ĐẠI HỌC BÁCH KHOA THÀNH PHỐ HỒ CHÍ MINH
**KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH**

- - - - - - - - - - - - - - - - - - - - - -


# HỆ ĐIỀU HÀNH

## Assignment 2

## Simple Operating System

Giáo viên hướng dẫn: Nguyễn Minh Trí
Sinh viên: 1412958 - Nguyễn Hồng Phúc
1610852 - Huỳnh Sâm Hà

Thành phố Hồ Chí Minh, 4/2018

# Mục lục

# Danh sách hình vẽ

# 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?

Giải thuật Priority Feedback Queue (PFQ) sử dụng tư tưởng của một số giải thuật khác gồm giaỉ 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 $run\_queue$. Khi CPU chuyển sang slot tiếp theo, nó sẽ tìm kiếm process trong hàng đợi này.
- $run\_queue$: 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.

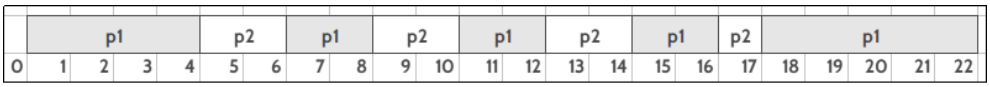
**Ưu điểm của giải thuật PFQ**

- 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 Result - Gantt Diagrams

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

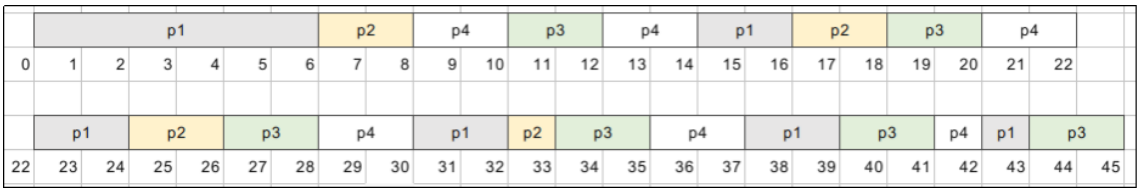
Dưới đây là giản đồ Gantt cho trường hợp trong log/sched1.txt trong source code.
**Test 0:**




**Hình 1:** *Lược đồ Gantt CPU thực thi các processes - test 0*

Trong test này, CPU xử lí trên 2 process p1 và p2 trong 22 time slot như lược đồ Gantt ở trên.

**Test 1:**




**Hình 2:** *Lược đồ Gantt CPU thực thi các processes - test 1*

Trong test này, CPU xử lí trên 4 process p1, p2, p3 và p4 trong 46 time slot như lược đồ Gantt ở trên.

## 1.3 Implementation

### 1.3.1 Priority Queue

Hàng đợi ưu tiên trong trường hợp này xử lý cho không quá 10 process, do đó, ta đơn giản chỉ cần dùng vòng lặp để xử lý chức năng mà một hàng đợi ưu tiên cần có.
Cụ thể với hàm $enqueue()$, ta chỉ cần đưa vào cuối hàng đợi nếu sẵn sàng (còn trống). Với hàm $dequeue()$, ta duyệt tìm process có độ ưu tiên cao nhất ra, đồng thời cập nhật lại trạng thái của queue khi xóa 1 phần tử.
Dưới đây là phần hiện thực hàng đợi ưu tiên cho Scheduler.

```
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];
  q->proc[i] = q->proc[--q->size];

  return res;
}
```

### 1.3.2 Scheduler

Nhiệm vụ của scheduler là quản lý việc cập nhật các process sẽ được thực thi cho CPU. Cụ thể scheduler sẽ quản lý 2 hàng đợi ready và run như ở trên đã mô tả. Trong assignment này, ta chỉ cần hiện thực tiếp hàm tìm một process cho CPU thực thi.
Cụ thể, với hàm o$get_proc()$, trả về một process trong hàng đợi ready, nếu hàng đợi ready rỗi, ta cấp nhật lại hàng đợi bằng các process đang chờ cho các slot tiếp theo trong hàng đợi run. Ngược lại, ta tìm ra process có độ ưu tiên cao từ hàng đợi này.
Dưới đây là phần hiện thực của chức năng nói trên.

```
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?

**Ưu điểm của giải thuật**

- Tiết kiệm bộ nhớ, sử dụng bộ nhớ hiệu quả.
- Mang các ưu điểm của giải thuật phân trang:
  Đơn giản việc cấp phát vùng nhớ.
  Khắc phục được phân mảnh ngoại.
- Giải quyết vấn đề phân mảnh ngoại của giải thuật phân đoạn bằng cách phân trang trong mỗi đoạn.

**Nhược điểm của giải thuật**

- Phân mảnh nội của giải thuật phân trang vẫn còn.

## 2.2 Result - Status of RAM

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

Dưới đây là kết quả của quá trình ghi log sau mỗi lệnh allocation và deallocation trong chương trình, cụ thể ghi lại trạng thái của RAM trong chương trình ở mỗi bước.

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

**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 Tìm bảng phân trang từ segment

Trong assignment này, mỗi địa chỉ được biểu diễn bởi 20 bits, trong đó 5 bits đầu tiên là segment, 5 bits tiếp theo là page, và 10 bits cuối là offset.
Chức năng này nhận vào 5 bits segment $index$ và bảng phân đoạn $seg\_table$, cần tìm ra bảng phân trang $res$ của segment tương ứng trong bảng phân đoạn nói trên.
Do bảng phân đoạn $seg\_table$ là một danh sách gồm các phần tử $u$ có cấu trúc $(v\_index, page\_table\_t)$, trong đó $v\_index$ là 5 bits segment của phần tử $u$ và $page\_table\_t$ là bảng phân trang tương ứng của segment đó. Vì vậy để tìm được $res$, ta chỉ cần duyệt trên bảng phân đoạn này, phần tử $u$ nào có $v\_index$ bằng $index$ cần tìm, ta trả về $page\_table$ tương ứng.
Dưới đây là phần hiện thực cho chức năng trên.

```
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 Ánh xạ địa chỉ ảo thành địa chỉ vật lý

Do mỗi địa chỉ gồm 20 bits với cách tổ chức như nói ở trên, do đó để tạo được địa chỉ vật lý, ta lấy 10 bits đầu (segment và page) nối với 10 bits cuối (offset). Mỗi $page\_table\_t$ lưu các phần tử có $p\_index$ là 10 bits đầu đó. do đó để tạo được địa chỉ vật lý, ta chỉ cần dịch trái 10 bits đó đi 10 bits offset rồi or (|) hai chuỗi này lại.
Dưới đây là phần hiện thực của chức năng trên.

```
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 Cấp phát memory

**2.3.3.1 Kiểm tra memory sẵn sàng** Bước này ta kiểm tra xem memory có sẵn sàng cả trên bộ nhớ vật lý và bộ nhớ luận lí hay không.
Trên vùng vật lý, ta duyệt kiểm tra số lượng trang còn trống, chưa được process nào sử dụng, nếu đủ số trang cần cấp phát thì vùng vật lý đã sẵn sàng. Ngoài ra để tối ưu thời gian tìm kiếm khi rơi vào trường hợp không đủ vùng nhớ, ta có thể tổ chức $\_mem\_stat$ dưới dạng danh sách, trong đó có quản lý kích thước, vùng nhớ trống, ... để truy xuất các thông tin cần thiết nhanh chóng.
Trên vùng nhớ luận lý, ta kiểm tra dựa trên break point của process, không vượt quá vùng nhớ cho phép.

```
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 Alloc memory** Các bước thực hiện:

- Duyệt trên vùng nhớ vật lý, tìm các trang rỗi, gán trang này được process sử dụng.
- Tạo biến $last\_allocated\_page\_index$ để cập nhật giá trị $next$ dễ dàng hơn.
- Trên vùng nhớ luận lý, dựa trên địa chỉ cấp phát, tính từ địa chỉ bắt đầu và vị trí thứ tự trang cấp phát, ta tìm được các segment, page của nó. Từ đó cập nhật các bảng phân trang, phân đoạn tương ứng.

Dưới đây là phần hiện thực chi tiết.

```
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 Thu hồi memory

**2.3.4.1 Thu hồi địa chỉ vật lý** 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.

```
...
  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 Cập nhật địa chỉ luận lý** 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.

```
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);
    }
  }
...
```

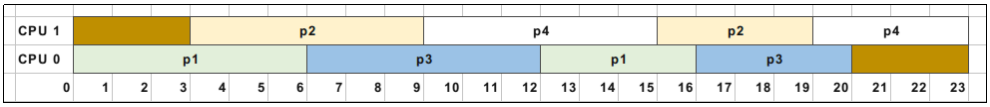
**2.3.4.3 Cập nhật break point** Chỉ thực hiện khi block cuối cùng trên địa chỉ luận lý được xóa, sau đó từ đó duyệt lần lượt ngược lại các trang, đến khi đến trang đang được sử dụng thì dừng.

```
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);
  }
...
```

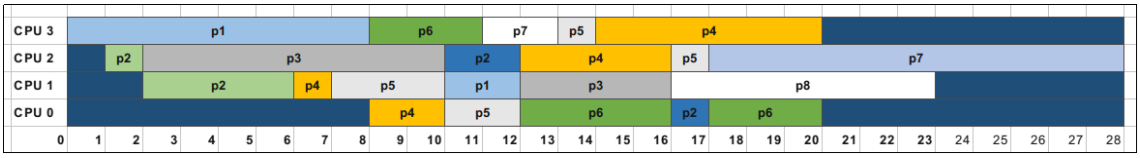
# 3 Put it all together

Sau khi kết hợp cả scheduling và memory, ta thực hiện make all và có kết qủa như các file log trong thư mục log/os*.txt
Dưới đây là giản đồ Gantt cho trường hợp trong log/all1.txt trong source code.
**Test 0:**




**Hình 3:** *Lược đồ Gantt CPU thực thi các processes cho make all*

**Test 1:**




**Hình 4:** *Lược đồ Gantt CPU thực thi các processes cho make all*

# Tài liệu