ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

BÁO CÁO

OPERATING SYSTEM - CO2017

ASSIGNMENT 1

GVHD: **Nguyễn Quang Hùng**

DSSV: Hồ Đức Hưng - 2013381
Hà Việt Đức - 2012987
Lê Trần Phúc Lộc - 2013684
Đoàn Thảo Nhi - 2014010

TP. Hồ Chí Minh, Tháng 2/2022

Mục lục

1 Scheduler

1.1 Giải thuật định thời (Scheduler Algorithms)

Q: What is the advantage of using priority feedback queue in comparison with other scheduling algorithms you have learned such as FIFO, Round Robin? Explain clearly your answer.

A: Giải thuật Priority Feedback Queue (PFQ) sử dụng tư tưởng của một số giải thuật khác gồm giaí thuật Priority Scheduling - mỗi process mang một độ ưu tiên để thực thi, giải thuật Multilevel Queue - sử dụng nhiều mức hàng đợi các process, giải thuật Round Robin - sử dụng quantum time cho các process thực thi. Dưới đây là các giải thuật định thời khác đã học:

- First Come First Served (FCFS)
- Shortest Job First (SJF)
- Shortest Remaining Time First (SRTF))
- Priority Scheduling (PS)
- Round Robin (RR)
- Multilevel Queue Scheduling (MLQS)
- Multilevel Feedback Queue (MLFQ)

Cụ thể, giải thuật PFQ sử dụng 2 hàng đợi là **ready_queue** và **run_queue** với ý nghĩa như sau:

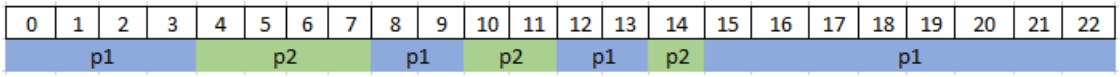
- **ready_queue**: hàng đợi chứa các process ở mức độ ưu tiên thực thi trước hơn so với hàng đợi
- **run_queue**: hàng đợi này chứa các process đang chờ để tiếp tục thực thi sau khi hết slot của nó mà chưa hoàn tất quá trình của mình. Các process ở hàng đợi này chỉ được tiếp tục slot tiếp theo khi **ready_queue** rỗi và được đưa sang hàng đợi **ready_queue** để xét slot tiếp theo.
- Cả hai hàng đợi đều là hàng đợi có độ ưu tiên, mức độ ưu tiên dựa trên mức độ ưu tiên của process trong hàng đợi.

Ưu điểm của giải thuật PFQ

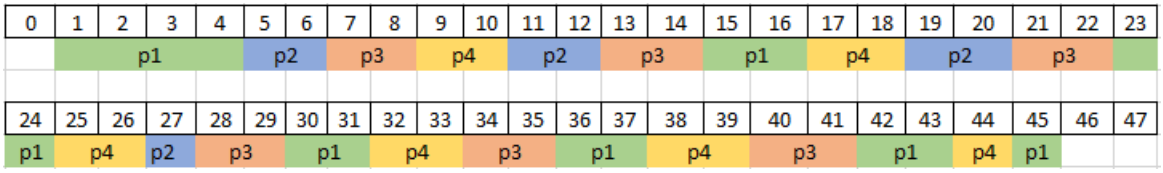
- Sử dụng time slot, mang tư tưởng của giải thuật RR với 1 khoảng quantum time, tạo sự công bằng về thời gian thực thi giữa các process, tránh tình trạng chiếm CPU sử dụng, trì hoãn vô hạn định.
- Sử dụng hai hàng đợi, mang tư tưởng của giải thuật MLQS và MLFQ, trong đó hai hàng đợi được chuyển qua lại các process đến khi process được hoàn tất, tăng thời gian đáp ứng cho các process (các process có độ ưu tiên thấp đến sau vẫn có thể được thực thi trước các process có độ ưu tiên cao hơn sau khi đã xong slot của mình).
- Tính công bằng giữa các process là được đảm bảo, chỉ phụ thuộc vào độ ưu tiên có sẵn của các process. Cụ thể xét trong khoảng thời gian **t0** nào đó, nếu các process đang thực thi thì hoàn toàn phụ thuộc vào độ ưu tiên của chúng. Nếu có 1 process **p0** khác đến, giả sử **ready_queue** đang sẵn sàng, process **p0** này vào hàng đợi ưu tiên và phụ thuộc vào độ ưu tiên của nó, cho dù trước đó các process khác có độ ưu tiên cao hơn đã thực thi xong, chúng cũng không thể tranh chấp với process **p0** được vì chúng đang chờ trong **run_queue** cho đến khi **ready_queue** là rỗi, tức **p0** đã được thực thi slot của nó.

1.2 Gantt diagrams

YÊU CẦU: draw Gantt diagram describing how processes are executed by the CPU.



Gantt diagram `sched_0`



Gantt diagram `sched_1`

1.3 Hiện thực

1.3.1 Priority Queue (Hàng đợi ưu tiên)

Priority queue (Hàng đợi ưu tiên) tuân thủ các quy tắc sau đây:

- Tất cả các phần tử đều có giá trị ưu tiên
- Phần tử nào có giá trị ưu tiên cao hơn sẽ được dequeue trước phần tử có giá trị ưu tiên thấp hơn.
- Nếu hai mục có độ ưu tiên bằng nhau thì mức độ ưu tiên của chúng được sắp xếp theo thứ tự của chúng trong hàng đợi

Trong bài tập lớn này, ta sẽ hiện thực hàng đợi ưu tiên theo dạng **ArrayQueue** với phần tử tối đa trong hàng đợi là 10 (`MAX_QUEUE_SIZE = 10`) với hai hàm `enqueue()` và `dequeue()`

- `enqueue()`: thêm một giá trị vào hàng đợi.
- `dequeue()`: trả 1 giá trị với độ ưu tiên cao nhất ra khỏi hàng đợi.

Các hàm `enqueue()` và `dequeue()` được hiện thực trong file `queue.c`, nguồn `src/queue.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

int empty(struct queue_t * q) {
   return (q->size == 0);
}

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;
   q->proc[q->size] = 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 == 0)
      return NULL;
```

```
    struct pcb_t * item = NULL;
    uint32_t maxPriority = q->proc[0]->priority;
    int idxOfMaxPriotity = 0;
    for(int i = 0 ; i < q->size ; i++)
    {
        if(maxPriority < q->proc[i]->priority)
        {
            maxPriority = q->proc[i]->priority;
            idxOfMaxPriotity = i;
        }
    }
    item = q->proc[idxOfMaxPriotity];
    for(int i = idxOfMaxPriotity + 1; i < q->size ; i++)
    {
        q->proc[i - 1] = q->proc[i];
    }
    q->size--;
    return item;
}
```

1.3.2 Scheduler

Mục **Scheduler** sẽ quản lý và cập nhật các process thông qua hàm `loader()`. Tuy nhiên, trong bài tập lớn này, ta chỉ hoàn thành quá trình tìm kiếm cho CPU với hàm `get_proc()` với hai hàng đợi có sẵn

- `get_proc()`: trả về một process từ `ready_queue`, nếu hàng đợi trống, chúng ta thêm tất cả process từ `run_queue`, sau đó, chúng ta tiếp tục `dequeue` và lấy process có độ ưu tiên cao nhất từ `ready_queue`

Mã code hiện thực trong file `sched.c`

```
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)) // if ready_queue is empty
    {
        while(!empty(&run_queue)) // push all process in run_queue back to ready_queue
        {
            enqueue(&ready_queue, dequeue(&run_queue));
        }
    }
    if(!empty(&ready_queue)) // if ready queue isn't empty
    {
        proc = dequeue(&ready_queue); // get a process from ready_queue
    }
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

2 Memory Management

2.1 Segment with paging

Q: In which system is segmentation with paging used (give an example of at least one system)? Explain clearly the advantage and disadvantage of segmentation with paging.

A: Trong **Segmentation with Paging**, bộ nhớ chính được phân chia thành các segments có kích thước thay đổi cái mà được chia thành các page có kích thước cố định.

- Pages nhỏ hơn segments.
- Mỗi segments có nhiều page table nghĩa là mọi chương trình đều có multiple page table.
- Địa chỉ logic được biểu diễn dưới dạng **Segment Number (base address), Page number và page offset.**

Ưu điểm của Segmentation with Paging:

- Nó làm giảm mức sử dụng memory.
- Kích thước của page table bị giới hạn bởi kích thước segments.
- Segment table chỉ có một mục nhập tương ứng với một segments thực tế.
- External Fragmentation không có ở đây.
- Đơn giản hóa việc cấp phát bộ nhớ.

Nhược điểm của Segmentation with Paging:

- Internal Fragmentation sẽ ở đó.
- Mức độ phức tạp cao hơn nhiều so với page.
- Page table cần được lưu trữ liền nhau trong bộ nhớ.

2.2 Memory status result (RAM)

YÊU CẦU : Show the status of RAM after each memory allocation and deallocation function call

2.3 Hiện thực

2.3.1 Search for Page table

Trong bài tập này ta giả định rằng kích thước của RAM ảo là 1 MB, vì vậy ta sử dụng 20 bit để đại diện cho mỗi byte của nó bao gồm

- 5 bit đầu tiên cho **segment index**
- 5 bit tiếp theo cho **page index**
- 10 bit cuối cho **offset**

Trong hàm `get_page_table()` thuộc file `mem.c` ta thực hiện các nhiệm vụ sau:

- Tìm page table có segment index của một process.

Hiện thực của `get_page_table()` được thực hiện như sau:

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

   /*
    * TODO: Given the Segment index [index], you must go through each
    * row of the segment table [seg_table] and check if the v_index
    * field of the row is equal to the index
    *
    * */
   if(!seg_table)
      return NULL;
   int i;
   for (i = 0; i < seg_table->size; i++) {
      // Enter your code here
      if(index == seg_table->table[i].v_index)
      {
         return seg_table->table[i].pages;
      }
   }
   return NULL;
}
```

2.3.2 Address Translation Scheme

- Trong bài tập lớn này, mỗi địa chỉ được biểu diễn bởi 20 bit trong đó gồm có 10 bits đầu (5 bits segment và 5 bits page) và 10 bits cuối là offset.
- Bởi vì ta cần translate từ **virtual address** thành **physical address** nên ta cần dùng **p_index** trong **page_table_t** làm 10 bits đầu kết hợp với 10 bits offset.
- Như vậy ta cần dịch trái 10 bits đầu rồi or (|) với chuỗi 10 bits offset.

Hiện thực của **translate()** được thực hiện như sau:

```
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 addess
       * to [p_index] field of page_table->table[i] to
       * produce the correct physical address and save it to
       * [*physical_addr]  */
      *physical_addr = page_table->table[i].p_index << OFFSET_LEN | offset;
      return 1;
    }
  }
  return 0;
}
```

2.3.3 Allocate Memory

Kiểm tra bộ nhớ có sẵn cho cả **physical space** (_mem_stat) và **virtual space** (break point).

```
  uint32_t cr_num_pages = 0;
  for (int i = 0; i < NUM_PAGES; i++) {
    if (_mem_stat[i].proc == 0) {
      cr_num_pages++;
    }
  }
  if (cr_num_pages >= num_pages &&
      proc->bp + num_pages * PAGE_SIZE < NUM_PAGES * PAGE_SIZE + PAGE_SIZE)
    mem_avail = 1;
  else
    ret_mem = 0;
```

Khi **Allocate**, duyệt qua bộ nhớ vật lý, tìm những trang rảnh, và gán những process đã được sử dụng trên những trang này. Đồng thời chúng ta phải cập nhật lại các field [id], [index], và [next]. Và cuối cùng thêm entry mới vào seg_table

```
    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_pages_use = 0;
    for (int i = 0, j = 0, k = 0; i < NUM_PAGES; i++) {
      if (_mem_stat[i].proc == 0) {
        _mem_stat[i].proc = proc->pid;
        _mem_stat[i].index = j;
        if (j != 0)
          _mem_stat[k].next = i;
        /*------------------------*/
        addr_t physical_addr = i << OFFSET_LEN;
        addr_t first_lv = get_first_lv(ret_mem + j * PAGE_SIZE);
        addr_t second_lv = get_second_lv(ret_mem + j * PAGE_SIZE);
        // addr_t offset = get_offset(ret_mem + j * PAGE_SIZE);
        int booler = 0;
```

```
        /*Neu da co first index trong page table*/
        for (int n = 0; n < proc->seg_table->size; n++) {
          if (proc->seg_table->table[n].v_index == first_lv) {
            proc->seg_table->table[n]
                .pages->table[proc->seg_table->table[n].pages->size]
                .v_index = second_lv;
            proc->seg_table->table[n]
                .pages->table[proc->seg_table->table[n].pages->size]
                .p_index = physical_addr >> OFFSET_LEN;
            proc->seg_table->table[n].pages->size++;
            booler = 1;
            break;
          }
        }
        /*Neu chua co first index trong seg table tao page table moi*/
        if (booler == 0) {
          int n = proc->seg_table->size;
          proc->seg_table->size++;
          proc->seg_table->table[n].pages =
              (struct page_table_t *)malloc(sizeof(struct page_table_t));
          proc->seg_table->table[n].pages->size++;
          proc->seg_table->table[n].v_index = first_lv;
          proc->seg_table->table[n].pages->table[0].v_index = second_lv;
          proc->seg_table->table[n].pages->table[0].p_index =
              physical_addr >> OFFSET_LEN;
        }
        /*------------------------*/
        k = i;
        j++;
        num_pages_use++;
        if (num_pages_use == num_pages) {
          _mem_stat[k].next = -1;
          break;
        }
      }
    }
  }

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

2.3.4 De–allocate Memory

Chuyển địa chỉ luận lý từ process thành vật lý, sau đó dựa trên giá trị next của mem, ta cập nhật lại chuỗi địa chỉ tương ứng đó.

```
    pthread_mutex_lock(&mem_lock);
addr_t physical_addr;
if (translate(address, &physical_addr, proc)) {
  // int number1=0,first_num=0;
  int next = -2;
  int i = 0, j = 0;
  /*Xoa tim ra vi tri ung voi address*/
  for (; i < NUM_PAGES; i++) {
    if (physical_addr == i << OFFSET_LEN) {
```

```
      // first_num=i;
      break;
    }
  }
  /*Ung voi vi tri vua tim duoc ta xoa nhung cai tiep theo*/
  next = i;
  while (next != -1) {
    // number1++;
    _mem_stat[next].proc = 0;
    next = _mem_stat[next].next;
  }
```

Dựa trên số trang đã xóa trên block của địa chỉ vật lý, ta tìm lần lượt các trang trên địa chỉ luận lý, dựa trên địa chỉ, ta tìm được segment, page tương ứng. Sau đó cập nhật lại bảng phân trang, sau quá trình cập nhật, nếu bảng trống thì xóa bảng này trong segment đi.

```
      pthread_mutex_lock(&mem_lock);
addr_t physical_addr;
if (translate(address, &physical_addr, proc)) {
  // int number1=0,first_num=0;
  int next = -2;
  int i = 0, j = 0;
  /*Xoa tim ra vi tri ung voi address*/
  for (; i < NUM_PAGES; i++) {
    if (physical_addr == i << OFFSET_LEN) {
      // first_num=i;
      break;
    }
  }
  /*with current index delete next*/
  next = i;
  while (next != -1) {
    // number1++;
    _mem_stat[next].proc = 0;
    next = _mem_stat[next].next;
    /*----------------*/
    /*Delete seg table and page table*/
    addr_t first_lv = get_first_lv(address + j * PAGE_SIZE);
    addr_t second_lv = get_second_lv(address + j * PAGE_SIZE);
    for (int n = 0; n < proc->seg_table->size; n++) {
      if (proc->seg_table->table[n].v_index == first_lv) {
        for (int m = 0; m < proc->seg_table->table[n].pages->size; m++) {
          if (proc->seg_table->table[n].pages->table[m].v_index ==
              second_lv) {
            /*----------------*/
            /*Update page table, delete the last element*/
            int k = 0;
            for (k = m; k < proc->seg_table->table[n].pages->size - 1; k++) {
              proc->seg_table->table[n].pages->table[k].v_index =
                  proc->seg_table->table[n].pages->table[k + 1].v_index;
              proc->seg_table->table[n].pages->table[k].p_index =
                  proc->seg_table->table[n].pages->table[k + 1].p_index;
            }
            proc->seg_table->table[n].pages->table[k].v_index = 0;
            proc->seg_table->table[n].pages->table[k].p_index = 0;
            proc->seg_table->table[n].pages->size--;
```

```
            /*----------------*/
            break;
          }
        }
        if (proc->seg_table->table[n].pages->size == 0) {
          /* if page-table empty so delete it */
          free(proc->seg_table->table[n].pages);
          int m = 0;
          for (m = n; m < proc->seg_table->size - 1; m++) {
            proc->seg_table->table[m].v_index =
                proc->seg_table->table[m + 1].v_index;
            proc->seg_table->table[m].pages =
                proc->seg_table->table[m + 1].pages;
          }
          proc->seg_table->table[m].v_index = 0;
          proc->seg_table->table[m].pages = NULL;
          proc->seg_table->size--;
        }
        break;
      }
    }
    j++;
  }
}
pthread_mutex_unlock(&mem_lock);
return 0;
```

3 Put It All Together

3.1 Synchronization

Q: What will be happen if the synchronization is not handled in your system? Illustrate the problem by example if you have any.

A: Khi có nhiều process truy xuất và thao tác đồng thời lên tài nguyên chung, kết quả của việc truy xuất đồng thời này phụ thuộc thứ tự thực thi của các lệnh thao tác dữ liệu. Để dữ liệu chia sẻ được nhất quán, cần đảm bảo sao cho các process lần lượt thao tác lên tài nguyên chung. Do đó, cần có cơ chế đồng bộ hoạt động của các process này. Nếu không có cơ chế đồng bộ thì process này sẽ có thể can thiệp vào một process khác trong khi đang sử dụng tài nguyên được chia sẻ. Tình trạng này được gọi là **Race Condition**.

3.1.1 Race condition

Race condition là một tình huống không mong muốn xảy ra khi mà một thiết bị hoặc hệ thống cố gắng thực hiện hai hay nhiều hoạt động cùng một lúc, nhưng do bản chất của thiết bị hoặc hệ thống, các hoạt động phải được thực hiện theo trình tự thích hợp để được thực hiện chính xác.

Race condition khá phổ biến trong khoa học máy tính và lập trình. Nó xảy ra khi các process hoặc các luồng cố gắng truy cập vào cùng tài nguyên đồng thời và gây ra lỗi trong hệ thống.

3.1.2 Examples

Một ví dụ đơn giản về **Race Condition** là việc sử dụng công tắc đèn. Trong một số ngôi nhà, có nhiều công tắc đèn được kết nối với một đèn trần chung. Nếu đèn sáng thì việc tắt một trong các công tắc sẽ làm đèn tắt. Tương tự, nếu đèn tắt, thì việc bật một trong hai công tắc sẽ làm đèn sáng.

Tiếp theo, hãy tưởng tượng điều gì có thể xảy ra nếu hai người cố gắng bật đèn bằng cả hai công tắc cùng một lúc. Một trong hai công tắc sẽ bật được đèn hoặc việc bật đồng thời có thể ngắt luôn cầu dao vì có sự cố.

Điều này có thể xảy ra tương tự với máy tính của chúng ta. Trong bộ nhớ hoặc bộ lưu trữ của máy tính, **Race Condition** sẽ diễn ra khi các lệnh đọc và ghi một lượng lớn dữ liệu, được nhận gần như cùng một lúc và máy tính cố gắng ghi đè dữ liệu trong khi các dữ liệu đang được đọc. Kết quả sẽ là một hoặc các điều sau:

- Máy tính sẽ bị treo.
- Lỗi khi đọc dữ liệu cũ.
- Lỗi khi đọc dữ liệu mới.

3.2 Hiện thực