pages < num_pages) return false;

  // Check virtual space
  if (proc->bp + num_pages*PAGE_SIZE >= RAM_SIZE) return false;

  return true;
}
```

2.3.3.2 Allocate memory : Implementation steps:

- Browse the physical memory, find idle pages, assign this page to be used by the process.
- Create *last_allocated_page_index* variable to make updating *next* value easier.
- On the logical memory area, based on the allocated address, from the starting address and the page position, we find its segments and pages. From there update the corresponding paging and segment tables.

Implementation details are described below.

```c
void allocate_memory_available(int ret_mem, int num_pages, struct pcb_t * proc) {

  int cnt_pages = 0; // count allocated pages
  int last_allocated_page_index = -1; // use for update field [next] of last allocated ↩
      page
  int i;
  for (i = 0; i < NUM_PAGES; i++) {
    if (_mem_stat[i].proc) continue; // page is used

    _mem_stat[i].proc = proc->pid; // the page is used by process [proc]
    _mem_stat[i].index = cnt_pages; // index in list of allocated pages

    if (last_allocated_page_index > -1) { // not initial page, update last page
      _mem_stat[last_allocated_page_index].next = i;
    }
    last_allocated_page_index = i; // update last page

    // Find or Create virtual page table
    addr_t v_address = ret_mem + cnt_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);
    if (v_page_table == NULL) {
      int idx = proc->seg_table->size;
      proc->seg_table->table[idx].v_index = v_segment;
      v_page_table
        = proc->seg_table->table[idx].pages
        = (struct page_table_t*) malloc(sizeof(struct page_table_t));
      proc->seg_table->size++;
    }
    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

    if (++cnt_pages == num_pages) {
      _mem_stat[i].next = -1; // last page in list
      break;
    }
  }
}
```

2.3.4 Memory recovery

2.3.4.1 Physical address revocation : Convert the logical address from the process to the physical, then based on the next value of the mem, we update the corresponding address string.

```c
  addr_t v_address = address; // virtual address to free in process
  addr_t p_address = 0;   // physical address to free in memory

  // Find physical page in memory
  if (!translate(v_address, &p_address, proc)) return 1;

  // Clear physical page in memory
  addr_t p_segment_page_index = p_address >> OFFSET_LEN;
  int num_pages = 0; // number of pages in list
  int i;
  for (i=p_segment_page_index; i!=-1; i=_mem_stat[i].next) {
    num_pages++;
    _mem_stat[i].proc = 0; // clear physical memory
  }
```

2.3.4.2 Logical address update : Based on the number of deleted pages on the block of the physical address, we find the pages on the logical address in turn, based on the address, we find the corresponding segment and page. Then update the paging table again, after the update process, if the table is empty, delete this table in the segment.

```c
static int remove_page_table(addr_t v_segment, struct seg_table_t * seg_table) {
  if (seg_table == NULL) return 0;
  int i;
  for (i = 0; i < seg_table->size; i++) {
    if (seg_table->table[i].v_index == v_segment) {
      int idx = seg_table->size-1;
      seg_table->table[i] = seg_table->table[idx];
      seg_table->table[idx].v_index = 0;
      free(seg_table->table[idx].pages);
      seg_table->size--;
      return 1;
    }
  }
  return 0;
}

...
  // Clear virtual page in process
  for (i = 0; i < num_pages; i++) {
    addr_t v_addr = v_address + i * PAGE_SIZE;
    addr_t v_segment = get_first_lv(v_addr);
    addr_t v_page = get_second_lv(v_addr);

    struct page_table_t * page_table = get_page_table(v_segment, proc->seg_table);
    if (page_table == NULL) {
      puts("============== Error ===============");
      continue;
    }
    int j;
    for (j = 0; j < page_table->size; j++) {
      if (page_table->table[j].v_index == v_page) {
        int last = --page_table->size;
        page_table->table[j] = page_table->table[last];
        break;
      }
    }
    if (page_table->size == 0) {
      remove_page_table(v_segment, proc->seg_table);
    }
  }
```

2.3.4.3 Update break point : Executed only when the last block on the logical address is deleted, then from there cycle back through the pages, until the page is being used, then stop.

```
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);
    addr_t last_page = get_second_lv(last_addr);
    struct page_table_t * page_table = get_page_table(last_segment, proc->seg_table);
    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;
  }
}

...
  // Update break pointer
  addr_t v_segment_page = v_address >> OFFSET_LEN;
  if (v_segment_page + num_pages * PAGE_SIZE == proc->bp) {
    free_mem_break_point(proc);
  }
```

3 Put it all together

Finally, we combine scheduler and Virtual Memory Engine to form a complete OS.Then we check work by fist compiling the whole source code with following command "make all" and get results by running the following command "make test_all" which is saved in file "result/os".
The Gantt diagrams below represent case of "result/all" in source code.

Test 0:

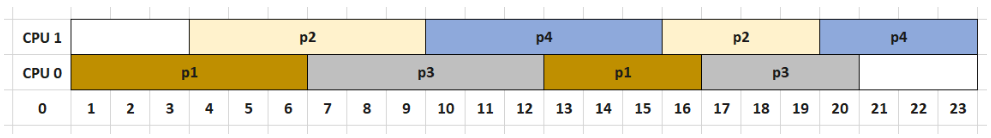

Figure 3: *Gantt diagram for Test 0*

Test 1:

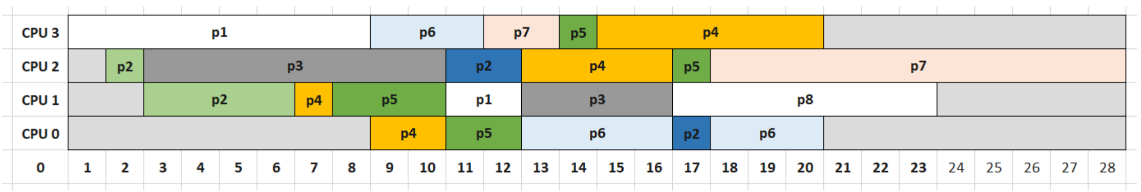

Figure 4: *Gantt diagram for Test I*