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

Hệ Điều Hành

Bài tập lớn số 2:

SIMPLE OPERATING SYSTEM

GVHD:	Nguyễn Hoài Nam	
NHÓM:	14	
SVTH:	Nguyễn Tấn Đạt	1810700
	Nguyễn Huỳnh Long	1811049
	Nguyễn Đình Khánh	1810992
	Nguyễn Hoàng Khang	1812545

Tp. Hồ Chí Minh, Tháng 6/2020

Mục lục

Nguyễn Tấn Đạt	Cấp phát bộ nhớ, Tìm bảng phân trang, Ánh xạ địa chỉ luận lý thành địa chỉ vật lý
Nguyễn Huỳnh Long	Thu hồi địa chỉ vật lý, Cập nhật địa chỉ luận lý, Cập nhật break point
Nguyễn Đình Khánh	Trả lời câu hỏi, Vẽ sơ đồ Gantt, Cấp phát bộ nhớ
Nguyễn Hoàng Khang	Scheduler, Vẽ sơ đồ Gantt, Priority Queue, Trả lời câu hỏi

Bảng phân chia công việc

1 Scheduler

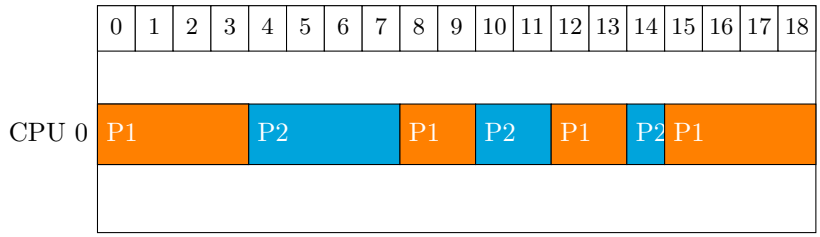
1.1 Trả lời câu hỏi

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

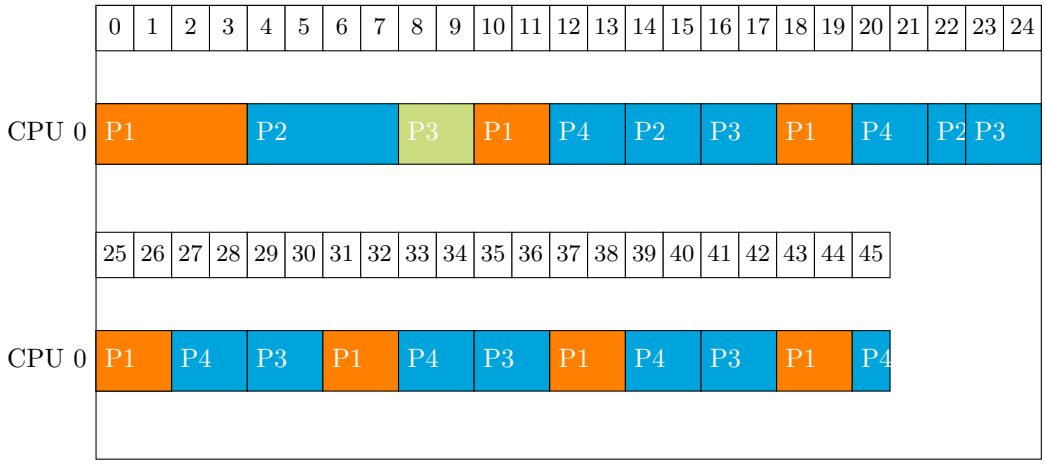
- Priority Feedback Queue (PFQ) có ý tưởng giống với các giải thuật là Round Robin (RR), Priority Scheduling, và Multilevel Queue nên PFQ thừa hưởng các lợi thế sau:
  - Priority Scheduling: sử dụng độ ưu tiên khi lựa chọn process giúp các process quan trọng hơn được xử lý trước.
  - Round Robin: sử dụng quantum time giúp các process đều được thực thi luân phiên và tránh tình trạng 'starvation' khi một process có độ ưu tiên thấp có thể sẽ không được thực thi.
  - Multilevel Queue: sử dụng nhiều hàng đợi (cụ thể trong PFQ là ready queue và run queue) giúp lựa chọn hiệu quả hơn.
- Tuy nhiên, ta có thể thấy các giải thuật trên tồn tại một số nhược điểm:
  - Priority Scheduling: một process có priority thấp có thể phải đợi vô thời hạn.
  - Round Robin: thời gian đợi trung bình lớn. Nếu như một process cần được thực thi trước (priority cao) phải đợi lâu thì hệ thống sẽ không đượoc vận hành tốt.
  - Multilevel Queue: Giải thuật có độ phức tạp cao, ngoài ra Process ở level thấp nhất có khả năng không được thực thi hoặc phải đợi thời gian dài do không được cung cấp tài nguyên.
- Khi áp dụng nhiều ý tưởng trên cho giải thuật, PFQ sẽ khắc phục nhược điểm của các giải thuật gốc:
  - Giả sử ta có một process q0 với priority rất cao và thời gian thực thi quá lớn. Một process mới q1 được thêm vào. Nếu chỉ có một queue, mặc dù q1 có priority cao nhưng không bằng q0, sau một hay nhiều quantum time thì q1 vẫn phải đợi rất lâu do thời gian thực thi của q0 quá lớn. Ở đây, PFQ sử dụng ready_queue để giữ những process đợi được đưa vào thực thi và run_queue để giữ những process đã được thực thi sau một quantum_time. Cơ chế này giúp q1 sau khi được thêm vào ready_queue, q0 thực thi sau quantum time sẽ được đưa vào run_queue. Lúc này, giải thuật lựa chọn process có priority cao nhất trong ready_queue để thực thi, nếu q1 có priority cao nhất thì sẽ được lựa chọn. Điều này tương tự với q2,q3,q4.. với các priority khác nhau. Như vậy, giải thuật giúp tất cả process được thực thi luân phiên nhưng vẫn ưu tiên ưu tiên process có priority cao hơn và đảm bảo không process nào chờ đợi vô hạn.

1.2 Kết quả hiện thực

Yêu cầu: draw Gantt diagram describing how processes are executed by the CPU.



Hình 1: Lược đồ Gantt CPU thực thi các process ở test 0



Hình 2: Lược đồ Gantt CPU thực thi các process ở test 1

1.3 Hiện thực

1.3.1 Priority Queue

Hàng đợi ưu tiên gồm hai hàm cơ bản là enqueue() và dequeue().
Trong đó, enqueue() thêm process vào hàng đợi nếu hàng đợi còn chỗ trống, dequeue() trả về process trong hàng đợi có độ ưu tiên cao nhất. Cả hai hàm sẽ cập nhật lại kích thước của hàng đợi nếu có thay đổi về số lượng process trong hàng đợi.

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

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

```
    }
  }
  struct pcb_t * res = q->proc[max_idx];
  q->proc[max_idx] = q->proc[q->size-1];
  q->size--;
  return res;
}
```

1.3.2 Scheduler

Vai trò của Scheduler là điều phối các process sẽ được thực thi tới CPU bằng hai hàng đợi ready và run. Scheduler thực hiện việc này với hàm `get_proc()`. `get_proc()` trả về một process trong hàng đợi ready, nếu hàng đợi ready rỗng thì ta sẽ chuyển tất cả process trong hàng đợi run vào hàng đợi ready trước.

```
struct pcb_t * get_proc(void) {
  struct pcb_t * proc = NULL;
  pthread_mutex_lock(&queue_lock);
  if (empty(&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 Quản lý bộ nhớ

2.1 Trả lời câu hỏi

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

- Ưu điểm
  - Sử dụng bộ nhớ một cách tiết kiệm và hiệu quả.
  - Tận dụng được các lợi thế của giải thuật phân trang:
    - Đơn giản trong việc cấp phát bộ nhớ.
    - Tránh được phân mảnh ngoại.
  - Kích thước của bảng phân trang được giới hạn bởi kích thước của segment.
- Nhược điểm:
  - Giải thuật vẫn còn phân mảnh nội.
  - Độ phức tạp của giải thuật cao.Bảng phân trang cần phải được lưu trữ liên tục bên trong bộ nhớ.

2.2 Kết quả hiện thực

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

2.2.1 m0:

Với đoạn mã sau:

```
1 7
alloc 13535 0
alloc 1568 1
free 0
alloc 1386 2
alloc 4564 4
write 102 1 20
write 21 2 1000
```

Ta thu được các trạng thái của RAM như sau:

```
-------------------------------------------
Aloccation
-------------------------------------------
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)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-------------------------------------------
Aloccation
-------------------------------------------
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)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-------------------------------------------
DEALLOCATE
-------------------------------------------
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-------------------------------------------
Aloccation
-------------------------------------------
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)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
--------------------------------------------
Aloccation
--------------------------------------------
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)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
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)
```

2.2.2 m1:

Với đoạn mã sau:

```
1 8
alloc 13535 0
alloc 1568 1
free 0
alloc 1386 2
alloc 4564 4
free 2
free 4
free 1
```

Ta thu được các trạng thái của RAM như sau:

```
--------------------------------------------
Aloccation
--------------------------------------------
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)
```

```
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-----------------------------------------------
Aloccation
-----------------------------------------------
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)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-----------------------------------------------
DEALLOCATE
-----------------------------------------------
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-----------------------------------------------
Aloccation
-----------------------------------------------
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)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-----------------------------------------------
Aloccation
-----------------------------------------------
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)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-----------------------------------------------
DEALLOCATE
-----------------------------------------------
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)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-----------------------------------------------
DEALLOCATE
-----------------------------------------------
```

```
014: 03800-03bff - PID: 01 (idx 000, nxt: 015)
015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
-----------------------------------------
DEALLOCATE
-----------------------------------------
vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv
```

2.3 Hiện thực

2.3.1 Tìm bảng phân trang

Trong bài tập này, mỗi địa chỉ được biểu diễn bằng 20 bits, trong đó 5 bits đầu là segment, 5 bits tiếp theo là page và 10 bits cuối là offset.

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

Hàm get_page_table nhận vào 5 bits segment *index* và bảng phân đoạn *seg_table* sau đó trả về bảng phân trang của segment tương ứng trong bảng phân đoạn nói trên

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

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

Bảng phân đoạn seg_table là một danh sách gồm các phần tử có câu trúc (v_index, page_table_t), trong đó v_index là 5 bits segment của phần tử và page_table_t là bảng phân trang tương ứng của segment đó. Chỉ cần duyệt trên bảng phân đoạn này, phần tử nào có v_index bằng *index* cần tìm, ta trả về page_table_t tương ứng.

```
struct seg_table_t {
  struct {
    addr_t v_index; // Virtual index
    struct page_table_t * pages;
  } table[1 << PAGE_LEN];
  int size; // Number of row in the first layer
};
```

2.3.2 Ánh xạ địa chỉ luận lý thành địa chỉ vật lý

Do mỗi địa chỉ gồm 20 bits với cách tổ chức như trên nên để tạo được địa chỉ vật lý, chúng 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. Do đó để tạo được địa chỉ vật lý, chúng ta cần dịch trái 10 bits đi 10 bits offset rồi or (|) hai chuỗi này lại.

```c
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;
      *physical_addr = *physical_addr | offset;
      return 1;
    }
  }
  return 0;
}
```

2.4 Cấp phát bộ nhớ

Các bước thực hiện:

1. Kiểm tra xem trạng thái có thể cấp phát của bộ nhớ:
   (a) Trên bộ nhớ vật lý: Dựa cấu trúc dạng danh sách của ***_mem_stat*** ta duyệt để tìm xem có đủ số page sẵn sàng được cấp phát hay không.
   (b) Trên bộ nhớ luận lý: dựa trên ***break point*** của proc để kiểm tra xem liệu có thể cấp phát đủ bộ nhớ cần thiết hay không.

```c
int i = 0;
int mem_avail = 0;
int num_free_pages = 0; // count free pages
for (i = 0; i < NUM_PAGES; i++) {
  if (_mem_stat[i].proc == 0) {
    num_free_pages += 1; // Count number of free pages in _mem_stat
    if (num_free_pages == num_pages) break;
  }
}
if (num_free_pages < num_pages) {
  mem_avail = 0; // check in physical memory
}
else if (proc->bp + num_pages*PAGE_SIZE >= RAM_SIZE) {
  mem_avail = 0; // check in logic memory
}
```

```
else {
  mem_avail = 1;
}

```

2. Cấp phát bộ nhớ nếu bước trên chỉ ra có thể cấp phát:

   (a) Duyệt trên bộ nhớ vật lý từ đầu đến cuối ***_mem_stat***, nếu trang trong trạng thái rảnh rỗi (phần proc được gán 0 trong trường hợp này) thì sẽ được cấp phát (gán) cho process yêu cầu cấp phát.

   (b) Trong mỗi lần cấp phát thì cập nhật giá trị next trong để giúp cho máy ảo truy xuất như là một vùng liên tiếp (sử dụng prevIdx (với khởi trị là -1) để giữ vị trí của vùng vừa được cấp phát).

   (c) Trên vùng nhớ luận lý: ta dựa trên địa chỉ cấp phát để tính vị trí segment, page. Rồi cập nhật bảng phân trang, phân đoạn tương ứng.

```
if (mem_avail) {
    ret_mem = proc->bp;
    proc->bp += num_pages * PAGE_SIZE;
    int count_alloc = 0; // count allocated pages
    int last_allocated_page_index = -1; // keep the lastest allocated
    block
    int i;
    for (i = 0; i < NUM_PAGES; i++) {
      if (_mem_stat[i].proc) continue;

      _mem_stat[i].proc = proc->pid;
      _mem_stat[i].index = count_alloc;

      if (last_allocated_page_index > -1) {
        _mem_stat[last_allocated_page_index].next = i;
      }
      last_allocated_page_index = i;

      addr_t v_address = ret_mem + count_alloc * PAGE_SIZE;
      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;
        proc->seg_table->table[idx].pages
            = (struct page_table_t*) malloc(sizeof(struct page_table_t))
    ;
        v_page_table = proc->seg_table->table[idx].pages; // allocate
    real memory for proc
        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;
      if (++count_alloc == num_pages) {
        _mem_stat[i].next = -1;
        break;
      }
    }
  }
```

2.5 Thu hồi bộ nhớ

2.5.1 Thu hồi địa chỉ vật lý

Đầu tiên chúng ta chuyển địa chỉ luận lý từ process thành địa chỉ vật lý. Sau đó dựa vào giá trị *next* trong *_mem_stat* của trang tương ứng chúng ta cập nhật lại giá trị *proc* bằng 0 ứng với việc không có process nào đang sử dụng trang đó.

```
int free_mem(addr_t address, struct pcb_t * proc)
{
//...
    addr_t virtual_addr = address;
  addr_t physical_addr = 0;

  if (!translate(virtual_addr, &physical_addr, proc))
    return 1;

  addr_t physical_segment_page_index = physical_addr >> OFFSET_LEN;
  int num_pages = 0;
  int i;
  for (i = physical_segment_page_index; i != -1; i = _mem_stat[i].next)
  {
    num_pages++;
    _mem_stat[i].proc = 0;
  }
//...
}
```

2.5.2 Cập nhật địa chỉ luận lý

Dựa trên số lượng trang đã xóa trên block của địa chỉ vật lý, chúng ta tìm lần lượt các trang trên địa chỉ luận lý. Dựa trên địa chỉ, chúng ta tìm được segment và page tương ứng. Sau đó cập nhật lại bảng phân trang, nếu bảng nào trống thì xóa bảng đó 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;
}
```

```
int free_mem(addr_t address, struct pcb_t * proc) {
//...
    for (i = 0; i < num_pages; i++)
  {
    addr_t v_addr = virtual_addr + 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)
    {
      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);
    }
  }
//...
}
```

2.5.3 Cập nhật break point

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

```
int free_mem(addr_t address, struct pcb_t * proc)
{
//...
    if (virtual_addr + num_pages * PAGE_SIZE == proc->bp)
    {
      free_break_point(proc);
    }
//...
}
```

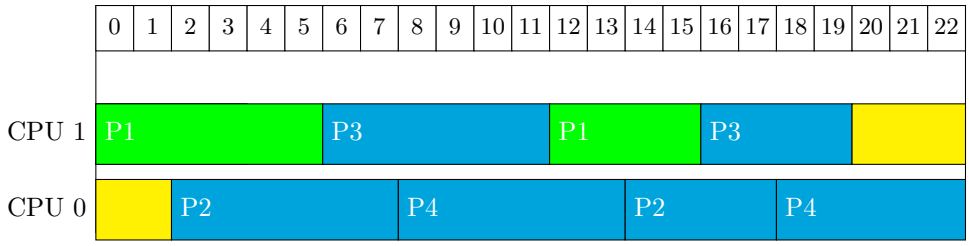
```
void free_break_point(struct pcb_t *proc)
{
  while (proc->bp >= PAGE_SIZE)
  {
    addr_t last_addr = proc->bp - PAGE_SIZE;
    addr_t last_segment = get_first_lv(last_addr);
    addr_t last_page = get_second_lv(last_addr);
    struct page_table_t *page_table = get_page_table(last_segment, proc->seg_table
    );
    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;
        }
```

```
22        }
23        if (i == page_table->size)
24        {
25          break;
26        }
27      }
28      if (last_page >= 0)
29      {
30        break;
31      }
32    }
33  }
```

3 Kết hợp tất cả

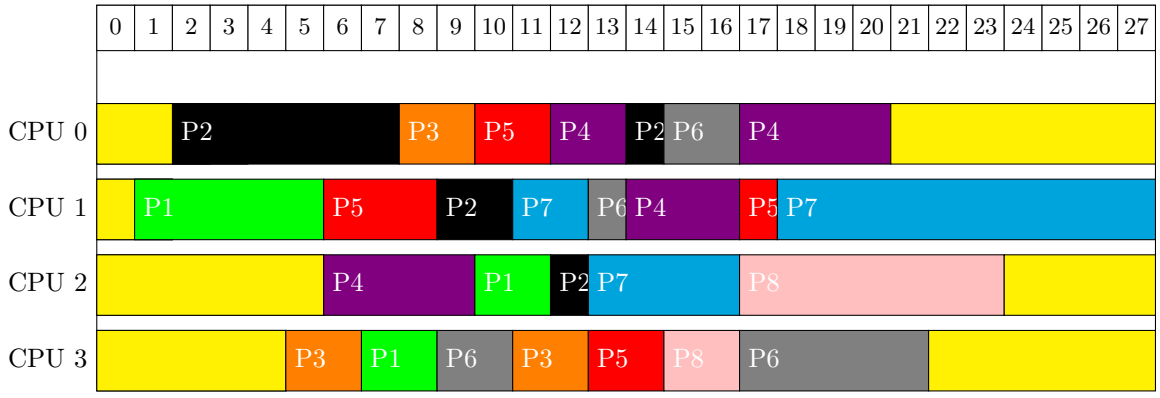
Thực hiện make all sau khi hoàn thành Scheduler và Memory Management

3.1 Test 0



Hình 3: Lược đồ Gantt thực thi các process cho make all

3.2 Test 1



Hình 4: Lược đồ Gantt thực thi các process cho make all