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: La Hoàng Lộc
Sinh viên: Phạm Minh Duy - 1912916
Nguyễn Đình Hiếu - 1913341
Nguyễn Hải Linh - 1913944
Nguyễn Thế Huy - 1812404

Thành phố Hồ Chí Minh, 6/2021

Mục lục

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?

Answer:
Giải thuật Priority Feedback Queue (PFQ) sử dụng tư tưởng của một số giải thuật khác gồm giải 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)

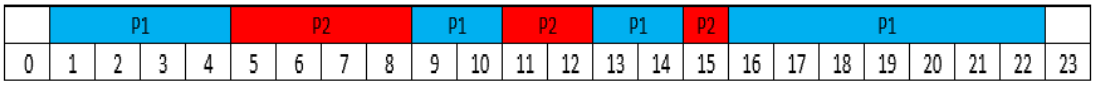
- Tính công bằng giữa các process là được đảm bảo, chỉ phụ thuộc vào độ ưu tiên có sẵn của các process. Cụ thể xét trong khoảng thời gian t0 nào đó, nếu các process đang thực thi thì hoàn toàn phụ thuộc vào độ ưu tiên của chúng. Nếu có 1 process p0 khác đến, giả sử ready_queue đang sẵn sàng, process 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 ready_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.
Text 0:

```
------ SCHEDULING TEST 0 ------------------------------------------
./os sched_0
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  10
Time slot  11
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  14
Time slot  15
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot  16
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  17
Time slot  18
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  19
Time slot  20
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  21
Time slot  22
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  23
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Hình 1: *Kết quả chạy test 0*

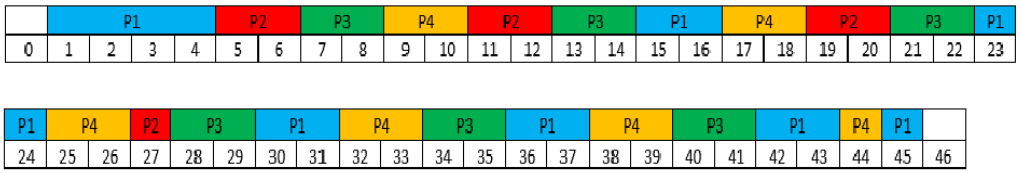


Hình 2: *Lược đồ Gantt CPU thực thi các processes - test 0*

Text 1:

```
------ SCHEDULING TEST 1 ------------------------------------
./os sched_1
Time slot   0
        Loaded a process at input/proc/s0, PID: 1
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   6
        Loaded a process at input/proc/s2, PID: 3
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
        Loaded a process at input/proc/s3, PID: 4
Time slot   8
Time slot   9
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  10
Time slot  11
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  14
Time slot  15
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  16
Time slot  17
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  18
Time slot  19
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  20
Time slot  21
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  22
Time slot  23
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  24
Time slot  25
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  26
Time slot  27
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  28
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
Time slot  29
Time slot  30
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  31
Time slot  32
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  33
Time slot  34
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  35
Time slot  36
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  37
Time slot  38
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  39
Time slot  40
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  41
Time slot  42
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  1
Time slot  43
Time slot  44
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  45
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  46
        CPU 0: Processed  1 has finished
        CPU 0 stopped

MEMORY CONTENT:
NOTE: Read file output/sched_1 to verify your result
```

Hình 3: *Kết quả chạy test 1*



Hình 4: *Lược đồ Gantt CPU thực thi các processes - test 1*

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:

```
#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) // Check if queue is full or not
    return;
  q->proc[q->size++] = proc;
}

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) // first check if ready queue is empty
    return NULL;

  // If ready queue is not null
  int highest_pri_index = 0; // Highest Priority Proc's Index
  int pWalker; // Break point checking where is the highest priority proc's index
  for (pWalker = 1; pWalker < q->size; pWalker++)
  {
    if (q->proc[pWalker]->priority >
              q->proc[highest_pri_index]->priority)
    {
      highest_pri_index = pWalker;
    }
  }
```

```
36   struct pcb_t *deq_proc = q->proc[highest_pri_index]; // dequeue proc
37   // shifting queue back
38   for (pWalker = highest_pri_index + 1; pWalker < q->size; pWalker++)
39   {
40     q->proc[pWalker - 1] = q->proc[pWalker];
41   }
42   q->size--;
43
44   return deq_proc;
45 }
```

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 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 hàm get_proc() :

```
1
2  #include "queue.h"
3  #include "sched.h"
4  #include <pthread.h>
5
6  static struct queue_t ready_queue;
7  static struct queue_t run_queue;
8  static pthread_mutex_t queue_lock;
9
10 int queue_empty(void) {
11   return (empty(&ready_queue) && empty(&run_queue));
12 }
13
14 void init_scheduler(void) {
15   ready_queue.size = 0;
16   run_queue.size = 0;
17   pthread_mutex_init(&queue_lock, NULL);
18 }
19
20 struct pcb_t * get_proc(void) {
21    struct pcb_t * proc = NULL;
22   /*TODO: get a process from [ready_queue]. If ready queue
23    * is empty, push all processes in [run_queue] back to
24    * [ready_queue] and return the highest priority one.
25    * Remember to use lock to protect the queue.
26    * */
27
28    pthread_mutex_lock(&queue_lock);
29    if (empty(&ready_queue)) { // check if ready queue is empty
30      // move all process is waiting in run_queue back to ready_queue
31      while (!empty(&run_queue)) {
32        enqueue(&ready_queue, dequeue(&run_queue));
33      }
34    }
35
36    if (!empty(&ready_queue)) {
37      proc = dequeue(&ready_queue);
38    }
```

```
39      pthread_mutex_unlock(&queue_lock);
40
41      return proc;
42  }
43
44  void put_proc(struct pcb_t * proc) {
45    pthread_mutex_lock(&queue_lock);
46    enqueue(&run_queue, proc);
47    pthread_mutex_unlock(&queue_lock);
48  }
49
50  void add_proc(struct pcb_t * proc) {
51    pthread_mutex_lock(&queue_lock);
52    enqueue(&ready_queue, proc);
53    pthread_mutex_unlock(&queue_lock);
54  }
```

2 Memory Management.

2.1 Question - Segmentation with Paging.

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

Answer:

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 và 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:

```
1  ============= Allocation ============
2  000: 00000-003ff - PID: 01 (idx 000, nxt: 001)
3  001: 00400-007ff - PID: 01 (idx 001, nxt: 002)
4  002: 00800-00bff - PID: 01 (idx 002, nxt: 003)
5  003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)
6  004: 01000-013ff - PID: 01 (idx 004, nxt: 005)
7  005: 01400-017ff - PID: 01 (idx 005, nxt: 006)
8  006: 01800-01bff - PID: 01 (idx 006, nxt: 007)
9  007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)
10 008: 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)
============ Allocation ============
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)
============ Deallocation ============
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,   // 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
   *
   * */

  int i;
  for (i = 0; i < seg_table->size; i++) {
    if (index == seg_table->table[i].v_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,  // 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 Cấp phát memory.

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.
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 tiên:

```
addr_t alloc_mem(uint32_t size, struct pcb_t *proc)
{
    pthread_mutex_lock(&mem_lock);
    addr_t ret_mem = 0;
    /* TODO: Allocate [size] byte in the memory for the
    * process [proc] and save the address of the first
    * byte in the allocated memory region to [ret_mem].
    * */

    uint32_t num_pages = ((size % PAGE_SIZE)==0)
                            ? size / PAGE_SIZE
                            : size / PAGE_SIZE + 1;
    int mem_avail = 0;
    /* 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. If so, set 1 to [mem_avail].
    * Hint: check [proc] bit in each page of _mem_stat
    * to know whether this page has been used by a process.
    * For virtual memory space, check bp (break pointer).
    * */

    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 <= RAM_SIZE)
    {
        mem_avail = 1;
    }
    else
        ret_mem = 0;
    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 use_page_number = 0;
        struct page_table_t * page_tbl;
        uint32_t index_in_list_page = 0;
        uint32_t last_page_allocated = 0;
        for (int i = 0; i < NUM_PAGES; i++)
        {
            if (_mem_stat[i].proc == 0)
            {
                _mem_stat[i].proc = proc->pid;
                _mem_stat[i].index = index_in_list_page;
                if (index_in_list_page != 0)
                    _mem_stat[last_page_allocated].next = i;

                addr_t physical_addr = i << OFFSET_LEN;
                addr_t v_addr = ret_mem + index_in_list_page * PAGE_SIZE;
                addr_t first_lv = get_first_lv(v_addr);
                addr_t second_lv = get_second_lv(v_addr);
                int have_first_index = 0;
                for (int n = 0; n < proc->seg_table->size; n++)
                {
                    if (proc->seg_table->table[n].v_index == first_lv)
                    {
                        page_tbl = proc->seg_table->table[n].pages;
                        page_tbl->table[page_tbl->size].v_index = second_lv;
                        page_tbl->table[page_tbl->size].p_index =
                              physical_addr >> OFFSET_LEN;
                        page_tbl->size++;
                        have_first_index = 1;
                        break;
                    }
```

```
                }
                if (have_first_index == 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));
                    page_tbl = proc->seg_table->table[n].pages;
                    page_tbl->size++;
                    proc->seg_table->table[n].v_index = first_lv;
                    page_tbl->table[0].v_index = second_lv;
                    page_tbl->table[0].p_index = physical_addr >> OFFSET_LEN;
                }
                last_page_allocated = i;
                index_in_list_page++;
                use_page_number++;
                if (use_page_number == num_pages)
                {
                    _mem_stat[last_page_allocated].next = -1;
                    break;
                }
            }
        }
    }
    //printf("-----------------Allocation-----------------\n");
    //dump();
    pthread_mutex_unlock(&mem_lock);
    return ret_mem;
}
```

2.3.4 Thu hồi memory

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.
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.
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.
Dưới đây là phần hiện thực chi tiên:

```
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);
    addr_t physical_addr;
    if (translate(address, &physical_addr, proc)) // find physical address
    {
        int next = -1;
        int i = 0;
```

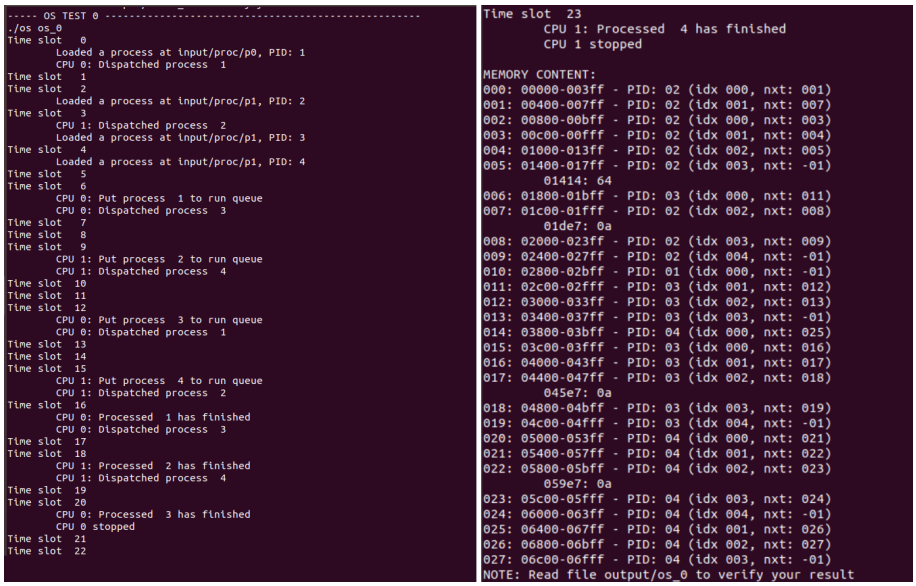
```
        for (; i < NUM_PAGES; i++)
        {
            if (physical_addr == i << OFFSET_LEN)
            {
                break;
            }
        }
        next = i;
    int del_pages = 0;
    addr_t v_address;
    struct page_table_t * page_tbl;
        while (next != -1) // delete pages
        {
            _mem_stat[next].proc = 0;
            next = _mem_stat[next].next; // next page index
            v_address = address + del_pages * PAGE_SIZE;
            addr_t first_lv = get_first_lv(v_address); // segment
            addr_t second_lv = get_second_lv(v_address); // page
            for (int n = 0; n < proc->seg_table->size; n++) // finding segment
            {
                if (proc->seg_table->table[n].v_index == first_lv)
                {
                    page_tbl = proc->seg_table->table[n].pages;
                    for (int m = 0; m < page_tbl->size; m++)
                    {
                        if (page_tbl->table[m].v_index == second_lv)
                        {
                            // xoa page
                            int k = 0;
             del_pages++;
                            for (k = m; k < page_tbl->size - 1; k++)
                            {
                                page_tbl->table[k].v_index =
                                  page_tbl->table[k + 1].v_index;
                                page_tbl->table[k].p_index =
                                  page_tbl->table[k + 1].p_index;
                            }

                            page_tbl->table[k].v_index = 0;
                            page_tbl->table[k].p_index = 0;
                            page_tbl->size--;
                            break;
                        }
                    }
                    if (page_tbl->size == 0)//xoa segment khi size = 0
                    {
                        free(page_tbl);
                        int m = 0;
                        //tinh tien phan tu trong seg_table
                        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;
                }
```

```
            }
        }
      proc->bp -= del_pages*PAGE_SIZE; // cap nhat break point
    }
    //printf("-----------------Free memory-----------------\n");
    //dump();
    pthread_mutex_unlock(&mem_lock);
    return 0;
}
```

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

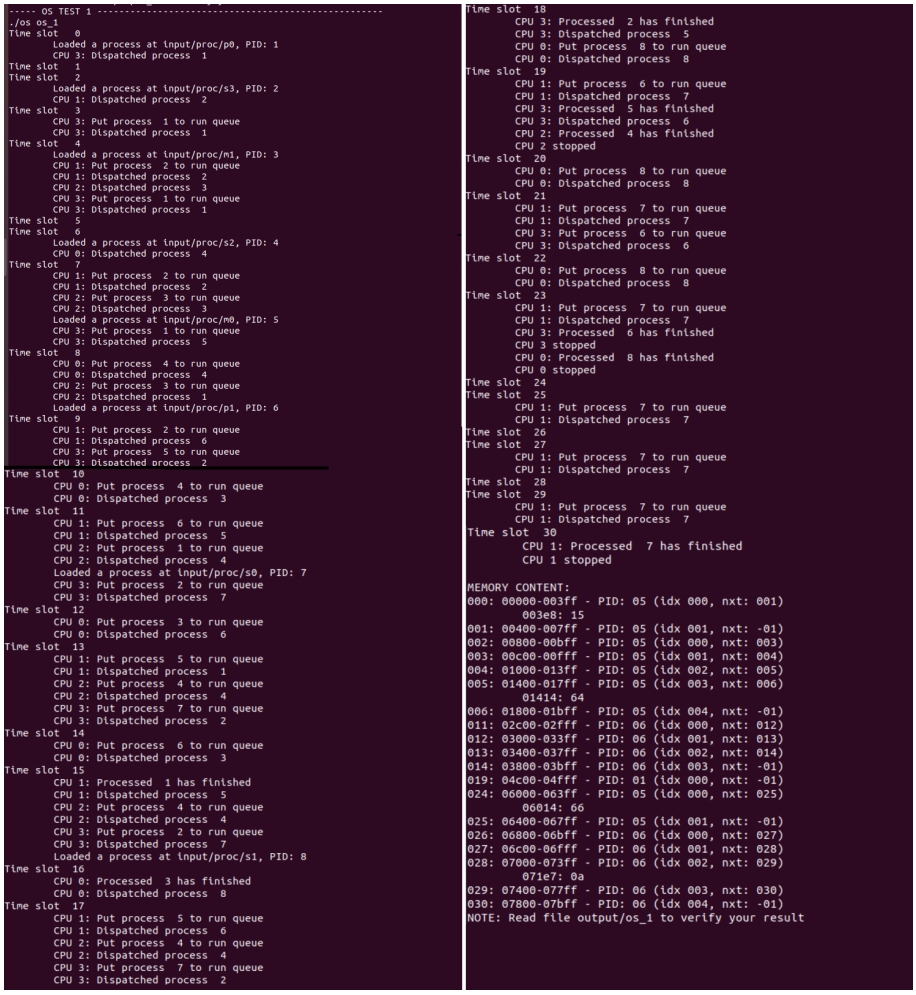
Test 0:



Hình 5: *Kết quả chạy Test 0*

Hình 6: *Lược đồ Gantt CPU thực thi các processes cho make all (2 CPU)*

Test 1:



Hình 7: *Kết quả chạy Test 1*

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CPU 0	IDLE						P4				P3		P6		P3	
CPU 1	IDLE		P2							P6		P5		P1		P5
CPU 2	IDLE				P3				P1			P4				
CPU 3	P1							P5		P2		P7		P2		P7

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
CPU 0	P8							STOPPED							
CPU 1	P5	P6		P7											STOPPED
CPU 2	P4			STOPPED											
CPU 3	P7	P2	P5	P6				STOPPED							

Hình 8: *Lược đồ Gantt CPU thực thi các processes cho make all (4 CPU)*