VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEM (CO2017)

Report

Simple Operating System

Advisor: **Nguyễn Thanh Quân**
Students: **Bùi Đức Anh** - **2112751**
Nguyễn Sỹ Dương - **2113097**
Cao Đức Vinh - **2115290**

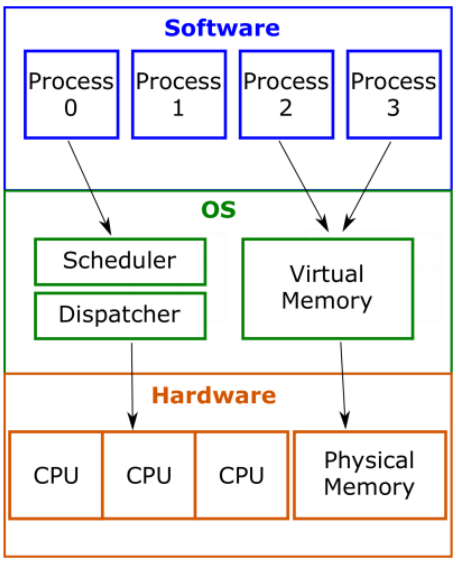
HO CHI MINH CITY, 2023

Contents

1 Lời mở đầu

Mục đích của bài tập lớn là về mô phỏng một hệ điều hành đơn giản (**Simple OS**), qua đó giúp sinh viên hiểu được những khái niệm cơ bản về định thời (**scheduling**), đồng bộ (**synchronization**) và quản lý bộ nhớ (**memory management**). Hình 1 thể hiện kiến trúc tổng quan của một hệ điều hành mà chúng ta sẽ hiện thực. Về tổng quan, hệ điều hành phải quản lý hai nguồn tài nguyên ảo: CPU(s) và RAM qua việc sử dụng hai thành phần chính:

- **Scheduler** (và Dispatcher): quyết định quá trình (process) nào được chạy trên CPU
- **Virtual Memory Engine** (VME): cô lập không gian bộ nhớ của từng quá trình. RAM vật lý được chia sẻ bởi nhiều quá trình nhưng mỗi quá trình sẽ bỏ qua sự tồn tại của các quá trình khác. Để làm điều này, mỗi quá trình sẽ có vùng nhớ ảo riêng, VME giúp ánh xạ và dịch địa chỉ ảo cung cấp bởi các quá trình thành địa chỉ vật lý tương ứng.



Hình 1: Các module chính

Qua các module trên, hệ điều hành cho phép nhiều quá trình được tạo bởi người dùng để chia sẻ và sử dụng tài nguyên máy tính ảo. Cho nên, trong bài tập lớn này, ta sẽ tập trung vào hiện thực scheduler/dispatcher và VME.

2 Giải thích input process

Trong bài tập lớn này, ngoài việc phải quan tâm đến các giải thuật thì chúng ta cũng cần quan tâm đến các process input đầu vào và các mô tả của nó trong hệ điều hành giả lập được tạo ra.

- **m0s**

```
1 6
alloc 300 0
alloc 100 1
free 0
alloc 100 2
write 102 1 20
write 1 2 1000
```

Process này thực hiện 6 lệnh như mô tả, và số timeslot cần chạy để thực hiện process là 6.

- **m1s**

```
1 6
alloc 300 0
alloc 100 1
free 0
alloc 100 2
free 2
free 1
```

Process này thực hiện 6 lệnh như mô tả, và số timeslot cần chạy để thực hiện process là 6.

- **p0s**

```
1 14
calc
alloc 300 0
alloc 300 4
free 0
alloc 100 1
write 100 1 20
read 1 20 20
write 102 2 20
read 2 20 20
write 103 3 20
read 3 20 20
calc
free 4
calc
```

Process này thực hiện 14 lệnh như mô tả, và số timeslot cần chạy để thực hiện process là 14.

- **p1s**

```
1 10
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc
```

  Process này thực hiện 10 lệnh calc như mô tả, và số timeslot cần chạy để thực hiện process là 10.

Ta có thể hiểu tương tự với các input process còn lại như **p2s, p3s, s0, s1, s2, s3, s4.**

3 Scheduler

3.1 Cơ sở lý thuyết

Trong bài tập lớn này, chúng ta sẽ sử dụng **giải thuật MLQ** (Multi Level Queue) để xác định tiến trình (process) sẽ được thực hiện trong quá trình định thời.

Sơ lược về giải thuật MLQ: là một giải thuật định thời CPU trong đó các tiến trình được phân thành nhiều hàng đợi khác nhau dựa trên mức độ ưu tiên của chúng. Các tiến trình ưu tiên cao được đặt trong các hàng đợi có mức độ ưu tiên cao hơn, trong khi các tiến trình ưu tiên thấp được đặt trong các hàng đợi có mức độ ưu tiên thấp hơn.

Các đặc trưng của giải thuật MLQ:

- Ready queue được chia thành nhiều hàng đợi riêng biệt theo một số tiêu chuẩn như:
  - Đặc điểm và yêu cầu định thời của tiến trình (process)
  - Foreground (interactive) và background process,...
- Process được gán cố định vào một hàng đợi, mỗi hàng đợi sử dụng giải thuật định thời riêng
- Hệ điều hành cần phải định thời cho các hàng đợi.
  - Fixed priority scheduling: phục vụ từ hàng đợi có độ ưu tiên cao đến thấp. Vấn đề: có thể có starvation
  - Time slice: mỗi hàng đợi được nhận một khoảng thời gian chiếm CPU và phân phối cho các process trong hàng đợi khoảng thời gian đó. Trong bài tập lớn lần này, các hàng đợi sẽ được thực thi bằng giải thuật định thời RR với số lượng phân bổ trong 1 lần chạy bằng max_priority trừ đi priority

3.2 Trả lời câu hỏi

Câu hỏi 1: Lợi thế của việc sử dụng Priority Queue so với các giải thuật định thời khác là gì?
Trả lời: Khi so sánh với các giải thuật định thời khác, Priority Queue có một số ưu điểm:

- **Đảm bảo ưu tiên cho các tiến trình quan trọng hơn**: Khi sử dụng Priority Queue, các tiến trình quan trọng hơn sẽ được ưu tiên thực thi trước, đảm bảo rằng các tiến trình này sẽ được xử lý đúng lúc và đảm bảo tính đúng đắn của hệ thống.
- **Tối ưu hóa sử dụng CPU**: Priority Queue giúp tối ưu hóa sử dụng CPU bằng cách ưu tiên thực thi các tiến trình quan trọng hơn. Khi các tiến trình quan trọng được thực thi nhanh hơn, CPU có thể được sử dụng hiệu quả hơn để xử lý các tiến trình khác.
- **Tính linh hoạt và động lực cao**: Hàng đợi ưu tiên có thể được cập nhật động lực và ưu tiên của các tiến trình dựa trên sự thay đổi của hệ thống. Điều này giúp đảm bảo tính linh hoạt của hệ thống và cho phép các tiến trình quan trọng được thực thi đúng lúc trong mọi tình huống.
- **Thích ứng với các tác vụ có độ ưu tiên khác nhau**: Hàng đợi ưu tiên cho phép xử lý các tác vụ có độ ưu tiên khác nhau một cách hiệu quả, giúp hệ thống có thể thích ứng với nhiều loại tác vụ và môi trường khác nhau.

3.3 Hiện thực Scheduler

3.3.1 Hiện thực code trong file queue.c

Hiện thức queue cơ bản theo nguyên tắc FIFO. Chúng ta có **MAX_QUEUE_SIZE = 10** được khai báo trong file queue.h, vậy nên mỗi queue không chứa quá 10 process.

- Hàm enqueue
  Chức năng: Hàm enqueue giúp đưa process mới vào trong ready queue

```
void enqueue(struct queue_t *q, struct pcb_t *proc)
{
    /* TODO: put a new process to queue [q] */
    q->proc[q->size] = proc;
    q->size++;
}

struct pcb_t *dequeue(struct queue_t *q)
{
    /* TODO: return a pcb whose priority is the highest
     * in the queue [q] and remember to remove it from q
     * */
    if (empty(q))
    {
        return NULL;
    }
    struct pcb_t *temp = q->proc[0];
    for (int i = 0; i < q->size - 1; i++)
    {
        q->proc[i] = q->proc[i + 1];
    }
    q->proc[q->size - 1] = NULL;
    q->size--;
    return temp;
}
```

- Hàm dequeue
  Chức năng: Hàm de_queue trả về process có có priority cao nhất trong ready queue và lấy nó ra khỏi ready queue.

```
struct pcb_t *dequeue(struct queue_t *q)
{
    /* TODO: return a pcb whose priority is the highest
     * in the queue [q] and remember to remove it from q
     * */
    if (empty(q))
    {
        return NULL;
    }
    struct pcb_t *temp = q->proc[0];
    for (int i = 0; i < q->size - 1; i++)
    {
        q->proc[i] = q->proc[i + 1];
    }
    q->proc[q->size - 1] = NULL;
    q->size--;
    return temp;
}
```

3.3.2 Hiện thực code trong file sched.c

Scheduler có chức năng quản lý việc cập nhật các process sẽ được thực thi cho CPU, cụ thể là quản lý 2 hàng đợi ready và run như ở trên đã mô tả. Cụ thể:

- Bổ sung hàm init_scheduler()

```
{
#ifdef MLQ_SCHED
  int i;
  for (i = 0; i < MAX_PRIO; i++)
  {
    mlq_ready_queue[i].size = 0;
    // init number of cpu each queue can use maximally
    mlq_ready_queue[i].slot_cpu_can_use = MAX_PRIO - 1;
  }
#endif
  ready_queue.size = 0;
  run_queue.size = 0;
  pthread_mutex_init(&queue_lock, NULL);
}
```

- Hàm struct pcb_t *get_proc()

```
{
  struct pcb_t *proc = NULL;
  /*TODO: get a process from [ready_queue].
   * Remember to use lock to protect the queue.
  * */
  struct pcb_t *proc = NULL;
  /*TODO: get a process from [ready_queue].
  * Remember to use lock to protect the queue.
  * */
  pthread_mutex_lock(&queue_lock);

  if(empty(&ready_queue)){
    /* if ready queue is empty, push all processes 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);
  }
  /*unlock*/
  pthread_mutex_unlock(&queue_lock);
  return proc;
}
```

Giải thích hàm struct pcb_t *get_proc(void) Lấy PCB của một tiến trình đang chờ ở hàng đợi sẵn sàng. Nếu hàng đợi rỗng tại thời điểm hàm được gọi, bạn phải di chuyển tất cả PCB của các quy trình đang chờ ở hàng đợi chạy trở lại vào hàng đợi sẵn sàng trước khi nhận một tiến trình từ hàng đợi sẵn sàng.

- Hàm get_mlq_proc()

```
struct pcb_t *get_mlq_proc(void)
{
  struct pcb_t *proc = NULL;
```

```
  bool flag = false;
  pthread_mutex_lock(&queue_lock);
  for (int i = prio; i < MAX_PRIO; i++)
  {
    if (!empty(&mlq_ready_queue[i]) && queue_slot[i] > 0)
    {
      prio = i;
      flag = true;
      break;
    }
  }
  if (!flag)
  {
    for (int i = 0; i < prio; i++)
    {
      if (!empty(&mlq_ready_queue[i]) && queue_slot[i] > 0)
      {
        prio = i;
        flag = true;
        break;
      }
    }
    if (!flag)
    {
      prio = 0;
      pthread_mutex_unlock(&queue_lock);
      return NULL;
    }
  }
  proc = dequeue(&mlq_ready_queue[prio]);
  queue_slot[prio]--;
  if (queue_slot[prio] == 0)
  {
    queue_slot[prio] = MAX_PRIO - prio;
    prio++;
    if (prio == MAX_PRIO)
    {
      prio = 0;
    }
  }
  pthread_mutex_unlock(&queue_lock);
  return proc;
}
```

Giải thích hàm struct pcb_t *get_mlq_proc(void)

- Ta tiến hành duyệt từng queue trong mlq_ready_queue từ vị trí prio đến cuối. Nếu lần đầu tiên tìm được queue không rỗng và slot > 0 thì dequeue queue đó để thực thi. Cùng với đó sẽ giảm slot đi 1.
- Nếu quá trình duyệt ở trên không lấy ra được process nào (ta nhận biết bằng một biến flag) thì sẽ một lần nữa duyệt từ đầu đến cuối tất cả hàng đợi trong ready queue với mục đích như cũ.
- Nếu lần duyệt thứ hai vẫn không lấy ra được process thì sẽ trả về NULL và gán prio $= 0$.
- Khi giải slot đi 1 sau khi đã lấy tiến trình ra khỏi hàng đợi, ta cần phải lưu s rằng nếu slot đã giảm về 0 thì phải phục hồi slot về giá trị MAX_PRIO - prio như ban đầu và tăng prio lên 1 (queue hiện tại hết lượt truy cập, nhường tài nguyên cho queue tiếp

theo) nếu prio tăng đến MAX_PRIO (đã duyệt xong queue cuối cùng) thì cập nhật prio = 0.

3.3.3 Trường hợp sched

Để chạy các test sched cần phải bỏ comment 3 dòng trong file **os-cfg.h**

4 2 3
0 p1s
1 p2s
2 p3s

Giải thích input

- Dòng đầu input "4 2 3" có nghĩa là
  - 4: Mỗi timeslice sẽ là 4
  - 2: 2 CPU là CPU0 và CPU1
  - 3: tổng cộng 3 process. Các process đều được định nghĩa là p1s nên thời gian thực thi của mỗi process là 10s. Các process có độ ưu tiên bé nhất là 1 nên số queue ở đây là 2.
- Gọi các process từ trên xuống lần lượt là process 1, process 2 và process 3. Do độ ưu tiên cao nhất của các process là 1 (ở process 1) nên số queue cần dùng là 2 queue, queue 0 (độ ưu tiên cao, được lấy 2 lần liên tiếp) và queue 1 (độ ưu tiên thấp hơn, chỉ được lấy 1 lần).

Listing 1: Output of sched

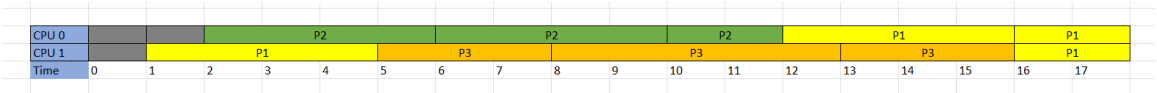
```
vinh@DESKTOP-AUULAN9:/mnt/c/VSCODE/C++/Assigment_L01_Nh m 9$ ./os sched
Time slot   0
ld_routine
        Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
Time slot   1
        CPU 1: Dispatched process  1
        Loaded a process at input/proc/p1s, PID: 2 PRIO: 0
Time slot   2
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
Time slot   3
Time slot   4
Time slot   5
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  3
Time slot   6
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   7
Time slot   8
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot   9
Time slot  10
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  11
Time slot  12
        CPU 0: Processed  2 has finished
```

```
        CPU 0: Dispatched process  1
Time slot  13
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  14
        CPU 1: Processed  3 has finished
Time slot  15
        CPU 1 stopped
Time slot  16
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  17
Time slot  18
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Giải thích output

- Tại timeslot 1, process 1 được load vào queue 1 và được thực thi ngay lập tức trong CPU0. (Hết 1 / 1 lần dispatch của queue 1)
- Tại timeslot 2, process 2 xuất hiện và được đưa vào queue 0, lúc này CPU1 đang trống nên process 2 được load vào CPU1 và thực thi. (1 / 2 lần dispatch của queue 0)
- Tại timeslot 5, process 3 xuất hiện sau đó và được đưa vào queue 0 đang trống. Tuy nhiên lúc này 2 CPU đều đã được sử dụng nên phải chờ.
- Sau khi hết timeslice 4s đến timeslot 4, process 1 được đưa ra khỏi CPU0 và trở về queue 1, lúc này ta thấy chỉ mới lấy từ queue 0 một lần, ta tiếp tục lấy process từ queue 0 một lần nữa nên process 3 được lấy ra và đưa vào CPU 0 thực thi (hết 2 / 2 lần dispatch của queue 0, trở về
- Đến timeslot 6, process 2 thực hiện xong 4s sẽ được đưa trở lại queue 0. Lúc này queue 0 đã được lấy 2 lần, con trỏ queue trở lại lấy các process từ queue 1, process 2 được đưa vào CPU1 đang trống queue 1).
- Tiếp tục tuần tự như vậy đến timeslot 17, khi CPU0 hoàn thành các process thì chương trình kết thúc

Biểu đồ Gantt và kết quả chạy test sched



Hình 2: GANTT

3.3.4 Trường hợp sched_0

```
2 1 2
0 s0
4 s1
```

Giải thích input

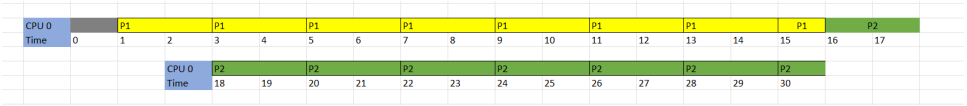
- Dòng đầu input "2 1 2" có nghĩa là
  - 2: Mỗi timeslice sẽ là 4
  - 1: 1 CPU là CPU0
  - 2: tổng cộng 2 process. Các process đều được định nghĩa là s0 nên thời gian thực thi của mỗi process là 15s. Các process có độ ưu tiên bé nhất là 1 nên số queue ở đây là 2.

Listing 2: Output of sched_0

```
vinh@DESKTOP-AUULAN9:/mnt/c/VSCODE/C++/Assigment_L01_Nh m 9$ ./os sched_0
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s0, PID: 2 PRIO: 0
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   6
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   8
Time slot   9
        CPU 0: Put process  1 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  1
Time slot  12
Time slot  13
        CPU 0: Put process  1 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  1
Time slot  16
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
Time slot  17
Time slot  18
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  19
Time slot  20
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  21
Time slot  22
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
```

```
Time slot   23
Time slot   24
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   25
Time slot   26
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   27
Time slot   28
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   29
Time slot   30
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   31
        CPU 0: Processed  2 has finished
        CPU 0 stopped
```

Biểu đồ Gantt và kết quả chạy testcase sched_0



Hình 3: GANTT

3.3.5 Trường hợp sched_1

```
2 1 4
0 s0
4 s1
6 s2
7 s3
```

Giải thích input

- Dòng đầu input "2 1 4" có nghĩa là
  - 2: Mỗi timeslice sẽ là 2
  - 1: 1 CPU là CPU0
  - 4: tổng cộng 4 process. Các process đều được định nghĩa s0 nên thời gian thực thi của mỗi process là 15s. Các process có độ ưu tiên thấp nhất là 4 nên số queue sử dụng là 5.
- Gọi các process từ trên xuống lần lượt là process 1, process 2, process 3 và process 4. Do độ ưu tiên cao nhất của các process là 4 (ở process 1) nên số queue cần dùng là 5 queue, queue 0 (độ ưu tiên cao, được lấy 2 lần liên tiếp), queue 4 (độ ưu tiên thấp hơn, chỉ được lấy 1 lần) các queue 1, 2, 3 được khai báo nhưng không được sử dụng do không có process có priority tương ứng

Listing 3: Output of sched_1

```
Time slot 0
ld_routine
   Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot 1
   CPU 0: Dispatched process 1
   Loaded a process at input/proc/s0, PID: 2 PRIO: 0
Time slot 2
   Loaded a process at input/proc/s0, PID: 3 PRIO: 0
Time slot 3
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 2
    Loaded a process at input/proc/s0, PID: 4 PRIO: 0
Time slot 4
Time slot 5
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 3
Time slot 6
Time slot 7
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 4
Time slot 8
Time slot 9
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 2
Time slot 10
Time slot 11
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 3
Time slot 12
Time slot 13
    CPU 0: Put process 3 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 4
Time slot 16
Time slot 17
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 2
Time slot 18
Time slot 19
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 3
Time slot 20
Time slot 21
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 4
Time slot 22
Time slot 23
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 2
Time slot 24
Time slot 25
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 1
Time slot 26
Time slot 27
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 3
```

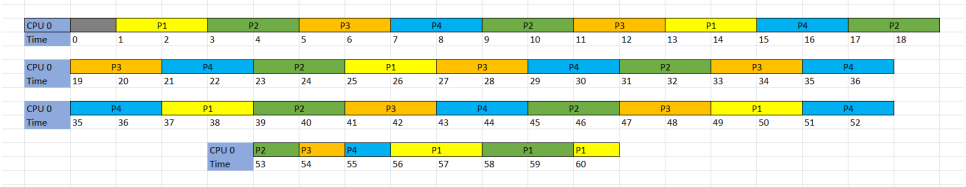
```
Time slot 28
Time slot 29
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 4
Time slot 30
Time slot 31
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 2
Time slot 32
Time slot 33
   CPU 0: Put process 2 to run queue
   CPU 0: Dispatched process 3
Time slot 34
Time slot 35
   CPU 0: Put process 3 to run queue
   CPU 0: Dispatched process 4
Time slot 36
Time slot 37
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 1
Time slot 38
Time slot 39
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 2
Time slot 40
Time slot 41
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 3
Time slot 42
Time slot 43
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 4
Time slot 44
Time slot 45
   ‘CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 2
Time slot 46
Time slot 47
    CPU 0: Put process 2 to run queue
    CPU 0: Dispatched process 3
Time slot 48
Time slot 49
    CPU 0: Put process 3 to run queue
    CPU 0: Dispatched process 1
Time slot 50
Time slot 51
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 4
Time slot 52
Time slot 53
    CPU 0: Put process 4 to run queue
    CPU 0: Dispatched process 2
Time slot 54
    CPU 0: Processed 2 has finished
    CPU 0: Dispatched process 3
Time slot 55
    CPU 0: Processed 3 has finished
    CPU 0: Dispatched process 4
Time slot 56
    CPU 0: Processed 4 has finished
    CPU 0: Dispatched process 1
Time slot 57
```

```
Time slot 58
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 59
Time slot 60
    CPU 0: Put process 1 to run queue
    CPU 0: Dispatched process 1
Time slot 61
    CPU 0: Processed 1 has finished
    CPU 0 stopped
```

Giải thích output

- Tại timeslot 0, process 1 được load vào queue 4 và được thực thi ngay lập tức. Sau đó 1 giây process 2 cũng đã xuất hiện và được đưa vào queue 0.
- Tại timeslot 2, process 1 đã thực hiện xong 2s thì sẽ được lấy ra khỏi CPU và đưa về queue 4 vì queue 4 chỉ có độ ưu tiên thấp, mỗi lần gọi chỉ được 1 lần dispatch. Lúc này, queue 0 đang có process 2, process này được load vào CPU và thực thi, lúc này process 3 được đưa vào queue 0 đang trống. Sau đó 1 giây tại timeslot 3, process 4 được đưa vào queue 0 ngay sau process 3.
- Tại timeslot 4, process 2 ra khỏi CPU, đưa về queue 0 sau process 4 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 2 / 5 lần dispatch), lúc này process 3 được đưa vào CPU và thực thi.
- • Tại timeslot 6, process 3 ra khỏi CPU, đưa về queue 0 sau process 2 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 3 / 5 lần dispatch), lúc này process 4 được đưa vào CPU và thực thi.
- • Tại timeslot 8, process 4 ra khỏi CPU, đưa về queue 0 sau process 3 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 4 / 5 lần dispatch), lúc này process 2 được đưa vào CPU và thực thi.
- • Tại timeslot 10, process 2 ra khỏi CPU, đưa về queue 0 sau process 4 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 5 / 5 lần dispatch), lúc này process 3 được đưa vào CPU và thực thi.
- • Tại timeslot 12, sau khi đủ 5 lần dispatch, hệ điều hành tiếp tục tìm các process ở các queue thấp hơn để thực hiện, nó xuống queue 4 và đưa process 1 vào CPU.
- • Tại timeslot 14, sau khi process 1 thực hiện xong (lần 1 / 1 lần dispatch) queue 4, hệ điều hành sẽ lại quay về dispatch 5 lần trên queue 0 và cứ tiếp tục như thế đến khi các process hoàn thành

Biểu đồ Gantt và kết quả chạy testcase sched_1



Hình 4: GANTT

4 Memory management

4.1 Cơ sở lý thuyết

Quản lý bộ nhớ (tiếng Anh: memory management) là việc điều hành bộ nhớ máy tính ở cấp bậc hệ thống. Mục đích quan trọng của việc quản lý bộ nhớ là cung cấp những cách thức để cấp phát động các ô nhớ cho chương trình khi được yêu cầu và giải phóng các ô nhớ đó khi không cần dùng nữa. Đây là việc rất quan trọng đối với bất kỳ hệ thống máy tính cao cấp nào vì sẽ có nhiều công việc được tiến hành ở mọi thời điểm.

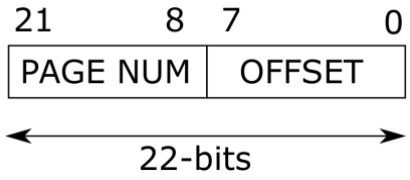
4.1.1 Bộ nhớ ảo (Virtual memory) của mỗi quá trình

Không gian vùng nhớ ảo (virtual memory space) được thiết kế sử dụng phương pháp ánh xạ bộ nhớ (memory mapping) cho mỗi process PCB.
Vùng nhớ (Memory area) là một dải liên tục độc các địa chỉ ô nhớ ảo được ánh xạ trực tiếp tới các địa chỉ vật lý tương ứng. Trong đó, có các vùng nhớ con liên tục (hoặc rời rạc) chứa nội dung của các quá trình (code, data,...) được gọi là **Memory Region**.

Địa chỉ CPU (CPU Address): địa chỉ được tạo ra bởi CPU nhằm truy cập tới một vị trí ô nhớ cụ thể. Trong hệ thống với cấu trúc phân trang, nó được chia thành:

- **Page number (p)**: được dùng như chỉ số của bảng phân trang đang lưu trữ địa chỉ nền của các trang ở bộ nhớ vật lý (**Physical memory**).

- **Page offset (d)**: kết hợp với địa chỉ nền để định nghĩa địa chỉ bộ nhớ vật lý mà sẽ được gửi cho **Memory Management Unit**



Hình 5: CPU Address

4.1.2 Bộ nhớ vật lý (Physical memory)

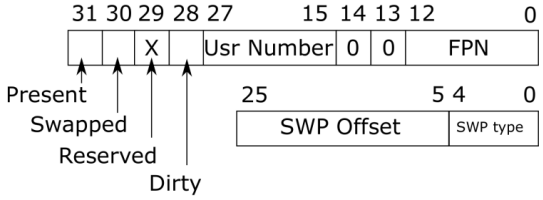
Tương tự như **bộ nhớ ảo** (**Virtual memory**), **bộ nhớ vật lý** (**Physical memory**) được chia thành các khối nhớ nhỏ có kích thước cố định gọi là **frame** hay **page frame**. Mỗi frame trong bài tập lớn này được lưu frame number sẵn có trong mã nguồn định nghĩa về cấu trúc của frame.

Như đã đề cập, tất cả vùng nhớ có vùng nhớ ảo được ánh xạ, và chúng độc lập với nhau. Tuy nhiên, tất cả ánh xạ đó đều có chỉ tới một thiết bị nhớ vật lý đơn lẻ (**Singleton physical device**). Trong bài tập lớn này có hai thiết bị vật lý cần quan tâm:

- **RAM (Random Access Memory)**: có thể được truy cập trực tiếp từ CPU address bus.
- **SWAP**: thiết bị nhớ thứ cấp, không thể truy cập trực tiếp từ CPU. Để thực thi các chương trình và dữ liệu lưu ở thiết bị này, chúng phải được chuyển lên bộ nhớ chính (**main memory**) để thực thi.

4.1.3 Dịch địa chỉ dựa trên phân trang (Paging-based address translation scheme)

Mỗi một **virtual page** của process đều có một bảng phân trang (**page table**). Bảng phân trang giúp **userspace process** tìm ra được **physical frame** của mỗi **virtual page** được ánh xạ tới. Trong mỗi **page table**, có các **Page Table Entries** (PTE). Trong các page table entries bao gồm một giá trị 32-bit cho mỗi virtual page, có định nghĩa về data và cấu trúc một PTE như sau:



Hình 6: Page Table Entry format

```
 * Bits 0-12  page frame number (PFN) if present
 * Bits 13-14 zero if present
 * Bits 15-27 user-defined numbering if present
 * Bits 0-4   swap type if swapped
 * Bits 5-25  swap offset if swapped
 * Bit  28    dirty
 * Bits 29    reserved
 * Bit  30    swapped
 * Bit  31    presented
```

Hình 7: Page table format

Memory Swapping: Khi thực thi một process, các nội dung trong page của process tương ứng sẽ được tải vào bất kỳ frame nào trong bộ nhớ chính, cụ thể ở đây là **MEMRAM**. Khi máy tính vượt quá lượng bộ nhớ chính cho phép, tức là không còn frame trống trong **MEMRAM**, hệ điều hành sẽ di chuyển những nội dung trong page không mong muốn đến bộ nhớ thứ cấp, cụ thể là vào **frame MEMSWAP** (swapping out). Ngược lại, cơ chế swapping in sẽ di chuyển các nội dung của page ta cần dùng để thực thi process tương ứng, nhưng hiên tại đang ở trong **MEMSWAP**, vào các frame **MEMRAM** bộ nhớ chính. Cơ chế thực hiện swapping out trong bài tập lớn này được thực hiện theo quy tắc **First In First Out (FIFO)**, tức là nội dung trong page được tải vào frame **MEMRAM** sớm nhất trong số các page đang ở trong **MEMRAM**, sẽ được sao chép hoặc chuyển vào frame trong **MEMSWAP**, sau đó giải phóng bộ nhớ trong frame **MEMRAM** đó

4.2 Trả lời câu hỏi

Câu hỏi 1: **Điều gì sẽ xảy ra nếu chúng ta chia địa chỉ thành nhiều hơn 2 cấp trong hệ thống quản lý bộ nhớ phân trang?**

Trong hệ thống quản lý bộ nhớ phân trang, địa chỉ bộ nhớ được chia thành hai cấp:

- **Trang (Page)**
- **Độ dời offset (Page offset)**

Tuy nhiên, nếu chúng ta chia địa chỉ thành nhiều hơn 2 cấp, điều này sẽ dẫn đến sự phức tạp hóa trong việc quản lý bộ nhớ và có thể ảnh hưởng đến hiệu suất hệ thống.

Khi chia địa chỉ thành nhiều hơn 2 cấp, ta sẽ cần sử dụng nhiều bảng trang để ánh xạ từng cấp địa chỉ vào vị trí bộ nhớ tương ứng. Việc sử dụng nhiều bảng trang này sẽ làm tăng thời gian truy cập vào bộ nhớ để ánh xạ địa chỉ, dẫn đến giảm hiệu suất của hệ thống.

Ngoài ra, việc chia địa chỉ thành nhiều hơn 2 cấp cũng đòi hỏi bộ nhớ phải lưu trữ nhiều thông tin hơn, dẫn đến sự lãng phí tài nguyên bộ nhớ. Điều này có thể gây ra các vấn đề liên quan đến khả năng lưu trữ và quản lý bộ nhớ, như thiếu bộ nhớ hoặc sự chậm trễ trong việc truy cập bộ nhớ.

Vì vậy, trong hệ thống quản lý bộ nhớ phân trang, chia địa chỉ thành nhiều hơn 2 cấp không được khuyến khích để đảm bảo tính đơn giản và hiệu quả của hệ thống quản lý bộ nhớ.

Câu hỏi 2: Lợi ích và bất cập trong việc phân đoạn bộ nhớ

Thiết kế nhiều phân đoạn bộ nhớ trong hệ điều hành có một số lợi ích quan trọng như sau:

- **Sử dụng hiệu quả bộ nhớ:** Với thiết kế nhiều phân đoạn, bộ nhớ có thể được sử dụng hiệu quả hơn bởi vì không cần phải tải toàn bộ chương trình hoặc quá trình vào bộ nhớ khi chúng được khởi động. Thay vào đó, chỉ có phần của chương trình hoặc quá trình cần được tải vào bộ nhớ, giúp tiết kiệm tài nguyên bộ nhớ.
- **Bảo vệ dữ liệu:** Với nhiều phân đoạn, các khu vực bộ nhớ khác nhau có thể được phân biệt và bảo vệ khỏi sự xâm nhập của các chương trình khác hoặc người dùng, giúp đảm bảo tính toàn vẹn của dữ liệu và cải thiện bảo mật hệ thống.

- **Tăng tốc độ thực thi:** Khi chỉ một phần của chương trình hoặc quá trình được tải vào bộ nhớ, việc truy cập và thực thi các phần của chương trình hoặc quá trình này sẽ nhanh hơn, giúp cải thiện tốc độ thực thi và hiệu suất của hệ thống.
- **Dễ dàng quản lý:** Thiết kế nhiều phân đoạn giúp quản lý bộ nhớ và các tài nguyên của hệ thống dễ dàng hơn, bởi vì các khu vực bộ nhớ khác nhau có thể được quản lý và giám sát độc lập nhau.

Tuy nhiên, thiết kế nhiều phân đoạn cũng có một số nhược điểm như tăng khối lượng công việc của bộ điều khiển bộ nhớ và độ phức tạp của quá trình quản lý bộ nhớ. Do đó, việc thiết kế phân đoạn bộ nhớ cần được cân nhắc kỹ lưỡng để đảm bảo rằng nó phù hợp với các yêu cầu và mục đích của hệ thống.

4.3 Hiện thực

4.3.1 ALLOC

- int **find_victim_page**(struct mm_struct *mm, int *retpgn);

```
int find_victim_page(struct mm_struct *mm, int *retpgn) {
  struct pgn_t *pg = mm->fifo_pgn;

  /* TODO: Implement the theorical mechanism to find the victim page */
  if (pg == NULL) return -1;

  struct pgn_t *prev = NULL;
  while (pg->pg_next) {
    prev = pg;
    pg = pg->pg_next;
  }

  *retpgn = pg->pgn;
  prev->pg_next = NULL;

  free(pg);

  return 0;
}

```

Chức năng: Hàm find_victim_page có chức năng là tìm và trả về trang (page) nạn nhân từ một không gian bộ nhớ ảo (virtual memory space) được đại diện bởi một cấu trúc dữ liệu mm_struct. Trang nạn nhân là trang cần bị loại bỏ để tạo chỗ cho trang mới(Trong hệ điều hành này sử dụng thuật toán FIFO).

- int **alloc_pages_range**(struct pcb_t *caller, int req_pgnum, struct framephy_struct **frm_lst);

```
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct
    framephy_struct **frm_lst) {
  int pgit, fpn;
  struct framephy_struct *newfp_str = NULL;

  for (pgit = 0; pgit < req_pgnum; pgit++) {
    newfp_str = (struct framephy_struct *)malloc(sizeof(struct
    framephy_struct));

    if (MEMPHY_get_freefp(caller->mram, &fpn) == 0) {
      newfp_str->fpn = fpn;
    }
    else { // ERROR CODE of obtaining somes but not enough frames
      int vicpgn, swpfpn;
      if (find_victim_page(caller->mm, &vicpgn) == -1
      || MEMPHY_get_freefp(caller->active_mswp, &swpfpn)==-1)
      {
        if (*frm_lst == NULL) {
          return -1;
        }
        else {
          struct framephy_struct *freefp_str;
          while (*frm_lst != NULL) {
            freefp_str = *frm_lst;
            *frm_lst = (*frm_lst)->fp_next;
            free(freefp_str);
```

```
          }
          return -3000;
        }
      }

      uint32_t vicpte = caller->mm->pgd[vicpgn];
      int vicfpn = PAGING_FPN(vicpte);
      __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
      pte_set_swap(&caller->mm->pgd[vicpgn], 0, swpfpn);
      newfp_str->fpn = vicfpn;
    }

    newfp_str->fp_next = *frm_lst;
    *frm_lst = newfp_str;
  }

  return 0;
}

```

Chức năng: Hàm alloc_pages_range có chức năng cấp phát một dãy các trang vật lý (physical pages) cho một process với cấu trúc pcb_t. Các trang vật lý này được lưu trữ trong danh sách liên kết frm_lst, và số lượng trang cần cấp phát được xác định bởi tham số req_pgnum.

- int **vmap_page_range**(struct pcb_t *caller, int addr, int pgnum, struct framephy_struct *frames, struct vm_rg_struct *ret_rg);

```
/*
 * vmap_page_range - map a range of page at aligned address
*/
int vmap_page_range(
    struct pcb_t *caller,          // process call
    int addr,                      // start address which is aligned to pagesz
    int pgnum,                     // num of mapping page
    struct framephy_struct *frames, // list of the mapped frames
    struct vm_rg_struct *ret_rg)  // return mapped region, the real mapped fp
{                                  // no guarantee all given pages are mapped
  struct framephy_struct *fpit;
  int pgit = 0;
  int pgn = PAGING_PGN(addr);

  ret_rg->rg_start = addr; // at least the very first space is usable
  ret_rg->rg_end = ret_rg->rg_start + pgnum * PAGING_PAGESZ;

  /* TODO map range of frame to address space
   *      [addr to addr + pgnum*PAGING_PAGESZ
   *      in page table caller->mm->pgd[]
   */
  for (; pgit < pgnum; ++pgit)
  {
    fpit = frames;
    pte_set_fpn(&caller->mm->pgd[pgn + pgit], fpit->fpn);
    frames = frames->fp_next;
    free(fpit);

    /* Tracking for later page replacement activities (if needed)
     * Enqueue new usage page */
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
  }

```

```
  return 0;
}
```

Chức năng: Hàm vmap_page_range có chức năng ánh xạ một dãy trang vật lý (physical pages) vào không gian địa chỉ ảo (virtual address space) của một tiến trình, được đại diện bởi cấu trúc pcb_t. Dãy trang vật lý này được đại diện bởi danh sách liên kết frames và được ánh xạ vào không gian địa chỉ ảo từ địa chỉ addr với số lượng trang là pgnum.

- int **__alloc**(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr);

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *
    alloc_addr) {
  /*Allocate at the toproof */
  pthread_mutex_lock(&mmvm_lock);
  struct vm_rg_struct rgnode;

  if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
  {
    caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
    caller->mm->symrgtbl[rgid].rg_end = rgnode.rg_end;

    *alloc_addr = rgnode.rg_start;
    pthread_mutex_unlock(&mmvm_lock);
    return 0;
  }

  /* TODO get_free_vmrg_area FAILED handle the region management (Fig.6)*/

  /*Attempt to increate limit to get space */
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
  int inc_sz = PAGING_PAGE_ALIGNSZ(size);
  // int inc_limit_ret
  int old_sbrk;

  old_sbrk = cur_vma->sbrk;

  /* TODO INCREASE THE LIMIT
   * inc_vma_limit(caller, vmaid, inc_sz)
   */
  inc_vma_limit(caller, vmaid, inc_sz);

  /*Successful increase limit */
  caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
  caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;

  *alloc_addr = old_sbrk;

  struct vm_area_struct *remain_rg = get_vma_by_num(caller->mm, vmaid);
  if (old_sbrk + size < remain_rg->sbrk) {
    struct vm_rg_struct *rg_free = malloc(sizeof(struct vm_rg_struct));
    rg_free->rg_start = old_sbrk + size;
    rg_free->rg_end = remain_rg->sbrk;
    enlist_vm_freerg_list(caller->mm, rg_free);
  }

  pthread_mutex_unlock(&mmvm_lock);
  return 0;
}
```

Chức năng: Hàm __alloc có chức năng cấp phát một vùng nhớ cho một tiến trình thông qua việc quản lý không gian địa chỉ ảo (virtual address space).

4.3.2 FREE

- int **__free**(struct pcb_t *caller, int vmaid, int rgid);

```
int __free(struct pcb_t *caller, int vmaid, int rgid) {
  pthread_mutex_lock(&mmvm_lock);

  if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ) {
    pthread_mutex_unlock(&mmvm_lock);
    return -1;
  }

  /* TODO: Manage the collect freed region to freerg_list */
  struct vm_rg_struct *rgnode = get_symrg_byid(caller->mm, rgid);

  if (rgnode->rg_start == 0 && rgnode->rg_end == 0) {
    pthread_mutex_unlock(&mmvm_lock);
    return -1;
  }

  struct vm_rg_struct *freerg_node = malloc(sizeof(struct vm_rg_struct));
  freerg_node->rg_start = rgnode->rg_start;
  freerg_node->rg_end = rgnode->rg_end;
  freerg_node->rg_next = NULL;

  rgnode->rg_start = rgnode->rg_end = 0;
  rgnode->rg_next = NULL;

  /*enlist the obsoleted memory region */
  enlist_vm_freerg_list(caller->mm, freerg_node);

  pthread_mutex_unlock(&mmvm_lock);
  return 0;
}
```

Chức năng: Hàm __free có chức năng giải phóng một vùng nhớ trong không gian địa chỉ ảo của một tiến trình, thông qua quản lý bảng địa chỉ ảo và danh sách vùng nhớ không sử dụng.

4.3.3 READ

- int **pg_getpage**(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller);

```
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
{
  uint32_t pte = mm->pgd[pgn];

  if (!PAGING_PAGE_PRESENT(pte))
  { /* Page is not online, make it actively living */
    int vicpgn = -1, swpfpn = -1;
    // uint32_t vicpte;

    int tgtfpn = PAGING_SWP(pte); // the target frame storing our variable

    /* TODO: Play with your paging theory here */
    /* Find victim page */
    find_victim_page(caller->mm, &vicpgn);
    int vicfpn = GETVAL(caller->mm->pgd[vicpgn], PAGING_PTE_FPN_MASK,
    PAGING_PTE_FPN_LOBIT);
    /* Get free frame in MEMSWP */
    MEMPHY_get_freefp(caller->active_mswp, &swpfpn);
```

```
    if (swpfpn == -1 || vicpgn == -1)
    {
      return -1;
    }
    /* Do swap frame from MEMRAM to MEMSWP and vice versa*/
    /* Copy victim frame to swap */
    __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
    /* Copy target frame from swap to mem */
    __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);

    /* Update page table */
    pte_set_swap(&mm->pgd[vicpgn], 0, swpfpn); //// swap

    /* Update its online status of the target page */
    // pte_set_fpn() & mm->pgd[pgn];
    pte_set_fpn(&pte, tgtfpn);
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
  }

  *fpn = PAGING_FPN(pte);

  return 0;
}
```

Chức năng: Hàm pg_getpage có chức năng lấy trang từ bộ nhớ và đảm bảo rằng trang đó đã được đưa vào bộ nhớ. Nếu trang không nằm trong bộ nhớ (không được hiện hành), hàm sẽ thực hiện một loạt các bước để đưa trang đó vào bộ nhớ, có thể thông qua quá trình swap.

4.4 Giải thích output

Trước tiên ta sẽ giải thích ý nghĩa của file input cho chương trình:

os_1_mlq_paging.txt

```
2 4 8
1048576 16777216 0 0 0
1 p0s  130
2 s3  39
4 m1s  15
6 s2  120
7 m0s  120
9 p1s  15
11 s0 38
16 s1 0
```

- **Dòng 1:** 2, 4, 8 lần lượt là số lượng CPU, số lượng processes và số lượng time slots.
- **Dòng 2:** thông tin về bộ nhớ ảo:
  - 1048576 là dung lượng của vùng nhớ ảo cho mỗi process (1 MB).
  - 16777216 là tổng dung lượng bộ nhớ ảo (16 MB).
  - 0 0 0 là dung lượng của các khu vực bộ nhớ thực (physical memory regions), ở đây có 3 khu vực.

- **Dòng 3-10:** Mô tả các quy tắc và độ ưu tiên cho việc tải các processes.
  - 1 p0s 130: Load process có PID=1 từ file p0s với độ ưu tiên là 130.
  - 2 s3 39: Schedule process có PID=2 với độ ưu tiên là 39.
  - 4 m1s 15: Load memory image từ file m1s cho process có PID=4 với độ ưu tiên là 15.
  - 6 s2 120: Schedule process có PID=6 với độ ưu tiên là 120.
  - 7 m0s 120: Load memory image từ file m0s cho process có PID=7 với độ ưu tiên là 120.
  - 9 p1s 15: Load process có PID=9 từ file p1s với độ ưu tiên là 15.
  - 11 s0 38: Schedule process có PID=11 với độ ưu tiên là 38.
  - 16 s1 0: Schedule process có PID=16 với độ ưu tiên là 0.

os_1_mlq_paging.output

```
Time slot   0
ld_routine
Time slot   1
  Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
  CPU 0: Dispatched process  1
Time slot   2
  Loaded a process at input/proc/s3, PID: 2 PRIO: 39
Time slot   3
  CPU 2: Dispatched process  2
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   4
  Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
Time slot   5
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
  CPU 3: Dispatched process  3
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  2
Time slot   6
  Loaded a process at input/proc/s2, PID: 4 PRIO: 120
  CPU 1: Dispatched process  4
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
ADDRESS | VALUE
Time slot   7
  CPU 3: Put process  3 to run queue
  CPU 3: Dispatched process  3
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  2
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
```

```
ADDRESS | VALUE
00000098: 100
  Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
Time slot   8
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  5
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
ADDRESS | VALUE
00000098: 100
Time slot   9
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  4
  CPU 3: Put process  3 to run queue
  CPU 3: Dispatched process  1
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  3
  Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
read region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
ADDRESS | VALUE
00000014: 102
00000098: 100
Time slot  10
write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
ADDRESS | VALUE
00000014: 102
00000098: 100
  CPU 1: Put process  5 to run queue
  CPU 1: Dispatched process  6
Time slot  11
  CPU 2: Put process  3 to run queue
  CPU 3: Processed  1 has finished
  Loaded a process at input/proc/s0, PID: 7 PRIO: 38
  CPU 2: Dispatched process  3
  CPU 3: Dispatched process  2
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  5
Time slot  12
  CPU 1: Put process  6 to run queue
  CPU 1: Dispatched process  4
Time slot  13
  CPU 3: Put process  2 to run queue
  CPU 3: Dispatched process  6
  CPU 0: Put process  5 to run queue
  CPU 0: Dispatched process  7
  CPU 2: Processed  3 has finished
  CPU 2: Dispatched process  2
Time slot  14
```

```
  CPU 1: Put process  4 to run queue
  CPU 1: Dispatched process  5
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
ADDRESS | VALUE
00000014: 103
00000098: 100
Time slot  15
  CPU 2: Put process  2 to run queue
  CPU 2: Dispatched process  4
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
ADDRESS | VALUE
00000014: 103
00000098: 100
000000a4: 102
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
  CPU 3: Put process  6 to run queue
  CPU 3: Dispatched process  2
Time slot  16
  CPU 3: Processed  2 has finished
  CPU 3: Dispatched process  6
  Loaded a process at input/proc/s1, PID: 8 PRIO: 0
  CPU 1: Put process  5 to run queue
  CPU 1: Dispatched process  5
write region=0 offset=0 value=0
print_pgtbl: 0 - 512
00000000: c0000000
00000004: 80000006
ADDRESS | VALUE
00000014: 103
00000098: 100
000000a4: 102
000000b0: 1
Time slot  17
  CPU 1: Processed  5 has finished
  CPU 1: Dispatched process  8
  CPU 2: Put process  4 to run queue
  CPU 2: Dispatched process  4
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  18
  CPU 3: Put process  6 to run queue
  CPU 3: Dispatched process  6
Time slot  19
  CPU 2: Put process  4 to run queue
  CPU 2: Dispatched process  4
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
  CPU 1: Put process  8 to run queue
  CPU 1: Dispatched process  8
Time slot  20
  CPU 3: Put process  6 to run queue
  CPU 3: Dispatched process  6
Time slot  21
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
```

```
  CPU 2: Processed  4 has finished
  CPU 2 stopped
  CPU 1: Put process  8 to run queue
  CPU 1: Dispatched process  8
Time slot  22
  CPU 3: Processed  6 has finished
  CPU 3 stopped
Time slot  23
  CPU 1: Put process  8 to run queue
  CPU 1: Dispatched process  8
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  24
  CPU 1: Processed  8 has finished
  CPU 1 stopped
Time slot  25
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  26
Time slot  27
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  28
  CPU 0: Processed  7 has finished
  CPU 0 stopped
```

- **Time slot 0:** Thực hiện ld_routine.
- **Time slot 1:**
  - Quá trình với PID 1 được nạp từ input/proc/p0s.
  - CPU 0 gán nhiệm vụ cho process 1.
- **Time slot 2:**
  - Quá trình với PID 2 được nạp từ input/proc/s3.
  - CPU 2 gán nhiệm vụ cho process 2.
  - CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 3:**
  - Quá trình với PID 3 được nạp từ input/proc/m1s.
  - CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 3 gán nhiệm vụ cho process 3.
  - CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 4:**
  - Quá trình với PID 4 được nạp từ input/proc/s2.
  - CPU 1 gán nhiệm vụ cho process 4.
  - Thực hiện một hoạt động ghi vào region 1, offset 20, với giá trị là 100.
  - In các mục bảng trang.
- **Time slot 5:**

  - CPU 3 đưa process 3 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.

- **Time slot 6:**
  - Quá trình với PID 5 được nạp từ input/proc/m0s.
  - CPU 1 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho process 5.

- **Time slot 7:**
  - CPU 3 đưa process 3 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - Thực hiện một hoạt động đọc từ region 1, offset 20, hiển thị giá trị là 100.
  - In các mục bảng trang.

- **Time slot 8:**
  - CPU 1 đưa process 5 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - Thực hiện một hoạt động ghi vào region 2, offset 20, với giá trị là 102.
  - In các mục bảng trang.

- **Time slot 9:**
  - CPU 0 đưa process 1 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 3 đưa process 3 vào hàng đợi chờ và gán nhiệm vụ cho process 1.
  - CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho process 3.
  - Quá trình với PID 6 được nạp từ input/proc/p1s.
  - Thực hiện một hoạt động đọc từ region 2, offset 20, hiển thị giá trị là 102.
  - In các mục bảng trang.

- **Time slot 10:**
  - Thực hiện một hoạt động ghi vào region 3, offset 20, với giá trị là 103.
  - In các mục bảng trang.

- **Time slot 11:**
  - CPU 1 đưa process 5 vào hàng đợi chờ và gán nhiệm vụ cho process 6.
  - CPU 2 đưa process 3 vào hàng đợi chờ.
  - CPU 3 xử lý và hoàn thành process 1.
  - Quá trình với PID 7 được nạp từ input/proc/s0.
  - CPU 2 gán nhiệm vụ cho process 3.
  - CPU 3 gán nhiệm vụ cho process 2.
  - CPU 0 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho process 5.

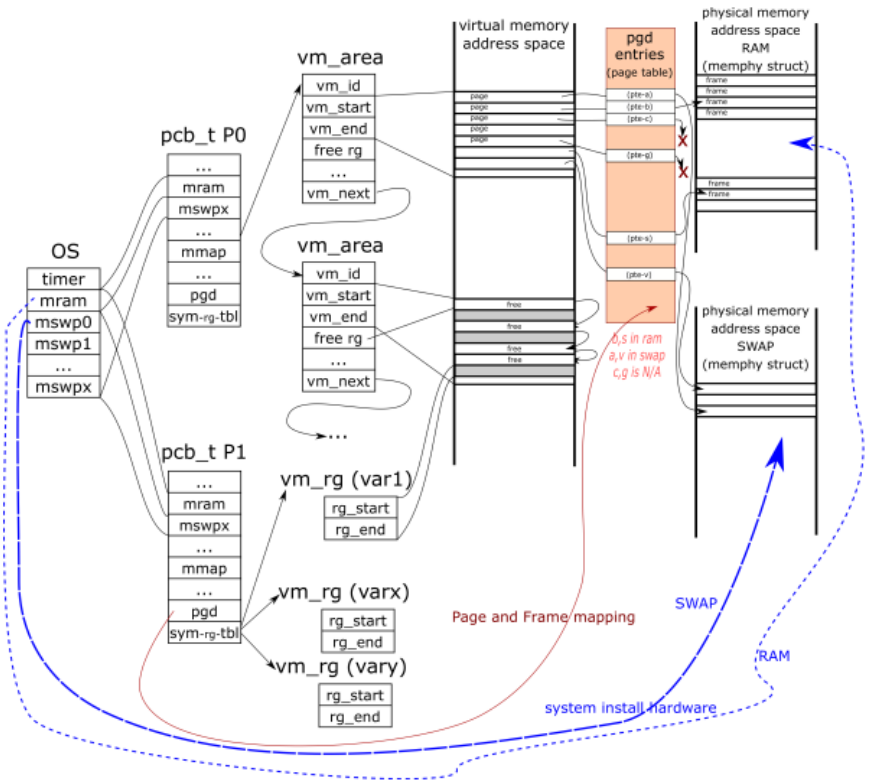
- **Time slot 12:**
  - CPU 1 đưa process 6 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 13:**
  - CPU 3 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho process 6.
  - CPU 0 đưa process 5 vào hàng đợi chờ và gán nhiệm vụ cho process 7.
  - CPU 2 xử lý và hoàn thành process 3.
  - CPU 2 gán nhiệm vụ cho process 2.
- **Time slot 14:**
  - CPU 1 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho process 5.
  - Thực hiện một hoạt động ghi vào region 1, offset 20, với giá trị là 102.
  - In các mục bảng trang.
- **Time slot 15:**
  - CPU 2 đưa process 2 vào hàng đợi chờ và gán nhiệm vụ cho process 4.
  - Thực hiện một hoạt động ghi vào region 2, offset 1000, với giá trị là 1.
  - In các mục bảng trang.
- **Time slot 16:**
  - CPU 3 xử lý và hoàn thành process 2.
  - CPU 3 gán nhiệm vụ cho process 6.
  - Quá trình với PID 8 được nạp từ input/proc/s1.
  - CPU 1 đưa process 5 vào hàng đợi chờ và gán nhiệm vụ cho process 8.
- **Time slot 17:**
  - CPU 1 xử lý và hoàn thành process 5.
  - CPU 1 gán nhiệm vụ cho process 8.
  - CPU 2 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 18:**
  - CPU 3 đưa process 6 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 19:**
  - CPU 2 đưa process 4 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 1 đưa process 8 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 20:**
  - CPU 3 đưa process 6 vào hàng đợi chờ và gán nhiệm vụ cho nó.

- **Time slot 21:**
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 2 xử lý và hoàn thành process 4.
  - CPU 2 dừng lại.
  - CPU 1 đưa process 8 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 22:**
  - CPU 3 xử lý và hoàn thành process 6.
  - CPU 3 dừng lại.
- **Time slot 23:**
  - CPU 1 đưa process 8 vào hàng đợi chờ và gán nhiệm vụ cho nó.
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 24:**
  - CPU 1 xử lý và hoàn thành process 8.
  - CPU 1 dừng lại.
- **Time slot 25:**
  - CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 26:**
  - Không có sự kiện quan trọng.
- **Time slot 27:** CPU 0 đưa process 7 vào hàng đợi chờ và gán nhiệm vụ cho nó.
- **Time slot 28:**
  - CPU 0 xử lý và hoàn thành process 7.
  - CPU 0 dừng lại.

5 Put It All Together

5.1 Cơ sở lý thuyết

Sau khi tổng hợp hai phần trên, ta sẽ có được một hệ điều hành đơn giản như sau:



Hình 8: Thao tác liên quan bộ nhớ ảo

5.2 Trả lời câu hỏi

Câu hỏi : Điều gì sẽ xảy ra nếu việc đồng bộ hóa không được xử lý trong hệ điều hành đơn giản? Minh họa bằng ví dụ vấn đề của hệ điều hành đơn giản này, nếu có?

Trong một hệ điều hành multi-tasking, đồng bộ hóa là rất quan trọng để đảm bảo rằng nhiều process hoặc thread không can thiệp vào việc thực thi lẫn nhau và truy cập vào tài nguyên được chia sẻ. Nếu đồng bộ hóa không được xử lý đúng cách, nó có thể dẫn đến tình trạng race conditions, deadlocks và các sự cố khác có thể khiến hệ thống trở nên không ổn định hoặc không phản hồi.

5.3 Kết quả hiện thực

Sử dụng lệnh **./os os_1_singleCPU_mlq_paging** cho file **os_1_singleCPU_mlq_paging** ta thu được ouput như sau:

```
Time slot   0
ld_routine
Time slot   1
  Loaded a process at input/proc/s4, PID: 1 PRIO: 4
Time slot   2
  CPU 0: Dispatched process  1
  Loaded a process at input/proc/s3, PID: 2 PRIO: 3
Time slot   3
Time slot   4
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
  Loaded a process at input/proc/m1s, PID: 3 PRIO: 2
Time slot   5
Time slot   6
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
  Loaded a process at input/proc/s2, PID: 4 PRIO: 3
Time slot   7
  Loaded a process at input/proc/m0s, PID: 5 PRIO: 3
Time slot   8
  CPU 0: Put process  1 to run queue
  CPU 0: Dispatched process  1
Time slot   9
  CPU 0: Processed  1 has finished
  Loaded a process at input/proc/p1s, PID: 6 PRIO: 2
  CPU 0: Dispatched process  3
Time slot  10
Time slot  11
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  6
  Loaded a process at input/proc/s0, PID: 7 PRIO: 1
Time slot  12
Time slot  13
  CPU 0: Put process  6 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  6
Time slot  16
  Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot  17
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  3
Time slot  18
Time slot  19
  CPU 0: Put process  3 to run queue
  CPU 0: Dispatched process  6
Time slot  20
Time slot  21
  CPU 0: Put process  6 to run queue
  CPU 0: Dispatched process  3
Time slot  22
Time slot  23
  CPU 0: Processed  3 has finished
  CPU 0: Dispatched process  6
Time slot  24
```

```
Time slot   25
  CPU 0: Put process   6 to run queue
  CPU 0: Dispatched process   6
Time slot   26
Time slot   27
  CPU 0: Processed   6 has finished
  CPU 0: Dispatched process   2
Time slot   28
Time slot   29
  CPU 0: Put process   2 to run queue
  CPU 0: Dispatched process   4
Time slot   30
Time slot   31
  CPU 0: Put process   4 to run queue
  CPU 0: Dispatched process   5
Time slot   32
Time slot   33
  CPU 0: Put process   5 to run queue
  CPU 0: Dispatched process   2
Time slot   34
Time slot   35
  CPU 0: Put process   2 to run queue
  CPU 0: Dispatched process   4
Time slot   36
Time slot   37
  CPU 0: Put process   4 to run queue
  CPU 0: Dispatched process   5
Time slot   38
Time slot   39
  CPU 0: Put process   5 to run queue
  CPU 0: Dispatched process   2
Time slot   40
Time slot   41
  CPU 0: Put process   2 to run queue
  CPU 0: Dispatched process   4
Time slot   42
Time slot   43
  CPU 0: Put process   4 to run queue
  CPU 0: Dispatched process   5
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
ADDRESS | VALUE
Time slot   44
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000003
00000004: 80000002
ADDRESS | VALUE
000000a4: 102
Time slot   45
  CPU 0: Put process   5 to run queue
  CPU 0: Dispatched process   2
Time slot   46
Time slot   47
  CPU 0: Put process   2 to run queue
  CPU 0: Dispatched process   4
Time slot   48
Time slot   49
  CPU 0: Put process   4 to run queue
  CPU 0: Dispatched process   5
```

```
write region=0 offset=0 value=0
print_pgtbl: 0 - 512
00000000: c0000000
00000004: 80000002
ADDRESS | VALUE
000000a4: 102
000000b0: 1
Time slot  50
  CPU 0: Processed  5 has finished
  CPU 0: Dispatched process  2
Time slot  51
Time slot  52
  CPU 0: Put process  2 to run queue
  CPU 0: Dispatched process  4
Time slot  53
Time slot  54
  CPU 0: Put process  4 to run queue
  CPU 0: Dispatched process  2
Time slot  55
  CPU 0: Processed  2 has finished
  CPU 0: Dispatched process  4
Time slot  56
Time slot  57
  CPU 0: Processed  4 has finished
  CPU 0: Dispatched process  8
Time slot  58
Time slot  59
  CPU 0: Put process  8 to run queue
  CPU 0: Dispatched process  8
Time slot  60
Time slot  61
  CPU 0: Put process  8 to run queue
  CPU 0: Dispatched process  8
Time slot  62
Time slot  63
  CPU 0: Put process  8 to run queue
  CPU 0: Dispatched process  8
Time slot  64
  CPU 0: Processed  8 has finished
  CPU 0: Dispatched process  7
Time slot  65
Time slot  66
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  67
Time slot  68
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  69
Time slot  70
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  71
Time slot  72
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  73
Time slot  74
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  75
Time slot  76
```

```
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  77
Time slot  78
  CPU 0: Put process  7 to run queue
  CPU 0: Dispatched process  7
Time slot  79
  CPU 0: Processed  7 has finished
  CPU 0 stopped
```

6 Tài liệu tham khảo

1. https://www.geeksforgeeks.org/memory-management-in-operating-system/

2. https://www.geeksforgeeks.org/process-schedulers-in-operating-system/

3. https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm