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

OPERATING SYSTEM (CO2017)

Assignment

"Simple Operating System"

Instructor(s): Hoàng Lê Hải Thanh
Class: L03

Students: Lê Quang Minh - 2212047
Trịnh Tuấn Kiệt - 2211777
Lê Đình Nghĩa - 2212220
Võ Trần Phi Phong - 2420006

HO CHI MINH CITY, November 2024

Contents

Member list & Workload

No.	Fullname	Student ID	Problems	% Work
1	Lê Quang Minh	2212047	- Scheduler - Memmory management - Viết báo cáo	100%
2	Trịnh Tuấn Kiệt	2211777	- Memory management - Viết báo cáo	100%
3	Lê Đình Nghĩa	2212220	- Memory management - Viết báo cáo	40%
4	Võ Trần Phi Phong	2420006	- Memory management - Viết báo cáo	100%

Table 1: Member list & workload

1 Scheduler

1.1 Cơ sở lý thuyết

1.1.1 Scheduler là gì?

Scheduler (Định thời), là một thành phần quan trọng có nhiệm vụ quản lý và phân phối tài nguyên CPU cho các tiến trình (process) đang hoạt động. Mục tiêu chính của scheduler là tối ưu hóa việc sử dụng CPU lựa chọn và sắp xếp các công việc để hệ thống máy tính thực thi và đảm bảo rằng các tiến trình được thực hiện một cách công bằng và hiệu quả.

Trong hệ điều hành đơn giản ta chỉ quan tâm đến việc lựa chọn và sắp xếp các tài nguyên (tiến trình) trong quá trình thực thi để đảm bảo hiệu suất CPU luôn ở mức cao (CPU ultilazation) và tính tương tác cao với người dùng (interactive enhancement). Hai mục tiêu này phục vụ cho nhu cầu dung hòa giữa hai hình thức xử lý của hệ điều hành là multi-programming và multi-tasking. Để thực hiện được Scheduler phải sử dụng các giải thuật để lựa chọn một tiến trình phù hợp từ ready-queue, cấp phát và trao quyền sử dụng CPU cho tiến trình đó. Việc dùng giải thuật nào hoàn toàn dựa vào yêu cầu và mục đích của lập trình viên thông qua các giả thuật đã được tiếp cận.

1.1.2 Các giải thuật định thời

Các giải thuật đã được tiếp cận bao gồm:

1. First come - First serve (FCFS): FCFS thực hiện các tiến trình theo thứ tự mà chúng đến hàng đợi. Tiến trình nào đến trước sẽ được phục vụ trước, thuật toán này rất đơn giản và không cần quá nhiều tài nguyên

để quản lý.

2. Shortest – Job – First (SJF): SJF chọn tiến trình có thời gian thực hiện ngắn nhất để thực hiện trước. Điều này có nghĩa là tiến trình đến trước và có thời gian thực hiện ngắn nhất sẽ được phục vụ trước.
3. Multilevel – Queue (MLQ): Hệ thống sử dụng nhiều hàng đợi khác nhau, mỗi hàng đợi có thể được chỉ định cho một loại tiến trình cụ thể (ví dụ: tiến trình tương tác, tiến trình nền, v.v.). Mỗi hàng đợi có thể sử dụng thuật toán lập lịch riêng.
4. Multilevel – Feedback – Queue (MLFQ): Giống như MLQ, MLFQ sử dụng nhiều hàng đợi, nhưng các hàng đợi này có thể có các mức độ ưu tiên khác nhau và thay đổi linh hoạt.
5. Round – Robin (RR): Các tiến trình được sắp xếp trong một hàng đợi FIFO (First In, First Out). Khi một tiến trình hết thời gian chia hoặc hoàn thành, CPU sẽ chuyển sang tiến trình tiếp theo trong hàng đợi.
6. Priority (Prio): là một thuật toán định thời trong hệ điều hành, trong đó các tiến trình được thực hiện dựa trên mức độ ưu tiên của chúng và có thể bị ngắt hoặc không.

Trong bài tập lớn này chúng ta sẽ phát triển một hệ điều hành sử dụng MLQ kết hợp Round Robin, theo quy ước prio có độ lớn càng thấp thì độ ưu tiên càng cao nhưng trong trường hợp không có độ ưu tiên, thì độ ưu tiên sẽ giảm dần theo tiến trình từ trên xuống.

Ban đầu các hàng đợi có độ ưu tiên khác nhau sẽ có số lần sử dụng CPU khác nhau quản lý bởi time_slot, được cấp phát cố định (time_sot = MAX_PRIO - prio), hàng đợi có độ ưu tiên càng cao thì càng có nhiều lượt

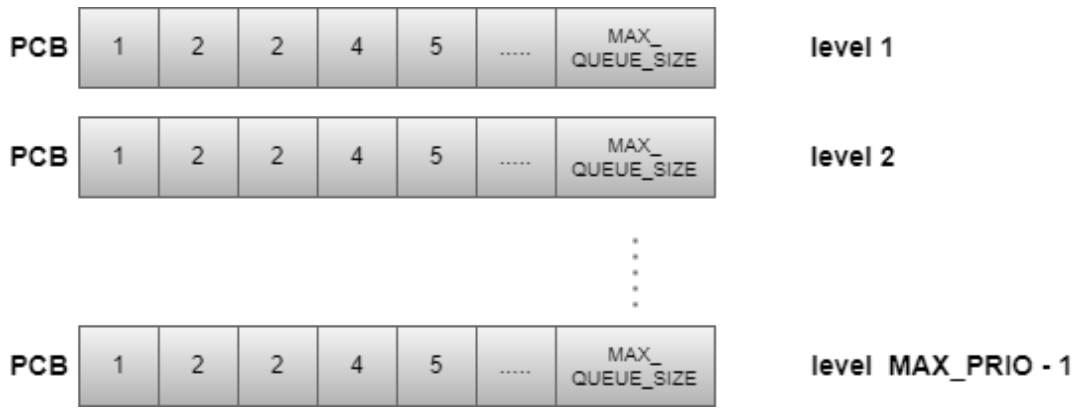

Figure 1: Mô tả hàng đợi trong MLQ

sử dụng CPU. Hàng đợi có độ ưu tiên cao nhất sẽ được chọn để thi thực trước cho đến khi đã sử dụng hết số slot cho phép, và chuyển CPU sang thực thi các process phía dưới. Tất cả hàng đợi sẽ được thay phiên nhau thực thi theo time slice cho trước (Round Robin). Khi tất cả các hàng đợi đều sử dụng hết số time_slot ban đầu, khi đó hệ thống sẽ cấp phát lại và tiếp tục thực hiện định thời.

1.2 Hiện thực

1.2.1 Phần code

a) File queue.c:

Listing 1: enqueue và dequeue

```
void enqueue(struct queue_t * q, struct pcb_t * proc) {
        /* TODO: put a new process to queue [q] */
  if(q->size >= MAX_QUEUE_SIZE || !q || !proc){// neu queue da day
    return;
  }
  q->proc[q->size] = proc;
  q->size++;
}
```

```
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 && !empty(q)){
//gia su pcb dau tien co do uu tien cao nhat
    struct pcb_t* maxproc = q->proc[0];
    if(q->size == 1){//neu chi co mot phan tu
      q->proc[0] = NULL;
      q->size--;
      return maxproc;
    }
    for(int i = 0; i < q->size - 1; i++){
      q->proc[i] = q->proc[i+1];//di chuyen len 1 vi tri
    }
    q->proc[q->size-1] = NULL;
    q->size--;
    return maxproc;
  }
  return NULL;
}
```

Trong hàm enqueue ta sẽ thêm 1 tiến trình vào hàng đợi nếu hàng đợi chưa đầy và tiến trình có tồn tại, sau đó tăng kích thước lên 1. Hàm dequeue sẽ lấy tiến trình có độ ưu tiên cao nhất từ hàng đợi, ở đây ta xử lí nếu chỉ có 1 tiến trình trong hàng đợi ta lấy tiến trình đó ra và xóa đi, nếu có nhiều hơn 1 sau khi lấy ra ta phải dịch tiến trình phía sau lên 1 đơn vị để đảm bảo xóa tiến trình vừa lấy, cuối cùng giảm kích thước hàng đợi.

b) File sched.c:

Listing 2: struct queue_t in queue.h

```
struct queue_t {
  struct pcb_t * proc[MAX_QUEUE_SIZE];
  int size;
  //Begin code
  #ifdef MLQ_SCHED
  int time_slot;
  #endif
  //End code
};
```

Khai báo biến time_slot để cố định thời gian sử dụng CPU của mỗi hàng đợi.

Listing 3: init_scheduler

```
    void init_scheduler(void) {
#ifdef MLQ_SCHED
    int i ;

  for (i = 0; i < MAX_PRIO; i ++){
    mlq_ready_queue[i].size = 0;
    mlq_ready_queue[i].time_slot = MAX_PRIO - i; //modified
  }
#endif
  ready_queue.size = 0;
  run_queue.size = 0;
  pthread_mutex_init(&queue_lock, NULL);
}
```

time_slot mỗi hàng đợi được tính theo công thức $time\_slot = MAX\_PRIO$ - $prio$ giảm dần theo độ ưu tiên.

Listing 4: get_mlq_proc

```
struct pcb_t * get_mlq_proc(void) {
```

```
    struct pcb_t * proc = NULL;
    /*TODO: get a process from PRIORITY [ready_queue].
     * Remember to use lock to protect the queue.
     * */
    pthread_mutex_lock(&queue_lock);//khoa hang doi
    int i = current_queue;
    int empty_queue = 0; // number of queue is empty
    while(empty_queue < MAX_PRIO){
      current_queue = i;
      if(!empty(&mlq_ready_queue[i])){ // queue not empty and size != 0
        if(mlq_ready_queue[i].time_slot > 0){
          proc = dequeue(&mlq_ready_queue[i]);
          mlq_ready_queue[i].time_slot--;
          break;
        }
      }
      else{
        empty_queue++;
      }
      if(i == MAX_PRIO - 1){
        for (int j = 0; j < MAX_PRIO; j++) {
          mlq_ready_queue[j].time_slot = MAX_PRIO - j;
        }
      }

      i = (i + 1) % MAX_PRIO;
    }
    if(empty_queue == MAX_PRIO){
      current_queue = (current_queue + 1) % MAX_PRIO;
    }

    pthread_mutex_unlock(&queue_lock);//mo khoa hang doi
    // printf("i am getting proc here");
    return proc;
}
```

Trong hàm get_mlq_proc ta sử dụng mutex_lock để tránh tình trang race condition giữa các biến và khởi tạo biến curretn_queue để chỉ index hiện tại của queue, ta sẽ duyệt qua các hàng đợi để lấy 1 tiến trình nếu như hàng đợi đó vẫn đang trong thời gian thực thi (còn time_slot) đến khi tất cả hàng đợi trống hoặc đã lấy được thì ta kiểm tra xem số lượng hàng đợi trống đã bằng MAX_PRIO hay chưa, nếu đúng ta cấp phát lại time_slot mở khóa mutex và trả về tiến trình đã lấy được.

1.2.2 Biểu đồ Gantt

Table 2: Bảng tiến trình được sử dụng

Tiến trình	**s0**	**s1**	**s2**	**s3**
Số lệnh	12 15	20 7	20 13	7 17
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc	calc	calc	calc
	calc		calc	calc
	calc		calc	calc
	calc		calc	calc
	calc		calc	calc
	calc		calc	calc
	calc		calc	calc
	calc			calc
	calc			calc
				calc
				calc

Ta thử nghiệm với MAX_RIO = 10.

a) File input sched_1

```
2 1 4
1048576 16777216 0 0 0
0 s0 0
4 s1 0
6 s2 0
7 s3 0
```

Từ input dòng đầu tiên ta thấy time slice = 2, số CPU dùng = 1 và có 4 tiến trình. Dòng thứ 2 là kích thước RAM và kích thước SWP, dòng này không ảnh hưởng đến kết quả nhưng cần để có thể chạy được.

Listing 5: output của sched_1

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 0
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 PRIO: 0
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 PRIO: 0
Time slot   7
```

```
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 4 PRIO: 0
Time slot   8
Time slot   9
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  10
Time slot  11
        CPU 0: Put process  3 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  4
Time slot  14
Time slot  15
        CPU 0: Put process  4 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  3
Time slot  18
Time slot  19
        CPU 0: Put process  3 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  4
Time slot  22
Time slot  23
        CPU 0: Put process  4 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  3
Time slot   26
Time slot   27
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot   28
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot   29
Time slot   30
        CPU 0: Put process  4 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  3
Time slot   33
Time slot   34
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot   35
Time slot   36
        CPU 0: Put process  4 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  3
Time slot   39
Time slot   40
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot   41
Time slot   42
```

```
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
Time slot  43
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  3
Time slot  44
Time slot  45
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  46
Time slot  47
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  48
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  4
Time slot  49
Time slot  50
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  51
Time slot  52
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  53
        CPU 0: Processed  4 has finished
        CPU 0 stopped
```

Giải thích: Đầu tiên ta khởi tạo time_slot cho các hàng đợi từ 0 đến MAX_PRIO - 1 với $time\_slot = MAX\_PRIO$ - $prio$.

mlq_ready_queue[0].time_slot = 10 - 0 = 10.

mlq_ready_queue[1].time_slot = 10 - 1 = 9.

mlq_ready_queue[2].time_slot = 10 - 2 = 8.

mlq_ready_queue[3].time_slot = 10 - 3 = 7.

.

.

.

mlq_ready_queue[9].time_slot = 10 - 9 = 1.

Vì tất cả tiến trình đều có prio = 0 nên time_slot = 10 cho tấc cả process và hàng đợi này trở thành hàng đợi FCFS.

- **Time 0** load process 1 đưa vào mlq_ready_queue[0].
- **Time 1** vì trong hàng đợi bây giờ chỉ có process 1 nên lấy process 1 ra thực thi.
- **Time 3** process thực thi xong đưa vào hàng đợi nhưng chưa có process nào khác nên vẫn thực thi process 1.
- **Time 4** load process 2 đưa vào mlq_ready_queue[0].
- **Time 5** process 1 kết thúc đưa vào hàng đợi, thực hiện process 2.
- **Time 6** load process 3 đưa vào mlq_ready_queue[0].
- **Time 7** process 2 kết thúc đưa vào hàng đợi, trong hàng đợi giờ có process 1 và process 3 nhưng process 1 đứng trước thực hiện trước. Đồng thời load process 4 đưa vào mlq_ready_queue[0].
- **Time 9** process 1 xong đưa vào hàng đợi. Hàng đợi được sắp xếp như sau: process 3 - 2 - 4 - 1, tiếp tục thực thi lần lượt cho đến khi hết số lệnh.
- **Time 53** process 4 đã hết số lệnh và kết thúc chương trình.

Biểu đồ Gantt:

b) File input sched_1 có đặt lại độ ưu tiên

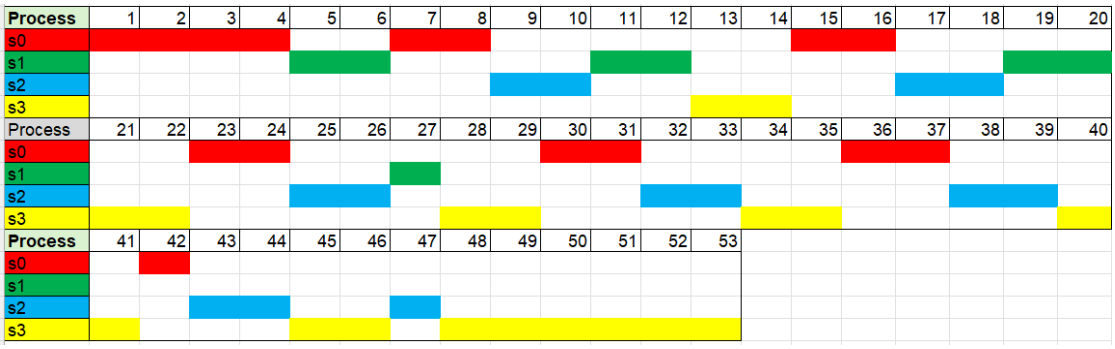

Figure 2: Biểu đồ Gantt cho kết quả sched_1

```
2 1 4
1048576 16777216 0 0 0
0 s0 5
4 s1 8
6 s2 4
7 s3 2
```

Listing 6: output của sched_1 có đặt lại độ ưu tiên

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 5
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 PRIO: 8
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   6
        Loaded a process at input/proc/s2, PID: 3 PRIO: 4
Time slot   7
```

```
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s3, PID: 4 PRIO: 2
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  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  14
Time slot  15
        CPU 0: Put process  2 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  4
Time slot  18
Time slot  19
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  20
Time slot  21
        CPU 0: Put process  4 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  4
Time slot  24
```

```
Time slot  25
        CPU 0: Put process  4 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  4
Time slot  28
Time slot  29
        CPU 0: Put process  4 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  3
Time slot  32
Time slot  33
        CPU 0: Put process  3 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  3
Time slot  36
Time slot  37
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  38
Time slot  39
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  40
Time slot  41
        CPU 0: Put process  3 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  1
Time slot   44
Time slot   45
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   46
Time slot   47
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   48
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
Time slot   49
Time slot   50
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   51
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot   52
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  3
Time slot   53
        CPU 0: Processed  3 has finished
        CPU 0 stopped
```

Giải thích: Đầu tiên ta khởi tạo hàng đợi như trên. mlq_ready_queue[0].time_slot = 10 - 0 = 10.

mlq_ready_queue[1].time_slot = 10 - 1 = 9.

mlq_ready_queue[2].time_slot = 10 - 2 = 8.

mlq_ready_queue[3].time_slot = 10 - 3 = 7.

.

.

.

mlq_ready_queue[9].time_slot = 10 - 9 = 1.

- **Time 0** load process 1 có PRIO = 5 đưa vào mlq_ready_queue[5] có time_slot = 10 - 5 = 5.
- **Time 1 đến 3** thực thi process 1 và tiếp tục đưa vào hàng đợi để thực thi.
- **Time 4** load process 2 có PRIO = 8 đưa vào mlq_ready_queue[8] có time_slot = 10 - 8 = 2.
- **Time 5** process 1 kết thúc được đưa vào hàng đợi, bây giờ hàng đợi có process 1 và 2 nhưng time_slot của process 1 vẫn còn (time_slot = 3) nên vẫn thực thi.
- **Time 6 và 7** load process 3 và 4 đưa vào mlq_ready_queue[4] và mlq_ready_queue[2] có time_slot là 6 và 8.
- **Time 11** process 1 kết thúc được đưa vào hàng đợi, process 1 đã hết time_slot nên nhường cho process 2 thực thi.
- **Time 15** process 2 đã hết time_slot và được đưa vào hàng đợi. Process 2 đang ở mlq_ready_queue[8] và thuật toán sẽ duyệt tiếp tục đến mlq_ready_queue[9] khi đó không có process nào nên khởi tạo lại time_slot cho các hàng đợi. Tại mlq_ready_queue[2] có process 4 nên process 4 sẽ được thực thi với time_slot = 8.
- **Time 31** process 4 đã hết time_slot và được đưa vào hàng đợi, tiếp tục duyệt các hàng đợi sau ta thấy process 3 ở mlq_ready_queue[4] có time_slot = 6 và được thực thi.

– **Time 43** process 1 ở mlq_ready_queue[5] thực thi cho đến khi hết số lệnh và nhường lại cho các process khác.

– **Time 48 đến 53** khi process 1 hoàn thành tiếp đến sẽ là process 2 và khi process 2 hoàn thành giải thuật sẽ khởi tạo lại time_slot cho process 3 và 4 và tiếp tục cho đến khi hết số lệnh.

Biểu đồ Gantt:

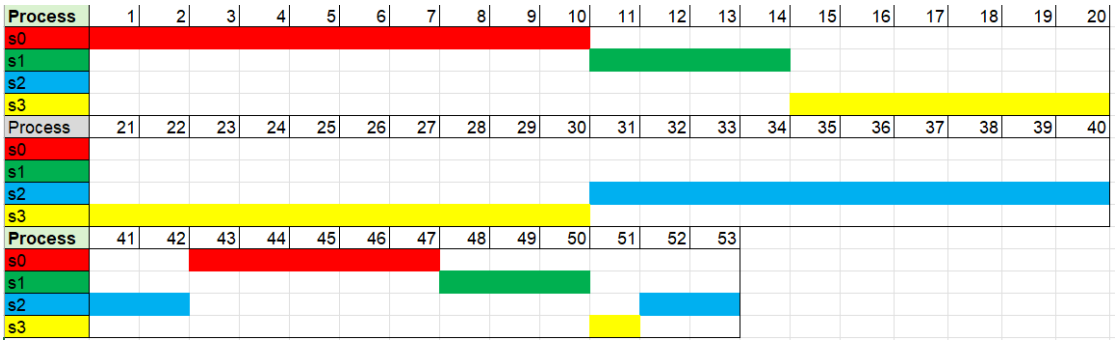

Figure 3: Biểu đồ Gantt kết quả của sched_1 có thêm độ ưu tiên

1.3 Câu hỏi lí thuyết

Question 1: What is the advantage of the scheduling strategy used in this assignment in comparison with other scheduling algorithms you have learned?. (Việc thực hiện chiến lược định thời như trong Bài tập lớn lần này có những ưu điểm gì so với việc thực hiện những chiến lược định thời khác mà bạn đã học?)

Trả lời: Nhằm đánh giá khách quan điểm đặc biệt của giải thuật ta sử dụng (MLQ + RR) nên ta chỉ xét các giải thuật có liên quan như: FCFS, SRTF, RR, MLFQ.

Table 3: So sánh các giải thuật định thời

Chiến lược định thời	**Ưu điểm**	**Nhược điểm**
First Come First Service (FCFS)	– Quá trình thực hiện đơn giản. – Mỗi tiến trình được phục vụ theo thứ tự, không có sự thiên lệch nào.	– Không tối ưu thời gian nếu có tiến trình dài đứng trước. – Thời gian chờ dài nếu có các tiến trình phức tạp.
Shortest Remaining Time First (SRTF)	– Thời gian đáp ứng tốt hơn (optional response time). – Giảm thời gian chờ đợi trung bình do tiến trình có thời gian ngắn nhất được thực hiện trước.	– Dễ bị ảnh hưởng bởi các tiến trình quá lớn, xảy ra stavation. – Phương pháp giải quyết rất phức tạp.
Round Robin(RR)	– Mọi tiến trình đều có lượng thời gian sử dụng CPU như nhau, đơn giản như FCFS. – Thích hợp cho các ứng dụng tương tác, giúp giảm thời gian chờ đợi cho người dùng, dễ quản lý. – Không xuất hiện Starvation: các tiến trình lần lượt sử dụng CPU trong 1 quantum(q) (time slice) cố định hoặc đến khi xong sau đó nhường lại cho tiến trình khác.	– Thời gian chờ có thể tăng nếu số lượng tiến trình càng nhiều, thời gian chờ đợi trong ready-queue tỷ lệ thuận với số tiến trình hiện có. – Tăng nguy cơ overhead: do đặc tính preemptive, cần phải kiểm tra và thực hiện chuyển ngữ cảnh thường xuyên. – Vấn đề cài đặt q: nếu q quá lớn, RR sẽ trở thành FCFS, nếu q quá bé dẫn đến chuyển ngữ cảnh liên tục gây lãng phí tài nghuyên.

Multilevel Feedback Queue (MLFQ)	– MLFQ cho phép các tiến trình nhận được CPU một cách nhanh chóng, đáp ứng như cầu lập tức. – Hạn chế Starvation: các tiến trình có độ ưu tiên thấp sẽ được tăng độ ưu tiên sau mỗi vòng lặp.	– Đây là một giải thuật có độ khó cao, phải thực hiện MLQ và switching giữa các mức. – có thể xảy ra overhead do biến động của quá trình và starvation cho các process có độ ưu tiên thấp. – Việc quản lý nhiều hàng đợi và chuyển đổi giữa chúng có thể phức tạp và tốn thời gian.

Giải thuật MLQ được triển khai:

Trong BTL này nhóm đã sử dụng hàng đợi đa mức để chứa các tiến trình, mỗi tiến trình được thực hiện theo time_slot khác nhau, được phân phát theo độ ưu tiên, độ ưu tiên càng cao thì time_slot càng nhiều *slot = MAX_PRIO - prio.*

So sánh giải thuật MLQ được cải tiến với:

– Round Robin: RR không thể xảy ra starvation nhưng MLQ thì lại có, bởi vì RR chia thời gian đồng đều cho tất cả tiến trình còn MLQ dùng time_slot thì sẽ ưu tiên cho các tiến trình có độ ưu tiên cao hơn, đảm bảo tính thoải mái trong việc sử dụng nhưng tiềm ẩn rủi ro.

– Multilevel Feedback Queue: Là giải thuật cải tiến của MLQ linh hoạt và phản hồi nhanh hơn nhưng không thể chủ động điều chỉnh thời gian thực thi của MLQ dùng time_slot.

Mặc dù MLQ tiềm ẩn starvation nhưng việc sử dụng time_slot có thể giảm thiểu khi tiến trình ở mỗi mức được cấp phát CPU cố định, khi hết thời gian buộc tiến trình đó phải nhường CPU cho tiến trình khác. Tuy nhiên

MLQ time_slot sẽ ít linh hoạt hơn khi không có cơ chế switching level.

2 Memory Management

2.1 Cơ sở lý thuyết

Bộ nhớ vật lý là gì?

Bộ nhớ vật lý (Physical Memory) là phần cứng thực tế của máy tính, được sử dụng để lưu trữ dữ liệu và các chương trình trong khi máy tính đang hoạt động. Bộ nhớ vật lý có thể kể đến như RAM (Random Access Memory), ROM (Read-Only Memory) hay cache. Bộ nhớ vật lý có dung lượng giới hạn, tùy thuộc vào phần cứng được cài đặt trên máy tính nhưng có tốc độ truy cập dữ liệu nhanh hơn so với các thiết bị lưu trữ như ổ cứng HDD hoặc SSD.

Bộ nhớ ảo là gì?

Bộ nhớ ảo (Virtual Memory) là một kỹ thuật quản lý bộ nhớ trong hệ điều hành, cho phép một máy tính chạy các chương trình lớn hơn lượng bộ nhớ vật lý (RAM) thực sự có sẵn. Bộ nhớ ảo sử dụng một phần ổ đĩa cứng làm bộ nhớ bổ sung để mở rộng không gian địa chỉ của bộ nhớ RAM. Bộ nhớ ảo chia không gian địa chỉ thành các khối nhỏ có kích thước cố định, gọi là trang (page). Bộ nhớ vật lý (RAM) cũng được chia thành các khối nhỏ có cùng kích thước, gọi là khung trang (page frame). Bộ nhớ ảo sử dụng bảng trang (Page Table) để ánh xạ địa chỉ logic thành địa chỉ vật lý.

Nếu chương trình cần nhiều bộ nhớ hơn lượng RAM có sẵn, các trang ít được sử dụng sẽ được chuyển từ RAM ra ổ đĩa cứng (khu vực gọi là swap space hoặc paging file). Khi cần lại, chúng sẽ được tải lại vào RAM, thay thế các trang khác nếu cần. Bộ nhớ ảo giúp chạy các chương trình yêu cầu bộ nhớ lớn hơn dung lượng RAM hiện có tránh xung đột lẫn nhau, nhưng

chuyển đổi qua lại nhiều lần giữa 2 bộ nhớ có thể dẫn đến giật lag, treo máy do ổ cứng chậm.

Phân trang là gì?

Paging là một phương pháp cho phép tiến trình truy cập vào bộ nhớ ảo. Tiến trình truy cập vào cái trang mà nó cần sẽ không phải đợi chờ nó được load trên bộ nhớ vật lý. Kỹ thuật phân trang cho phép ta lưu trữ và truy xuất dữ liệu từ bộ nhớ thứ 2 của máy tính (ổ cứng, SSD) hoặc từ bộ nhớ ảo đến bộ nhớ chính (RAM).

Phân trang được sử dụng để chia nhỏ bộ nhớ vật lý thành các khối có kích thước cố định, gọi là khung trang (frame), và chia nhỏ bộ nhớ logic (của các tiến trình) thành các khối tương ứng, gọi là trang (page). Khi CPU muốn gọi 1 tiến trình nào đó trong bộ nhớ thì CPU cần được cung cấp thông tin về số trang (page number) và số offset (offset) của tiến trình đó trong bộ nhớ thứ cấp

Kỹ thuật này giúp ánh xạ bộ nhớ logic của một tiến trình sang bộ nhớ vật lý một cách linh hoạt, giúp quản lý bộ nhớ hiệu quả hơn và tránh phân mảnh bộ nhớ (fragmentation). Hình dưới đây mô tả rõ về cách thức hoạt động của việc chuyển từ bộ nhớ ảo sang bộ nhớ chính. p là page number và d là offset. Khi CPU yêu cầu 1 tiến trình nào đó, CPU sẽ cung cấp địa chỉ trang và offset của tiến trình đó. Lúc này, hệ điều hành sẽ đi vào Page Table (nơi chứa các frame number) và lấy ra số trang của frame đó kết hợp với offset đã có ta sẽ có được địa chỉ vật lý của Process cần tìm.

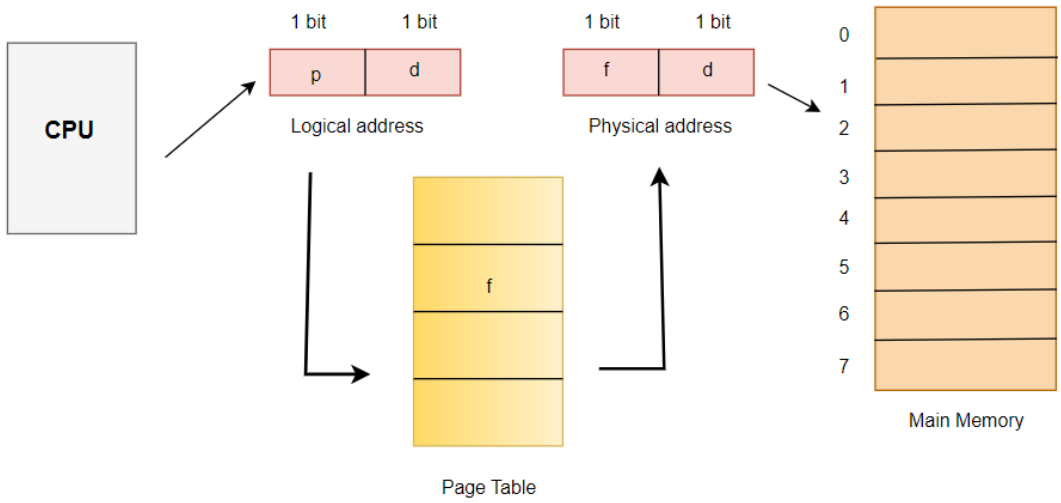

Figure 4: Mô tả phân trang

Phân đoạn là gì?

Phân đoạn (segmentation) là một kỹ thuật quản lý bộ nhớ trong các hệ điều hành, cho phép chia bộ nhớ của một tiến trình thành các khối hoặc phân đoạn (segments) có kích thước và mục đích sử dụng khác nhau, mỗi phân đoạn đại diện cho một phần dữ liệu hoặc mã lệnh cụ thể. Không giống như phân trang (paging), các phân đoạn có kích thước không cố định mà phụ thuộc vào nhu cầu của tiến trình.

Mỗi phân đoạn phản ánh cấu trúc logic của chương trình, giúp dễ dàng quản lý các thành phần của tiến trình như mã lệnh, dữ liệu, hoặc ngăn xếp. Các phân đoạn mã có thể được chia sẻ giữa các tiến trình, giúp tiết kiệm bộ nhớ, các phân đoạn khác nhau có thể được bảo vệ bằng cách gán các quyền truy cập khác nhau (chỉ đọc, đọc/ghi).

Tuy nhiên vì các phân đoạn có kích thước thay đổi, bộ nhớ vật lý có thể bị phân mảnh khi các phân đoạn được cấp phát và thu hồi, với độ phức tạp và kích thước lớn viện quản lý và vấn đề chi phí sẽ có khó khăn.

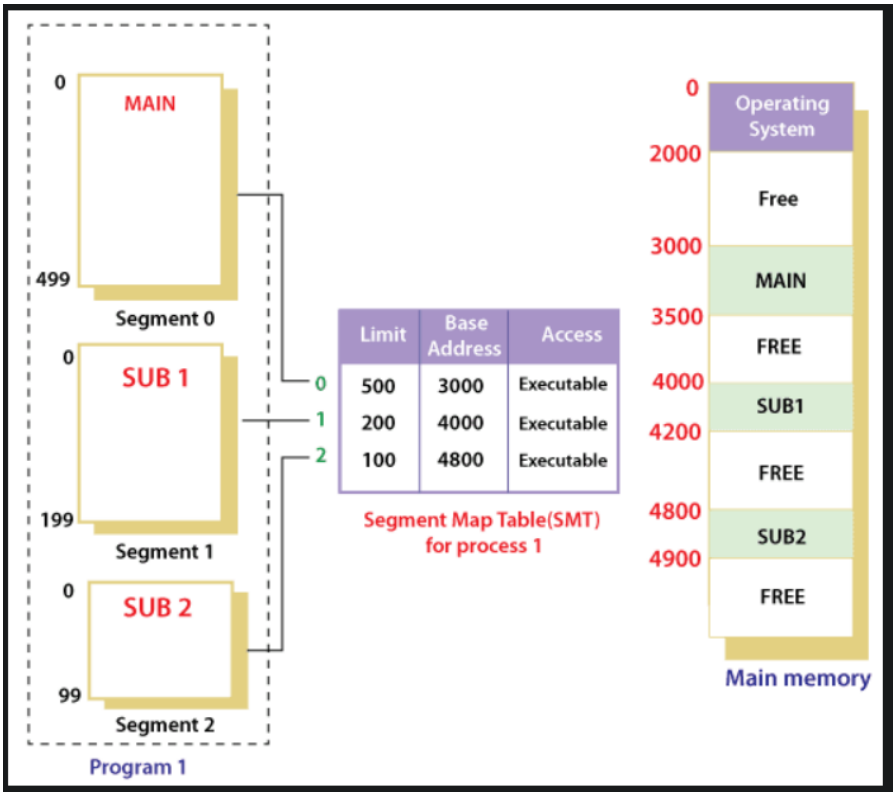

Figure 5: Mô tả phân đoạn

2.2 Hiện thực

2.2.1 The virtual memory mapping in each process

Không gian bộ nhớ ảo được tổ chức như một ánh xạ bộ nhớ cho mỗi quá trình PCB. Địa chỉ ảo bao gồm nhiều vm_areas (liên tiếp). Trong thực tế, mỗi vùng có thể là code, stack hoặc heap. Do đó, quá trình giữ trong pcb của nó một con trỏ của nhiều vùng bộ nhớ liên tiếp.

Khu vực nhớ: mỗi khu vực nhớ có phạm vi liên tục trong [vm_start, vm_end]. Mặc dù trải dài toàn bộ phạm vi, nhưng vùng có thể sử dụng thực tế bị giới hạn bởi sbrk. Trong vùng giữa vm_start và sbrk, có nhiều vùng được struct vm_rg_struct và các slot trống được theo dõi bởi vm_freerg_list.

Vùng nhớ: vùng nhớ được quản lý bằng mảng symrgtbl[PAGING_MAX_SYMTBL_SZ]. Kích thước mảng được cố định bằng một hằng số, biểu thị số lượng biến được cho phép trong mỗi chương trình. struct vm_rg_struct symrgtbl được sử dụng để xác định điểm bắt đầu và điểm kết thúc của vùng nhớ và con trỏ rg_next được dành cho việc theo dõi trong tương lai.

Ánh xạ bộ nhớ: được biểu diễn bằng struct mm_struct, theo dõi tất cả các vùng bộ trong một vùng bộ nhớ liền kề tách biệt. Trong mỗi struct ánh xạ bộ nhớ, nhiều vùng bộ nhớ được xác định bởi struct vm_area_struct *mmap. pgd là thư mục bảng trang và chứa tất cả các mục bảng trang. Mỗi mục ánh xạ số trang thành số khung trong hệ thống quản lý bộ nhớ phân trang. symrgtbl là một triển khai đơn giản của bảng ký hiệu.

Địa chỉ CPU: địa chỉ do CPU tạo ra để truy cập vào một vị trí bộ nhớ cụ thể. Trong hệ thống dựa trên phân trang, nó được chia thành:

- Page number (p): được sử dụng như một index trong bảng trang lưu trữ địa chỉ cơ sở cho mỗi trang trong bộ nhớ vật lý.
- Page offset (d): kết hợp với địa chỉ cơ sở để xác định địa chỉ bộ nhớ vật lý được gửi đến Memory Management Unit (MMU)

Question 2: In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments? (Trong hệ điều hành đơn giản này, chúng ta hiện thực một thiết kế phân đoạn bộ nhớ hoặc vùng nhớ trong khai báo mã nguồn. Ưu điểm của đề xuất thiết kế nhiều phân đoạn là gì?)

Trả lời: việc thiết kế nhiều phân đoạn bộ nhớ trong hệ điều hành có những ưu điểm vượt trội, như giúp sử dụng bộ nhớ hiểu quả, khả năng bảo

vệ, đa nhiệm và tính module. Nó cho phép hệ thống quản lý các ứng dụng phức tạp hiệu quả hơn, cung cấp một framework mạnh mẽ cho môi trường tính toán hiện đại.

- Sử dụng bộ nhớ hiệu quả: các phân đoạn cho phép phân bổ bộ nhớ chỉ khi cần, giảm thiểu lãng phí. Mỗi phân đoạn có thể tăng hoặc giảm kích thước một cách độc lập, đáp ứng tốt các chương trình có nhu cầu bộ nhớ khác nhau.
- Quản lý mã nguồn và dữ liệu đơn giản: mã nguồn, dữ liệu và các phân đoạn stack được quản lý riêng biệt, giúp hệ điều hành và chương trình dễ dàng tổ chức và truy cập các loại bộ nhớ khác nhau. Mỗi phân đoạn có không gian địa chỉ riêng, ngăn ngừa sự chồng chéo và giúp việc truy xuất đơn giản hơn.
- Bảo vệ bộ nhớ: các phân đoạn khác nhau có thể được cấp quyền truy cập riêng biệt (ví dụ: mã nguồn chỉ đọc, dữ liệu đọc-ghi), từ đó giúp tăng cường bảo mật. Lỗi trong một phân đoạn (ví dụ như tràn bộ đệm trong stack) không ảnh hưởng các phân đoạn khác, cải thiện tính ổn định của hệ thống.
- Hỗ trợ cho các hệ thống đa nhiệm (multitasking system) và đa người dùng (multi-user system): hệ điều hành có thể quản lý bộ nhớ cho nhiều quá trình bằng cách chuyển đổi giữa các phân đoạn, cho phép đa nhiệm hiệu quả. Mỗi quá trình có các phân đoạn riêng, đảm bảo sự cô lập bộ nhớ giữa người dùng và quá trình.
- Thiết kế chương trình theo dạng module và linh hoạt: chương trình có thể được chia thành nhiều phân đoạn (ví dụ: các thư viện, module khác nhau), hỗ trợ thiết kế phần mềm theo module.

– Hỗ trợ tốt hơn cho các chương trình lớn và phức tạp: phân đoạn cho phép các chương trình sử dụng nhiều bộ nhớ hơn so với không gian địa chỉ cho phép, đặc biệt hữu ích trên các hệ thống 32-bit. Ngoài ra, các chương trình có thể có nhiều phân đoạn stack và heap cho các luồng hoặc tác vụ khác nhau, cải thiện tính song song và quản lý tài nguyên.

2.2.2 The system physical memory

Question 3: What will happen if we divide the address to more than 2-levels in the paging memory management system? (Điều gì sẽ xảy ra nếu chung ta chia địa chỉ thành hơn 2 cấp trong hệ thống quản lý bộ nhớ phân trang?)

Trả lời: việc sử dụng nhiều hơn 2 cấp phân trang sẽ tăng khả năng quản lý hiệu quả các không gian địa chỉ lớn và phức tạp của hệ thống nhưng cũng làm cho độ trễ truy cập bộ nhớ cao hơn và tăng tính phức tạp.

Ưu điểm:

– Sử dụng bộ nhớ hiệu quả: phân trang đa cấp giúp giảm bộ nhớ cần thiết để lưu trữ các bảng trang bằng cách sử dụng các bảng phân tán nhỏ hơn thay vì một bảng liền kề lớn. Hệ thống chỉ phân bổ các phần cần thiết của hệ thống phân cấp bảng trang, giảm thiểu việc sử dụng bộ nhớ cho các không gian địa chỉ thưa thớt.

– Khả năng mở rộng cho các không gian địa chỉ lớn: cho phép hệ thống hỗ trợ các không gian địa chỉ ảo lớn hơn:

  + Phân trang 2 cấp: phổ biến trong các hệ thống 32-bit.

  + Phân trang 4 cấp: tiêu chuẩn cho các hệ thống 64-bit hiện đại (ví

dụ: x86-64).

+ Phân trang 5 cấp: được sử dụng cho các không gian địa chỉ ảo 64-bit rất lớn (lên đến 128 tebibyte).

Nhược điểm:

– Truy cập chậm hơn: mỗi lần truy cập bộ nhớ yêu cầu nhiều lần tra cứu bảng. Ví dụ, trong phân trang 4 cấp, hệ thống phải truy cập thư mục trang, bảng cấp 2, bảng cấp 3 và cuối cùng là bảng trang. Nhiều cấp hơn có nghĩa là nhiều lần truy cập bộ nhớ hơn cho mỗi lần biên dịch, có khả năng làm tăng thời gian truy cập bộ nhớ vật lý.

– Tăng độ phức tạp: quản lý và duy trì các bảng trang đa cấp làm tăng thêm độ phức tạp cho mã quản lý bộ nhớ của hệ điều hành. Từ đó việc gỡ lỗi và triển khai quản lý bộ nhớ ảo trở nên phức tạp hơn.

2.2.3 Paging-based address translation scheme

Question 4: What is the advantage and disadvantage of segmentation with paging? (Ưu điểm và nhược điểm của phân đoạn với phân trang là gì?)

Trả lời: phân đoạn với phân trang kết hợp hai kỹ thuật quản lý bộ nhớ: phân đoạn và phân trang. Phương pháp kết hợp này tận dụng thế mạnh của cả hai trong khi giảm thiểu một số điểm yếu riêng lẻ của chúng.

Ưu điểm:

– Sử dụng bộ nhớ hiệu quả: phân trang loại bỏ tình trạng phân mảnh ngoại (khoảng trống không sử dụng trong bộ nhớ), đây là vấn đề thường gặp trong phân đoạn. Bộ nhớ được phân bổ thành các phân đoạn có kích thước thay đổi (đơn vị logic) nhưng được quản lý nội bộ dưới dạng

các trang có kích thước cố định, cân bằng giữa tính linh hoạt và hiệu quả.

- Nâng cao khả năng bảo vệ bộ nhớ: các phân đoạn có quyền kiểm soát truy cập (đọc, ghi, thực thi) ở cấp độ phân đoạn, cung cấp khả năng bảo vệ tốt hơn cho các phần quan trọng của bộ nhớ. Mỗi phân đoạn được phân trang riêng biệt nên các tiến trình và luồng được cô lập hiệu quả hơn, giảm nguy cơ hỏng hóc trên các phân đoạn.
- Đa nhiệm hiệu quả: phân trang cho phép tải và dỡ các phân đoạn nhanh chóng, giúp hệ điều hành dễ dàng chuyển đổi giữa các quy trình.

Nhược điểm:

- Tăng độ phức tạp: việc quản lý cả phân đoạn và phân trang làm tăng thêm độ phức tạp cho hệ thống quản lý bộ nhớ của hệ điều hành. Hệ thống phải biên dịch địa chỉ logic bằng cách phân đoạn, sau đó biên dịch tiếp bằng cách phân trang, điều này làm tăng thêm độ phức tạp cho việc phân giải địa chỉ.
- Tăng chi phí bộ nhớ: phân trang yêu cầu nhiều bảng trang cho mỗi phân đoạn, dẫn đến việc sử dụng thêm bộ nhớ để lưu trữ các bảng này. Việc duy trì một bảng phân đoạn cùng với các bảng trang sẽ tiêu tốn thêm bộ nhớ, đặc biệt là khi có nhiều phân đoạn.

2.2.4 Multiple memory segments

Question 5: What is the reason for data being separated into two different segments: data and heap? (Điều gì sẽ xảy ra nếu hệ thống đa lõi có mỗi lõi CPU có thể chạy trong một bối cảnh khác nhau và mỗi lõi có MMU riêng và một phần lõi (TLB) của nó ? Trong CPU hiện đại, TLB 2 cấp hiện nay

rất phổ biến. Tác động của các cấu hình phần cứng bộ nhớ mới này đối với sơ đồ dịch thuật của chúng ta là gì?)

Trả lời: việc phân tách dữ liệu thành hai phân đoạn riêng biệt - data và heap - trong bộ nhớ của chương trình phục vụ các mục đích khác nhau liên quan đến quản lý bộ nhớ, tổ chức và hiệu suất.

Phân đoạn data: được sử dụng để phân bổ bộ nhớ tĩnh, nơi các biến có kích thước cố định, xác định trước được lưu trữ. Đây thường là các biến toàn cục hoặc tĩnh được khởi tạo với giá trị cố định. Kích thước của phân đoạn dữ liệu thường được xác định tại thời điểm biên dịch và không thay đổi trong suốt thời gian chạy. Ví dụ: biến toàn cục, biến tĩnh và hằng số được biết trước khi chương trình chạy.

Phân đoạn heap: được sử dụng để phân bổ bộ nhớ động, trong đó bộ nhớ được phân bổ tại thời điểm chạy cho các biến có kích thước không xác định cho đến khi chương trình được thực thi. Bộ nhớ này được quản lý thủ công (thông qua malloc, free). Kích thước của heap có thể tăng hoặc giảm một cách linh hoạt trong quá trình thực thi chương trình. Ví dụ: bộ nhớ được phân bổ cho các đối tượng, cấu trúc dữ liệu, mảng hoặc nội dung được tạo động khác có thể thay đổi kích thước trong thời gian chạy.

Việc tách data và heap thành hai phân đoạn riêng biệt cho phép quản lý bộ nhớ hiệu quả hơn, ngăn ngừa các vấn đề về phân mảnh và giúp cho hiệu suất tốt hơn. Nó cũng giúp tổ chức bộ nhớ dựa trên cách thức và thời điểm sử dụng, đảm bảo dữ liệu tĩnh và bộ nhớ động được xử lý khác nhau, cho phép hệ thống phân bổ, truy cập và bảo vệ chúng hiệu quả hơn.

2.2.5 Put it all together

Chạy thử test case

Input: os_1_mlq_paging_small_1K với các thông số đầu vào:

– Time slice = 2, Number of CPUs = 4, Number of process = 8.

– RAM_SZ = 2048, SWP_SZ_0 = 16777216.

– SWP_SZ_1 = 0, SWP_SZ_2 = 0, SWP_SZ_3 = 0.

Các quá trình:

p0s	s3	m1s	s2	m0s	p1s	s0	s1
1 14	7 17	1 6	20 13	1 6	1 11	12 15	20 7
calc	calc	alloc 300 0	calc	alloc 300 0	calc	calc	calc
alloc 300 0	calc	alloc 100 1	calc	alloc 100 1	calc	calc	calc
alloc 300 4	calc	free 0	calc	free 0	calc	calc	calc
free 0	calc	alloc 100 2	calc	alloc 100 2	calc	calc	calc
alloc 100 1	calc	free 2	calc	write 102 1 20	calc	calc	calc
write 100 1 20	calc	free 1	calc	write 1 2 1000	calc	calc	calc
read 1 20 20	calc		calc		calc	calc	calc
write 102 2 20	calc		calc		calc	calc	
read 2 20 20	calc		calc		calc	calc	
write 103 3 20	calc		calc		calc	calc	
read 3 20 20	calc		calc		calc	calc	
calc	calc		calc			calc	
free 4	calc		calc			calc	
calc	calc					calc	
	calc					calc	
	calc						
	calc						

Table 4: Bảng các quá trình cho os_1_mlq_paging_small_1K

Tiến hành chạy testcase với input os_1_mlq_paging_small_1K:

```
$ ./os os_1_mlq_paging_small_1K
```

```
Time slot   0
ld_routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
Time slot   1
        CPU 3: Dispatched process  1
[Operation]     PID #1: CALC
Time slot   2
        Loaded a process at input/proc/s3, PID: 2 PRIO: 39
[Operation]     PID #1: ALLOC
        CPU 0: Dispatched process  2
[Operation]     PID #2: CALC
Time slot   3
[Mapping for ALLOC]     PID #1 with frame mapped to 1
[Mapping for ALLOC]     PID #1 with frame mapped to 0
 RAM mapping
 Number of mapped frames:   2
 Number of remaining frames:    6
--------------------
        Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  1
[Operation]     PID #1: ALLOC
[Mapping for ALLOC]     PID #1 with frame mapped to 3
[Mapping for ALLOC]     PID #1 with frame mapped to 2
 RAM mapping
 Number of mapped frames:   4
 Number of remaining frames:    4
--------------------
Time slot   4
        CPU 1: Dispatched process  3
[Operation]     PID #3: ALLOC
[Operation]     PID #2: CALC
[Mapping for ALLOC]     PID #3 with frame mapped to 5
[Mapping for ALLOC]     PID #3 with frame mapped to 4
 RAM mapping
 Number of mapped frames:   6
 Number of remaining frames:    2
--------------------
[Operation]     PID #1: FREE
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
[Operation]     PID #2: CALC
[Operation]     PID #3: ALLOC
[Mapping for ALLOC]     PID #3 with frame mapped to 6
 RAM mapping
 Number of mapped frames:   7
 Number of remaining frames:    1
--------------------
Time slot   5
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
[Operation]     PID #3: FREE
[Operation]     PID #2: CALC
Time slot   6
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  4
[Operation]     PID #4: CALC
[Operation]     PID #4: CALC
Time slot   7
        CPU 2: Dispatched process  1
[Operation]     PID #1: ALLOC
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
get free vm success.
 RAM mapping
 Number of mapped frames:   7
 Number of remaining frames:    1
--------------------
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
[Operation]     PID #2: CALC
[Operation]     PID #3: ALLOC
get free vm success.
 RAM mapping
 Number of mapped frames:   7
 Number of remaining frames:    1
--------------------
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  5
Time slot   8
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
[Operation]     PID #3: FREE
[Operation]     PID #1: WRITE
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
[Operation]     PID #2: CALC
[Operation]     PID #5: ALLOC
[Page Replacement]      PID #5 with Victim frame number: 0      PTE:80000001
[Mapping for ALLOC]     PID #5 with frame mapped to 0
[Mapping for ALLOC]     PID #5 with frame mapped to 7
 RAM mapping
 Number of mapped frames:   8
 Number of remaining frames:    0
--------------------
MEMPHY_dump:
|Location -- Value
----------------------
        Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
[Operation]     PID #5: ALLOC
[Page Replacement]      PID #5 with Victim frame number: 0      PTE:80000000
```

Figure 6: Kết quả chạy testcase với input os_1_mlq_paging_small_1K

```
Time slot   9
        CPU 2: Put process  1 to run queue
[Mapping for ALLOC]     PID #5 with frame mapped to 0
        CPU 2: Dispatched process  6
[Operation]     PID #3: FREE
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
[Operation]     PID #2: CALC
 RAM mapping
 Number of mapped frames:   8
 Number of remaining frames:    0
--------------------
[Operation]     PID #6: CALC
        CPU 3: Put process  5 to run queue
        CPU 3: Dispatched process  4
        CPU 1: Processed  3 has finished
        CPU 1: Dispatched process  5
[Operation]     PID #5: FREE
Time slot  10
[Operation]     PID #6: CALC
[Operation]     PID #4: CALC
[Operation]     PID #2: CALC
        Loaded a process at input/proc/s0, PID: 7 PRIO: 38
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  6
[Operation]     PID #6: CALC
[Operation]     PID #4: CALC
[Operation]     PID #5: ALLOC
get free vm success.
 RAM mapping
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  7
 Number of mapped frames:   8
 Number of remaining frames:    0
--------------------
[Operation]     PID #7: CALC
        CPU 3: Put process  4 to run queue
        CPU 3: Dispatched process  2
[Operation]     PID #2: CALC
        CPU 1: Put process  5 to run queue
Time slot  12
[Operation]     PID #6: CALC
[Operation]     PID #7: CALC
        CPU 1: Dispatched process  4
[Operation]     PID #4: CALC
[Operation]     PID #2: CALC
[Operation]     PID #4: CALC
        CPU 2: Put process  6 to run queue
Time slot  13
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
[Operation]     PID #7: CALC
        CPU 2: Dispatched process  6
[Operation]     PID #6: CALC
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  2
Time slot  14
[Operation]     PID #7: CALC
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  5
[Operation]     PID #5: WRITE
write region=1 offset=20 value=102
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000007
00000008: 80000000
MEMPHY_dump:
|Location -- Value
|220 -- 100
------------------------
[Operation]     PID #6: CALC
[Operation]     PID #2: CALC
[Operation]     PID #2: CALC
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  6
[Operation]     PID #6: CALC
[Operation]     PID #5: WRITE
write region=2 offset=1000 value=1
print_pgtbl: 0 - 768
00000000: 80000000
00000004: 80000007
00000008: 80000000
MEMPHY_dump:
|Location -- Value
|20 -- 102
|220 -- 100
------------------------
Invalid address: out of bound. READ operation failed.
```

Figure 7: Kết quả chạy testcase với input os_1_mlq_paging_small_1K

```
Time slot  15
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
[Operation]     PID #7: CALC
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  8
[Operation]     PID #8: CALC
[Operation]     PID #6: CALC
Time slot  16
[Operation]     PID #7: CALC
        CPU 1: Processed  5 has finished
        CPU 1: Dispatched process  2
[Operation]     PID #2: CALC
Time slot  17
[Operation]     PID #2: CALC
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
[Operation]     PID #7: CALC
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  6
[Operation]     PID #6: CALC
[Operation]     PID #8: CALC
        CPU 3: Put process  8 to run queue
[Operation]     PID #7: CALC
        CPU 3: Dispatched process  8
[Operation]     PID #8: CALC
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
[Operation]     PID #2: CALC
[Operation]     PID #6: CALC
Time slot  18
[Operation]     PID #8: CALC
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  6
[Operation]     PID #6: CALC
Time slot  19
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
[Operation]     PID #2: CALC
[Operation]     PID #7: CALC
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  8
[Operation]     PID #8: CALC
[Operation]     PID #7: CALC
        CPU 2: Processed  6 has finished
Time slot  20
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
[Operation]     PID #2: CALC
        CPU 2: Dispatched process  4
[Operation]     PID #4: CALC
[Operation]     PID #4: CALC
        CPU 1: Processed  2 has finished
        CPU 1: Dispatched process  1
Time slot  21
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
[Operation]     PID #7: CALC
[Operation]     PID #1: READ
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: c0000000
00000004: c0000020
00000008: 80000003
00000012: 80000002
MEMPHY_dump:
|Location -- Value
|20 -- 102
|220 -- 100
-------------------------
[Operation]     PID #8: CALC
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
[Operation]     PID #4: CALC
Time slot  22
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  8
[Operation]     PID #8: CALC
[Operation]     PID #1: WRITE
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: c0000000
00000004: c0000020
00000008: 80000003
00000012: 80000002
MEMPHY_dump:
|Location -- Value
|20 -- 102
|220 -- 100
[Operation]     PID #7: CALC
-------------------------
Invalid address: region not found. READ operation failed.
        CPU 3: Processed  8 has finished
        CPU 3 stopped
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
[Operation]     PID #7: CALC
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot  23
Invalid address: region not found. READ operation failed.
[Operation]     PID #1: READ
read region=2 offset=20 value=-126
print_pgtbl: 0 - 1024
00000000: c0000000
00000004: c0000020
00000008: 80000003
[Operation]     PID #4: CALC
00000012: 80000002
MEMPHY_dump:
|Location -- Value
|20 -- 102
|220 -- 100
-----------------------
        CPU 2: Put process  4 to run queue
        CPU 2: Dispatched process  4
[Operation]     PID #4: CALC
```

Figure 8: Kết quả chạy testcase với input os_1_mlq_paging_small_1K

```
Time slot  24
[Operation]     PID #7: CALC
[Operation]     PID #1: WRITE
write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: c0000000
00000004: c0000020
00000008: 80000003
00000012: 80000002
MEMPHY_dump:
|Location -- Value
|20 -- 102
|220 -- 100
-----------------------
Invalid address: region not found. READ operation failed.
[Operation]     PID #4: CALC
Time slot  25
        CPU 0: Put process  7 to run queue
        CPU 0: Dispatched process  7
[Operation]     PID #7: CALC
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Invalid address: region not found. READ operation failed.
[Operation]     PID #1: READ
read region=3 offset=20 value=-126
print_pgtbl: 0 - 1024
00000000: c0000000
00000004: c0000020
00000008: 80000003
00000012: 80000002
MEMPHY_dump:
|Location -- Value
|20 -- 102
|220 -- 100
-----------------------
        CPU 2: Put process  4 to run queue
Time slot  26
        CPU 0: Processed  7 has finished
[Operation]     PID #1: CALC
        CPU 2: Dispatched process  4
[Operation]     PID #4: CALC
        CPU 0 stopped
        CPU 2: Processed  4 has finished
        CPU 1: Put process  1 to run queue
Time slot  27
        CPU 1: Dispatched process  1
[Operation]     PID #1: FREE
        CPU 2 stopped
Time slot  28
[Operation]     PID #1: CALC
Time slot  29
        CPU 1: Processed  1 has finished
        CPU 1 stopped
```

Figure 9: Kết quả chạy testcase với input os_1_mlq_paging_small_1K

Question 6: What will happen if the synchronization is not handled in your simple OS? Illustrate the problem of your simple OS (assignment outputs) by example if you have any. (Điều gì sẽ xảy ra nếu việc đồng bộ hóa không được xử lý trong hệ điều hành đơn giản của bạn? Hãy minh họa vấn đề của hệ điều hành đơn giản của bạn bằng ví dụ nếu có. Lưu ý: Bạn cần chạy hai phiên bản HĐH đơn giản của mình: chương trình có/không có đồng bộ hóa, sau đó quan sát hiệu suất của chúng dựa trên kết quả demo và giải thích sự khác biệt của chúng.)

Trả lời: Nếu việc đồng bộ hóa không được xử lý đúng cách trong hệ điều hành, một số vấn đề quan trọng có thể phát sinh, đặc biệt là trong môi trường đa luồng hoặc đa tiến trình. Những vấn đề này chủ yếu ảnh hưởng đến tính nhất quán của dữ liệu, tính ổn định và tính chính xác của hệ thống.

Sự không nhất quán của dữ liệu (race condition): khi nhiều luồng hoặc quá trình truy cập dữ liệu chia sẻ cùng lúc và sửa đổi dữ liệu, kết quả cuối cùng có thể phụ thuộc vào thứ tự thực hiện. Nếu không đồng bộ hóa, sẽ khó dự đoán được quá trình/luồng nào truy cập dữ liệu trước, dẫn đến dữ liệu không nhất quán hoặc bị hỏng. Ví dụ: hai luồng cập nhật cùng một biến mà không đồng bộ hóa có thể đọc các giá trị cũ hoặc không chính xác, gây ra lỗi không mong muốn (ví dụ: cập nhật số dư tài khoản ngân hàng không chính xác).

Deadlock: xảy ra khi hai hoặc nhiều quá trình hoặc luồng đang chờ sử dụng tài nguyên, nhưng không tiến trình nào có thể tiếp tục. Điều này thường xảy ra khi các tiến trình có nhiều khóa và mỗi tiến trình giữ một khóa mà tiến trình kia đang chờ. Ví dụ: luồng A giữ khóa 1 và chờ khóa 2, trong khi luồng B giữ khóa 2 và chờ khóa 1. Cả hai đều bị kẹt trong trạng thái chờ, không thể tiếp tục.

Starvation: xảy ra khi một luồng hoặc tiến trình liên tục bị từ chối quyền truy cập vào tài nguyên vì các luồng hoặc tiến trình khác liên tục được ưu tiên hơn luồng hoặc tiến trình đó. Điều này có thể xảy ra nếu thuật toán lập lịch không được thiết kế đúng cách hoặc nếu quyền truy cập tài nguyên không được cân bằng tốt. Ví dụ: một luồng có mức độ ưu tiên thấp hơn có thể không bao giờ nhận được CPU hoặc quyền truy cập vào các tài nguyên được chia sẻ vì các luồng có mức độ ưu tiên cao hơn luôn được cấp quyền truy cập trước.

Tăng độ phức tạp trong gỡ lỗi: khi đồng bộ hóa không được xử lý đúng cách, hệ thống có thể hoạt động đúng trong một số trường hợp nhưng lại không hoạt động trong những trường hợp khác, đặc biệt là khi tải cao hoặc hoạt động đồng thời. Điều này tạo ra các lỗi không xác định khó tái tạo và

gỡ lỗi. Ví dụ: một chương trình có thể hoạt động tốt trong môi trường đơn luồng nhưng lại tạo ra kết quả không thể đoán trước trong môi trường đa luồng do tình trạng race condition.

Cạn kiệt tài nguyên: đồng bộ hóa kém có thể dẫn đến việc sử dụng quá nhiều tài nguyên (như thời gian CPU hoặc bộ nhớ). Ví dụ: khi không có cơ chế khóa thích hợp, các luồng có thể liên tục sử dụng CPU trong khi chờ các tài nguyên không khả dụng.

3 Source code

Source code: Assignment_OS_241

4 Tài liệu tham khảo

[1] *Slide môn học CO2017 - Operating Systems*. 2024.

[2] Abraham Silberschatz, Greg Gagne, Peter B. Galvin. *Operating System Concepts, 10th Edition*. Wiley, 2018.

[3] GeeksforGeeks. *Multilevel Queue (MLQ) CPU Scheduling*. 2024. From: Multilevel Queue (MLQ) CPU Scheduling.