ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA

Assignment 2
Simple Operating System

Tutor:	Trần Ngọc Anh Tú	
Class:	L08	
Student members:	Nguyễn Hữu Thắng	1713239
	Võ Đình Khương	1711835
	Đặng Hữu Thiên	1713269
	Nguyễn Đình Thuần	1713375

Ngày 22 tháng 5 năm 2019

Mục lục

1 Scheduler

1.1 Question - Priority Feedback Queue

QUESTION: Trình bày lợi ích của giải thuật Priority Feedback Queue (PFQ) với các giải thuật lập lịch đã được học ?

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.

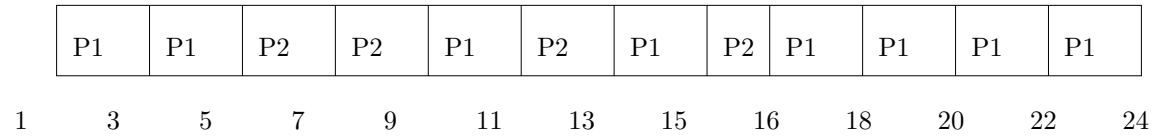
Ưu điểm của giải thuật PFQ

- 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

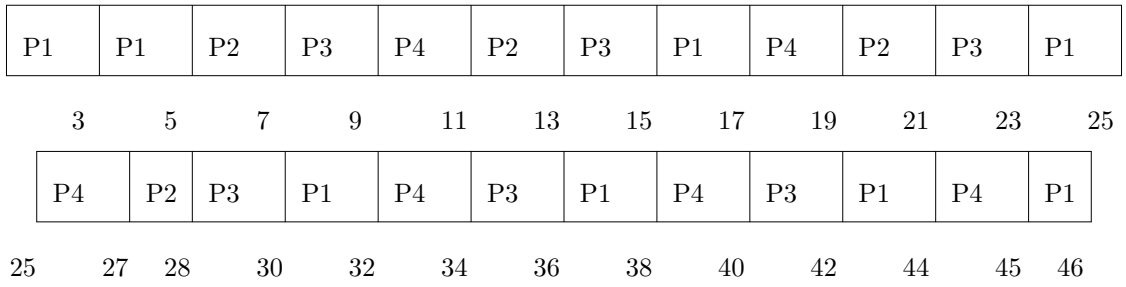
REQUIREMENT: Draw Gantt diagram describing how processes are executed by the CPU.

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:



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

Để hiện thực priority queue, nhóm chúng em chỉ đơn giản sử dụng vòng lặp để xử lí các chức năng của hàng đợi ưu tiên vì trong trường hợp này có ít process. Trường hợp nhiều process hơn chúng em sẽ thực hiện bằng binary heap để đạt tốc độ cao hơn.

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


struct pcb_t * dequeue(struct queue_t * q) {
  // If queue empty
  if(empty(q))
    return NULL;

  // If queue not empty
  unsigned int priority = q->proc[0]->priority;
  unsigned int index = 0;
  // Find element hold highest priotiry
  for (int i = 1; i < q->size; ++i){
    if (q->proc[i]->priority > priority){
      priority = q->proc[i]->priority;
      index = i;
    }
  }
  struct pcb_t * result = q->proc[index];

  for (int i = index + 1; i < q->size; ++i){
    q->proc[i - 1] = q->proc[i];
  }
  q->size--;

  return result;
}
```

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 của chức năng nói trên.

```
pcb_t* get_proc(void) {
  pcb_t * proc = NULL;

  pthread_mutex_lock(&queue_lock);

  if(empty(&ready_queue))
  while(!empty(&run_queue)) {
    proc = dequeue(&run_queue);
    enqueue(&ready_queue, proc);
  }

  proc = dequeue(&ready_queue);

  pthread_mutex_unlock(&queue_lock);

  return proc;
}
```

2 Memory Management

2.1 Question - Segmentation with Paging

Câu hỏi: Ưu điểm và nhược điểm của kết hợp phân trang và phân đoạn ?

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.

Test m0

```
------- MEMORY MANAGEMENT TEST 0 ----------------------------------------
./mem input/proc/m0
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)
NOTE: Read file output/m0 to verify your result
```

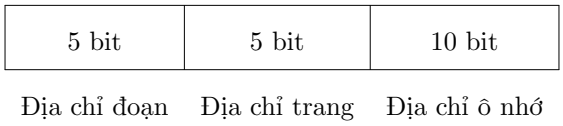
Test m1

```
------- MEMORY MANAGEMENT TEST 1 ----------------------------------------
./mem input/proc/m1
NOTE: Read file output/m1 to verify your result (your implementation should print ↩
    nothing)
```

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 :



Bảng phân đoạn *seg_table* là một danh sách gồm các phần tử u có cấu trúc như sau :

- *v_index* là 5 bits segment của phần tử u.
- *page_table_t* là bảng phần trang tương ứng của segment đó.

Chức năng này nhận vào 5 bits segment *index* và bảng phân đoạn *seg_table*, để tìm ra bảng phân trang *res*.
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.

```
1  static page_table_t* get_page_table(
2    addr_t index,        // segment level index
3    seg_table_t* seg_table) // first level table
4  {
5    if(seg_table == NULL)
6      return NULL;
7
8    for(int i = 0; i < seg_table->size; ++i)
9      if(seg_table->table[i].v_index == index)
10       return seg_table->table[i].pages;
11
12   return NULL;
13 }
```

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.

```
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 */
  page_table_t* page_table = NULL;
  page_table = get_page_table(first_lv, proc->seg_table);
  if (page_table == NULL) return 0;

  for (int 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_addr = (p_index << OFFSET_LEN) | offset;
      return 1;
  }

  return 0;
}
```

2.3.3 Cấp phát memory

2.3.3.a 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.
- Trên vùng nhớ luận lý, ta kiểm tra dựa trên break point của process, không vượt quá vùng nhớ cho phép.

```
1  bool isMemAvail(int num_pages, struct pcb_t * proc) {
2    // check physical space
3    unsigned int empty_page = 0;
4
5    for(int i = 0; i < NUM_PAGES; ++i)
6      if(_mem_stat[i].proc == 0)
7        if(++empty_page == num_pages)
8          break; // enough memory
9
10   if(empty_page < num_pages)
11     return false;
12
13   if(proc->bp + num_pages*PAGE_SIZE >= RAM_SIZE)
14     return false; // check virtual space
15
16   return true;
17 }
```

2.3.3.b 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 allocMemAvail(int ret_mem, int num_pages, pcb_t* proc) {
  int count_alloc_pages = 0;
  int last_alloc_page_index = -1;

  pthread_mutex_lock(&mem_lock);
  for(int i = 0; i < NUM_PAGES; ++i) {
    if(_mem_stat[i].proc != 0) continue; // page is used

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

    // if not initial page, update "next" field
    if(last_alloc_page_index > -1)
      _mem_stat[last_alloc_page_index].next = i;
    // update last page index
    last_alloc_page_index = i;

    addr_t virtual_addr = ret_mem + count_alloc_pages*PAGE_SIZE;
    addr_t first_lv     = get_first_lv(virtual_addr);
    addr_t second_lv    = get_second_lv(virtual_addr);

    page_table_t* page_table = get_page_table(first_lv, proc->seg_table);

    if(page_table == NULL) {
      int idx = proc->seg_table->size++;
      proc->seg_table->table[idx].v_index = first_lv;
      page_table
        = proc->seg_table->table[idx].pages
        = (page_table_t*)malloc(sizeof(page_table_t));
      page_table->size = 0;
    }

    int idx = page_table->size++;
    page_table->table[idx].v_index = second_lv;
    page_table->table[idx].p_index = i;

    if(++count_alloc_pages == num_pages)
      {_mem_stat[i].next = -1; break;}
  }
  pthread_mutex_unlock(&mem_lock);
}
```

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.

```
...
// find physical page in memory
if(!translate(virtual_addr, &physical_addr, proc)) return 1;

// clear physical page in memory
addr_t p_segm_page_index = physical_addr >> OFFSET_LEN;
int num_pages = 0;

for(int i = p_segm_page_index; i != -1; i = _mem_stat[i].next) {
  _mem_stat[i].proc = 0;     // clear physical memory
  num_pages = num_pages + 1;  // count pages
}
...
```

2.3.5 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.

```
...
  for(int i = 0; i < num_pages; ++i) {
    addr_t v_page_addr = virtual_addr + i*PAGE_SIZE;
    addr_t first_lv  = get_first_lv(v_page_addr);
    addr_t second_lv = get_second_lv(v_page_addr);

    page_table_t * page_table = get_page_table(first_lv, proc->seg_table);
    if(page_table == NULL) {puts("Error\n"); continue;}

    for(int j = 0; j < page_table->size; ++j)
      if(page_table->table[j].v_index == second_lv) {
        int last = --page_table->size;
        page_table->table[j] = page_table->table[last];
        break;
      }

    if(page_table->size == 0)
      remove_page_table(first_lv, proc->seg_table);
  }
  proc->bp = proc->bp - num_pages*PAGE_SIZE;
...
```

```
static int remove_page_table(
  addr_t index,       // segment level index
  seg_table_t* seg_table) // first level table
{
  if(seg_table == NULL) return 0;

  for(int i = 0; i < seg_table->size; ++i)
  if(seg_table->table[i].v_index == index) {
    int idx = --seg_table->size;
    seg_table->table[i] = seg_table->table[idx];
    seg_table->table[idx].v_index = 0;
    free(seg_table->table[idx].pages);
    return 1;
  }
  return 0;
}
```

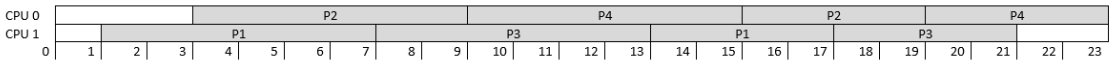
3 Put it all together

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
Dưới đây là giản đồ Gantt cho trường hợp trong log/all1.txt trong source code.
Test 0

```
------ OS TEST 0 ------------------------------------------------------
./os os_0
Time slot   0
         Loaded a process at input/proc/p0, PID: 1
Time slot   1
         CPU 1: Dispatched process  1
Time slot   2
         Loaded a process at input/proc/p1, PID: 2
Time slot   3
         CPU 0: Dispatched process  2
         Loaded a process at input/proc/p1, PID: 3
Time slot   4
         Loaded a process at input/proc/p1, PID: 4
Time slot   5
Time slot   6
Time slot   7
         CPU 1: Put process  1 to run queue
         CPU 1: Dispatched process  3
Time slot   8
Time slot   9
         CPU 0: Put process  2 to run queue
         CPU 0: Dispatched process  4
Time slot  10
Time slot  11
Time slot  12
Time slot  13
         CPU 1: Put process  3 to run queue
         CPU 1: Dispatched process  1
Time slot  14
Time slot  15
         CPU 0: Put process  4 to run queue
         CPU 0: Dispatched process  2
Time slot  16
Time slot  17
         CPU 1: Processed  1 has finished
         CPU 1: Dispatched process  3
Time slot  18
Time slot  19
         CPU 0: Processed  2 has finished
         CPU 0: Dispatched process  4
Time slot  20
Time slot  21
         CPU 1: Processed  3 has finished
         CPU 1 stopped
Time slot  22
Time slot  23
         CPU 0: Processed  4 has finished
         CPU 0 stopped

MEMORY CONTENT:
000: 00000-003ff - PID: 01 (idx 000, nxt: -01)
001: 00400-007ff - PID: 03 (idx 000, nxt: 006)
002: 00800-00bff - PID: 02 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 02 (idx 001, nxt: 004)
004: 01000-013ff - PID: 02 (idx 002, nxt: 005)
005: 01400-017ff - PID: 02 (idx 003, nxt: -01)
         01414: 64
006: 01800-01bff - PID: 03 (idx 001, nxt: 012)
007: 01c00-01fff - PID: 02 (idx 000, nxt: 008)
008: 02000-023ff - PID: 02 (idx 001, nxt: 009)
009: 02400-027ff - PID: 02 (idx 002, nxt: 010)
         025e7: 0a
010: 02800-02bff - PID: 02 (idx 003, nxt: 011)
011: 02c00-02fff - PID: 02 (idx 004, nxt: -01)
```

```
012: 03000-033ff - PID: 03 (idx 002, nxt: 013)
013: 03400-037ff - PID: 03 (idx 003, nxt: -01)
014: 03800-03bff - PID: 04 (idx 000, nxt: 025)
015: 03c00-03fff - PID: 03 (idx 000, nxt: 016)
016: 04000-043ff - PID: 03 (idx 001, nxt: 017)
017: 04400-047ff - PID: 03 (idx 002, nxt: 018)
        045e7: 0a
018: 04800-04bff - PID: 03 (idx 003, nxt: 019)
019: 04c00-04fff - PID: 03 (idx 004, nxt: -01)
020: 05000-053ff - PID: 04 (idx 000, nxt: 021)
021: 05400-057ff - PID: 04 (idx 001, nxt: 022)
022: 05800-05bff - PID: 04 (idx 002, nxt: 023)
        059e7: 0a
023: 05c00-05fff - PID: 04 (idx 003, nxt: 024)
024: 06000-063ff - PID: 04 (idx 004, nxt: -01)
025: 06400-067ff - PID: 04 (idx 001, nxt: 026)
026: 06800-06bff - PID: 04 (idx 002, nxt: 027)
027: 06c00-06fff - PID: 04 (idx 003, nxt: -01)
NOTE: Read file output/os_0 to verify your result
```



Hình 1: *Lược đồ Gantt CPU thực thi các processes cho make all*

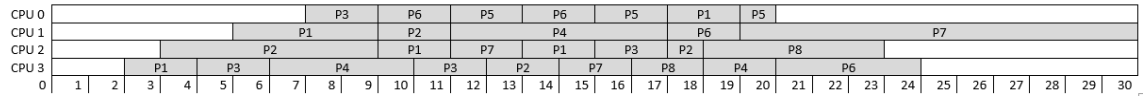
Test 1

```
------ OS TEST 1 ------------------------------------------------------
./os os_1
Time slot   0
Time slot   1
        Loaded a process at input/proc/p0, PID: 1
Time slot   2
        CPU 3: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 2
Time slot   3
        CPU 2: Dispatched process  2
Time slot   4
        Loaded a process at input/proc/m1, PID: 3
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  3
Time slot   5
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 1: Dispatched process  1
Time slot   6
        Loaded a process at input/proc/s2, PID: 4
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  4
Time slot   7
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 0: Dispatched process  3
        Loaded a process at input/proc/m0, PID: 5
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  5
Time slot   8
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  4
        Loaded a process at input/proc/p1, PID: 6
Time slot   9
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  1
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  6
        CPU 1: Put process  5 to run queue
        CPU 1: Dispatched process  2
Time slot  10
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  3
Time slot  11
```

```
        Loaded a process at input/proc/s0, PID: 7
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  7
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  5
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  4
Time slot  12
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  2
Time slot  13
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  1
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  6
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
Time slot  14
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  7
Time slot  15
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  3
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  5
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  4
Time slot  16
        Loaded a process at input/proc/s1, PID: 8
        CPU 3: Put process  7 to run queue
        CPU 3: Dispatched process  8
Time slot  17
        CPU 2: Processed  3 has finished
        CPU 2: Dispatched process  2
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  1
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  6
Time slot  18
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  4
        CPU 2: Processed  2 has finished
        CPU 2: Dispatched process  8
Time slot  19
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  7
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  5
Time slot  20
        CPU 3: Processed  4 has finished
        CPU 3: Dispatched process  6
        CPU 0: Processed  5 has finished
        CPU 0 stopped
        CPU 2: Put process  8 to run queue
        CPU 2: Dispatched process  8
Time slot  21
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  22
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
        CPU 2: Put process  8 to run queue
        CPU 2: Dispatched process  8
Time slot  23
        CPU 2: Processed  8 has finished
        CPU 2 stopped
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  24
        CPU 3: Processed  6 has finished
        CPU 3 stopped
Time slot  25
        CPU 1: Put process  7 to run queue
```

```
        CPU 1: Dispatched process  7
Time slot  26
Time slot  27
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  28
Time slot  29
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  30
        CPU 1: Processed  7 has finished
        CPU 1 stopped

MEMORY CONTENT:
000: 00000-003ff - PID: 05 (idx 000, nxt: 001)
        003e8: 15
001: 00400-007ff - PID: 05 (idx 001, nxt: -01)
002: 00800-00bff - PID: 05 (idx 000, nxt: 003)
003: 00c00-00fff - PID: 05 (idx 001, nxt: 004)
004: 01000-013ff - PID: 05 (idx 002, nxt: 005)
005: 01400-017ff - PID: 05 (idx 003, nxt: 006)
006: 01800-01bff - PID: 05 (idx 004, nxt: -01)
011: 02c00-02fff - PID: 06 (idx 000, nxt: 012)
012: 03000-033ff - PID: 06 (idx 001, nxt: 013)
013: 03400-037ff - PID: 06 (idx 002, nxt: 016)
014: 03800-03bff - PID: 05 (idx 000, nxt: 015)
        03814: 66
015: 03c00-03fff - PID: 05 (idx 001, nxt: -01)
016: 04000-043ff - PID: 06 (idx 003, nxt: -01)
025: 06400-067ff - PID: 01 (idx 000, nxt: -01)
026: 06800-06bff - PID: 06 (idx 000, nxt: 027)
027: 06c00-06fff - PID: 06 (idx 001, nxt: 028)
028: 07000-073ff - PID: 06 (idx 002, nxt: 029)
        071e7: 0a
029: 07400-077ff - PID: 06 (idx 003, nxt: 030)
030: 07800-07bff - PID: 06 (idx 004, nxt: -01)
NOTE: Read file output/os_1 to verify your result
```



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

Tài liệu

[1] Wikipedia. http://en.wikipedia.org, last access: 20/05/2019.

[2] Silberschatz, Galvin, and Gagne, Operating System Concepts.

[3] Tanenbaum, Modern Operating Systems.