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

OPERATING SYSTEMS (CO201D)

Honored Program Exercise

"Completely Fair Scheduler"

Instructor(s): Lê Thanh Vân
Nguyễn Phương Duy

Students: Nguyễn Thiện Minh - 2312097
Huỳnh Đức Nhân - 2312420

HO CHI MINH CITY, MAY 2025

Contents

List of Figures

List of Tables

1 Cây Đỏ Đen (Red-Black Tree)

1.1 Mục đích và vai trò

Trong quá trình hiện thực bộ định thời CFS, chúng tôi sử dụng cây đỏ đen (Red-Black Tree) làm cấu trúc dữ liệu chính để lưu trữ và quản lý các tiến trình đang chờ thực thi. Cây đỏ đen là một loại cây nhị phân tìm kiếm tự cân bằng, giúp duy trì thời gian truy xuất, thêm và xóa phần tử hiệu quả với độ phức tạp trung bình là O(logn) cho mỗi thao tác.

Việc sử dụng cây đỏ đen phù hợp với triết lý của thuật toán CFS là luôn ưu tiên các tiến trình có vruntime nhỏ nhất, do đó thao tác lấy phần tử nhỏ nhất (`removeminRBTree`) là vô cùng quan trọng và được tối ưu nhờ cấu trúc cây cân bằng.

1.2 Cấu trúc dữ liệu

Định nghĩa của cây đỏ đen được xây dựng bao gồm hai kiểu dữ liệu chính:

- `RBNode`: Đại diện cho một nút trong cây, bao gồm dữ liệu là con trỏ đến một `pcb_t` (tiến trình), màu sắc (RED hoặc BLACK), và ba con trỏ đến con trái, con phải, và cha.
- `RBTree`: Là cấu trúc cây bao gồm con trỏ đến nút gốc (`root`), một nút đặc biệt `TNULL` đại diện cho lá NULL trong cây đỏ đen, và một hàm so sánh `cmp` dùng để xác định thứ tự các tiến trình dựa trên `vruntime`.

```
typedef enum { RED, BLACK } Color;

struct RBNode {
    struct pcb_t *data;
    Color color;
    struct RBNode *left, *right, *parent;
};

struct RBTree {
    int (*cmp)(struct pcb_t *, struct pcb_t *);
    struct RBNode *root;
    struct RBNode *TNULL;
};

int initializeRBTree(struct RBTree **tree, int (*compare)(struct pcb_t *, struct
    pcb_t *));
int insertRBTree(struct RBTree *tree, struct pcb_t *data);
int removeminRBTree(struct RBTree *tree, struct pcb_t **data);
```

1.3 Các hàm chính

Các thao tác chính được định nghĩa và sử dụng bao gồm:

- **initializeRBTree**: Cấp phát bộ nhớ và khởi tạo một cây đỏ đen rỗng với nút `TNULL`, đồng thời gán hàm so sánh để xác định độ ưu tiên.
- **insertRBTree**: Thêm một tiến trình vào cây theo thứ tự được xác định bởi hàm `cmp`. Sau khi thêm, hàm sẽ tự động cân bằng cây theo nguyên lý của cây đỏ đen (sửa màu và thực hiện xoay cây nếu cần).

- **removeminRBTree**: Lấy ra và xóa tiến trình có giá trị vruntime nhỏ nhất trong cây. Vì cây được sắp xếp theo thứ tự tăng dần của `vruntime`, thao tác này tương ứng với việc lấy nút trái nhất của cây.

1.4 Ưu điểm khi dùng cây đỏ đen

Việc sử dụng cây đỏ đen trong bộ định thời CFS mang lại nhiều lợi ích:

- Đảm bảo sự cân bằng động: Không giống như danh sách liên kết hay hàng đợi, cây đỏ đen giữ trạng thái cân bằng, giúp tránh trường hợp thao tác có độ phức tạp O(n).
- Hỗ trợ tìm tiến trình có độ ưu tiên cao nhất nhanh chóng.
- Dễ mở rộng và tái sử dụng trong các bộ định thời khác.

1.5 Kết luận

Tóm lại, cây đỏ đen đóng vai trò trung tâm trong việc lưu trữ và truy xuất tiến trình của thuật toán CFS. Việc hiện thực và sử dụng đúng cách các thao tác như chèn, xóa và tìm phần tử nhỏ nhất là yếu tố then chốt để đảm bảo hiệu năng và tính đúng đắn của bộ định thời này.

2 Completely Fair Scheduler (CFS)

2.1 Hàm thêm, lấy, trả tiến trình

Tương tự với phần hiện thực giải thuật định thời MLQ, có 4 hàm chính ta cần thực hiện: `init_scheduler`, `add_proc`, `put_proc` và `get_proc`. Công dụng và sử dụng của 4 hàm như sau:

- **init_scheduler**: Khởi tạo bộ định thời: Được gọi bởi hàm `main` từ `os.c`, với mục đích khởi tạo các biến cần thiết cho định thời. Vì đang hiện thực CFS, nên ta chỉ đơn giản là tạo một cây đỏ đen mới để lưu các tiến trình (process).
- **add_proc**: Thêm process mới vào cây đợi: Được gọi bởi hàm `ld_routine` từ `os.c`, với mục đích thêm process mới được load vào cây.
- **get_proc**: Lấy process để thực thi: Được gọi bởi `cpu_routine` từ `os.c` với mục đích lấy process được ưu tiên nhất ra khỏi hàng đợi để thực thi. Vì đang thực hiện CFS nên độ ưu tiên sẽ được so sánh bằng `vruntime` và thời gian chạy của process đó cũng sẽ được tính ở hàm này.
- **put_proc**: Trả process về cây đợi: Được gọi bởi `cpu_routine` từ `os.c` với mục đích trả lại process vào cây đợi sau khi đã chạy xong thời gian của nó. Vì một process trong quá trình chờ sẽ không thay đổi `vruntime` nên `vruntime` của mỗi process sẽ được cập nhật ở hàm này trước khi thêm vào cây.

```
1 void add_proc(struct pcb_t * proc) {
2
3     pthread_mutex_lock(&queue_lock);
4
5     proc->pcb_tree = pcb_tree;
6     total_weight += proc->weight;
7     insertRBTree(pcb_tree, proc);
```

```
8
9       pthread_mutex_unlock(&queue_lock);
10  }
```

Như đã trình bày ở trên, hàm `add_proc` chỉ tiến hành gán cây đợi cho process và thêm process đó vào cây. Ngoài ra, trong tất cả mỗi khi thêm hoặc xóa process ra khỏi cây ta đều cần cập nhật tương ứng với biến toàn cục `total_weight`.

```
1   struct pcb_t * get_cfs_proc(void) {
2       struct pcb_t * proc = NULL;
3
4       pthread_mutex_lock(&queue_lock);
5       if (removeminRBTree(pcb_tree, &proc) == 0) {
6           proc->time_slice = target_latency;
7           proc->running_list = &running_list;
8           enqueue(&running_list, proc);
9
10          proc->time_slice = round(proc->weight / total_weight * target_latency);
11          if (proc->time_slice < 1) proc->time_slice = 1;
12
13          total_weight -= proc->weight;
14      }
15      pthread_mutex_unlock(&queue_lock);
16      return proc;
17  }
```

Vì hàm `get_proc` lấy process ra để thực thi nó nên trước khi trả về, ta sẽ gán nó vào `running_list` với ý nghĩa rằng process này đang chạy. Đồng thời, thời gian chạy `time_slice` của process này cũng được tính theo công thức: $time\_slice = \frac{weight\_of\_task}{total\_weight} \times target\_latency$ và luôn có giá trị tối thiểu bằng 1 để đảm bảo không lặp vô hạn.

```
1   void put_proc(struct pcb_t * proc) {
2
3       pthread_mutex_lock(&queue_lock);
4
5       total_weight += proc->weight;
6       proc->vruntime += 1.0 * proc->time_slice / proc->weight;
7       insertRBTree(pcb_tree, proc);
8       remove_proc(&running_list, proc);
9       proc->running_list = NULL;
10
11      pthread_mutex_unlock(&queue_lock);
12  }
```

Vì hàm `put_proc` trả process đang chạy về lại hàng đợi nên ta cần gỡ process đó ra khỏi `running_list`. Mặc khác, trước khi thêm process lại vào cây, ta cần cập nhật giá trị `vruntime` để đảm bảo process đó giữ đúng độ ưu tiên của mình.

2.2 Một số thay đổi khác

Vì cần tính toán `time_slice` cho process được lấy ra bởi hàm `get_proc`, biến `target_latency` sẽ là một biến toàn cục static, được truyền vào từ `os.c` thông qua `init_scheduler`. Đồng thời, quá trình thực thi process trong `cpu_routine` sẽ lấy số `time_slot` bằng với `proc->time_slice` đã được tính toán ở trên.

Mặt khác, vì nhu cầu tính waiting time nên nhóm cũng đã điều chỉnh loader và timer cho quá trình chạy là như nhau trên một máy. Waiting time của một process sẽ được cập nhật khi process đó thực thi xong về kết quả sẽ được in ra vào cuối chương trình.

3 Giải Thích Testcase

3.1 Input Format

System Input Format

```
[target_latency] [N = Number of CPUs] [M = Number of Processes]
[RAM_SIZE] [SWAP_SIZE_0] [SWAP_SIZE_1] [SWAP_SIZE_2] [SWAP_SIZE_3]
[time_0] [path_0] [niceness_0]
[time_1] [path_1] [niceness_1]
...
[time_{M-1}] [path_{M-1}] [niceness_{M-1}]
```

- `target_latency`: Mục tiêu độ trễ của hệ thống.
- `N`: Số lượng CPU trong hệ thống.
- `M`: Số lượng tiến trình cần chạy.
- `RAM_SIZE`: Kích thước bộ nhớ RAM.
- `SWAP_SIZE_i`: Kích thước của vùng nhớ SWAP thứ i, với $i = 0..3$.
- `time_i`: Thời điểm tạo tiến trình thứ i.
- `path_i`: Đường dẫn đến file mô tả tiến trình thứ i.
- `niceness_i`: Mức độ ưu tiên (niceness) của tiến trình thứ i.

Process File Format

```
[priority] [N = Number of Instructions]
instruction_0
instruction_1
...
instruction_{N-1}
```

- `priority`: Độ ưu tiên của tiến trình (Không sử dụng).
- `N`: Số lượng lệnh cần thực hiện.
- `instruction_i`: Lệnh thứ i của tiến trình.

3.2 Testcase 1: 1 CPU test

Input File

`os_cfs_1`

```
10 1 3
0 0 0 0 0
0 proc_1 10
1 proc_1 0
2 proc_1 -10
```

proc_1

```
0 10
calc
...
calc
```

Output Content

```
Time slot   0
ld_routine
        Loaded a process at input/proc/proc_1, PID: 1, NICENESS: 10
        CPU 0: Dispatched process  1
Time slot   1
        Loaded a process at input/proc/proc_1, PID: 2, NICENESS: 0
Time slot   2
        Loaded a process at input/proc/proc_1, PID: 3, NICENESS: -10
Time slot   3
Time slot   4
Time slot   5
Time slot   6
Time slot   7
Time slot   8
Time slot   9
Time slot  10
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
Time slot  11
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  14
Time slot  15
Time slot  16
Time slot  17
Time slot  18
Time slot  19
Time slot  20
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  21
Time slot  22
Time slot  23
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  24
Time slot  25
Time slot  26
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  2
Time slot  27
Time slot  28
```

```
Time slot   29
Time slot   30
        CPU 0: Processed  2 has finished
        CPU 0 stopped
============================================
Total waiting time: 30
Process 1: 0
Process 2: 19
Process 3: 14
Average waiting time: 11.000000
```

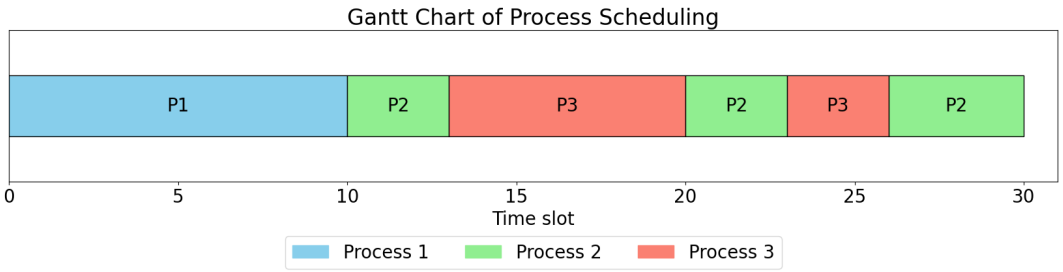

Figure 1: Biểu đồ Gantt mô phỏng 1 CPU cho các tiến trình

Thông tin tiến trình

PID	Niceness	Weight
1	10	512
2	0	1024
3	-10	2048

Table 1: Thông tin trọng số theo niceness

Mô phỏng theo Time Slot

Time	Sự kiện	PID	RBT	Nội dung
0	Load	1	[1]	vruntime(P1) = 0
0	Dispatch	1	[]	Time slice $= \frac{512}{512} \times 10 = 10$
1	Load	2	[2]	vruntime(P2) = 0

2	Load	3	[2,3]	vruntime(P3) = 0
10	Finish	1	[2,3]	–
10	Dispatch	2	[3]	Time slice = $\frac{1024}{3072} \times 10 \approx 3$
13	Preempt	2	[3,2]	vruntime(P2) += $\frac{3}{1024} \approx 0.0029$
13	Dispatch	3	[2]	Time slice = $\frac{2048}{3072} \times 10 \approx 7$
20	Preempt	3	[2,3]	vruntime(P3) += $\frac{7}{2048} = 0.0034$
20	Dispatch	2	[3]	Time slice = $\frac{1024}{3072} \times 10 \approx 3$
23	Preempt	2	[3,2]	vruntime(P2) += $\frac{3}{1024} \approx 0.0059$
23	Dispatch	3	[2]	Time slice = $\frac{2048}{3072} \times 10 \approx 7$
26	Finish	3	[2]	–
26	Dispatch	2	[]	Time slice = $\frac{1024}{1024} \times 10 = 10$
30	Finish	2	[]	–

3.3 Testcase 2: 2 CPU test

Input Files

os_cfs_2

```
10 2 4
0 0 0 0 0
0 proc_2 -15
0 proc_2 -5
0 proc_2 5
0 proc_2 15
```

proc_2

```
0 20
calc
...
calc
```

Output Content

```
Time slot   0
ld_routine
        Loaded a process at input/proc/proc_2, PID: 1, NICENESS: -15
        Loaded a process at input/proc/proc_2, PID: 2, NICENESS: -5
        Loaded a process at input/proc/proc_2, PID: 3, NICENESS: 5
        Loaded a process at input/proc/proc_2, PID: 4, NICENESS: 15
        CPU 1: Dispatched process  1
        CPU 0: Dispatched process  2
Time slot   1
Time slot   2
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  4
Time slot   7
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  1
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot   8
Time slot   9
Time slot  10
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  11
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot  14
Time slot  15
Time slot  16
Time slot  17
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  18
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  19
Time slot  20
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
Time slot  21
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  22
        CPU 1: Processed  1 has finished
        CPU 1: Dispatched process  4
Time slot  23
Time slot  24
Time slot  25
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  3
Time slot  26
Time slot  27
```

```
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  28
Time slot  29
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  30
Time slot  31
Time slot  32
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  33
Time slot  34
Time slot  35
Time slot  36
Time slot  37
        CPU 1: Processed  3 has finished
        CPU 1 stopped
Time slot  38
Time slot  39
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  40
Time slot  41
Time slot  42
Time slot  43
        CPU 0: Processed  4 has finished
        CPU 0 stopped
===========================================
Total waiting time: 43
Process 1: 2
Process 2: 9
Process 3: 17
Process 4: 23
Average waiting time: 12.750000
```

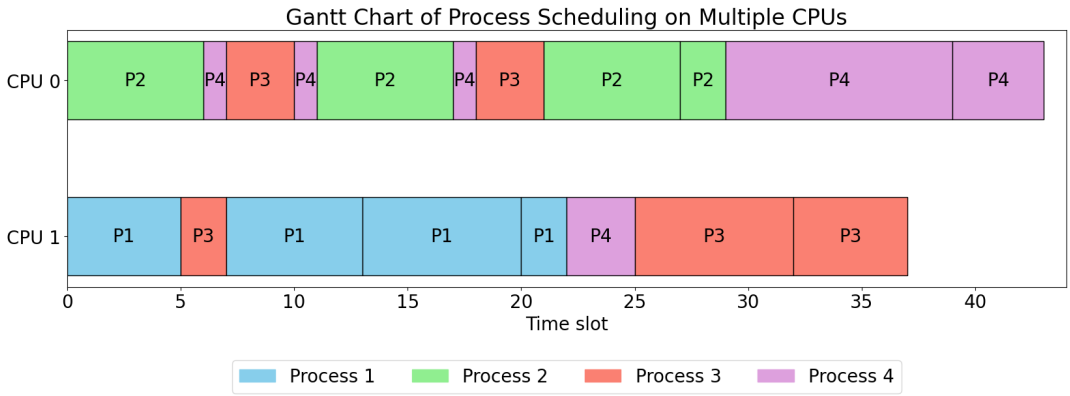

Figure 2: Biểu đồ Gantt mô phỏng 2 CPU cho các tiến trình

Explanation:

Biểu đồ Gantt mô tả quá trình lập lịch cho 4 tiến trình (P1–P4) trên 2 CPU song song, với mỗi tiến trình có chỉ số `niceness` khác nhau, từ đó ảnh hưởng đến mức độ ưu tiên trong lập lịch. Tất cả các tiến trình cùng sử dụng một file thực thi `proc_2`, nhưng được cấp phát tài nguyên và

thời gian thực thi khác nhau dựa theo độ ưu tiên.

Cụ thể, tiến trình P1 với `niceness = -15` (ưu tiên cao nhất) thường xuyên được cấp CPU sớm và chạy liên tục trong các slot đầu trên CPU 1. Tiến trình này kết thúc sớm nhất với thời gian chờ (waiting time) là 2. Tiến trình P2 (`niceness = -5`) cũng được ưu tiên tương đối, phân bố rải rác trên CPU 0 và hoàn tất ở slot 29 với thời gian chờ 9. Trong khi đó, P3 (`niceness = 5`) và đặc biệt là P4 (`niceness = 15` — thấp nhất) bị ngắt quãng thường xuyên, phải chờ lâu và hoàn thành muộn hơn, lần lượt với thời gian chờ là 17 và 23.

Việc phân phối tiến trình giữa hai CPU nhìn chung là hợp lý: CPU 0 chủ yếu xử lý P2 và P4, còn CPU 1 dành phần lớn thời gian cho P1 và P3. Điều này giúp khai thác tốt khả năng song song nhưng cũng tạo ra khá nhiều chuyển ngữ cảnh, dẫn đến tổng thời gian chờ là 131, tương ứng với thời gian chờ trung bình là 12.75.

Tóm lại, thuật toán lập lịch đã thể hiện rõ sự công bằng tương đối giữa các tiến trình với mức độ ưu tiên khác nhau, đồng thời tận dụng tối đa hai CPU. Tuy nhiên, vẫn còn có thể tối ưu tổng thời gian chờ bằng cách giảm tần suất chuyển ngữ cảnh và cân bằng mức ưu tiên theo thời gian đã chờ.

4 Trả lời câu hỏi

Để so sánh ưu điểm và nhược điểm giữa CFS và MLQ, trước hết ta sẽ phân tích về Turn Around Time và Waiting Time của 2 kiểu định thời trong Testcase sau. Vì MLQ sử dụng `time_slot` còn CFS sử dụng `target_latency` nên chỉ số đầu tiên trong testcase sẽ được luân phiên thay đổi: 8 nếu là CFS và 4 nếu là MLQ (chỉ số được chọn để cả hai mô hình có cùng TAT).

```
8 2 3
2048 16777216 0 0 0
0 p1s 1
1 p2s 0
2 p3s 0
```

Trong các process p1s, p2s, p3s lần lượt là 10, 12 và 11 lệnh calc. Vì testcase này khá đơn giản nên giá trị prio (hoặc niceness) đều để ở mức là 1, 0, 0. Mô hình định thời MLQ thu được các khoảng thời gian như sau:

```
// CPU running...
==========================================
Turnaround time: 19
Process 1: 9
Process 2: 0
Process 3: 2
Average waiting time: 3.666667
```

Mặt khác, mô hình định thời CFS thu được TAT và waiting time như sau:

```
// CPU running...
==========================================
Turnaround time: 19
Process 1: 5
Process 2: 0
Process 3: 6
Average waiting time: 3.666667
```

Nhìn vào cách phân bố waiting time vào các process ở hai mô hình, nhóm có một vài nhận xét như sau:

- Ở cả MLQ và CFS, process 2 đều có `waiting time = 0`. Với mô hình MLQ, điều này là hợp lý vì process 2 có độ ưu tiên cao nhất (cùng với process 3) và vì thời gian bắt đầu chạy của 2 CPU là lệch nhau nên process 2 không phải tranh chấp với process 3 vốn đang chạy vào thời điểm đó. Ngược lại, với mô hình CFS thì process 2 lại mang vẻ "may mắn" hơn vì chỉ phải tranh chấp CPU 1 lần vào lúc process 3 đang chạy và process 1 thì đã chạy trước đó (ta có thể thấy rõ hơn ở biểu đồ Gantt). ở cả hai mô hình, process 2 đều được chạy riêng CPU với process 1 và 3 nên không phải tranh chấp.

- Ở mô hình MLQ, vì process 3 có độ ưu tiên cao hơn nên chỉ phải chờ process 1 chạy hết quatumn và sau đó liên tục giành quyền CPU. Do đó, waiting time của process 3 là 2 trong khi của process 1 là 9, điều này thể hiện rằng mô hình MLQ ưu tiên các tiến trình có prio thấp và có khả năng gây starvation cho các tiến trình có prio cao hơn.

- Ở mô hình CFS, waiting time lại được phân bố đều cho process 1 và 3 dù process 3 có chỉ số niceness cao hơn. Vì thời gian được cấp cho process 1 vào ban đầu là khá lớn nên thời gian phải chờ của nó thấp đi, ngược lại process 3 phải chờ process 1 nên thời gian chờ tăng lên. Điều này thể hiện tính công bằng (Fair) của mô hình, nhưng đồng thời độ ưu tiên cho các tiến trình chưa được thể hiện rõ.

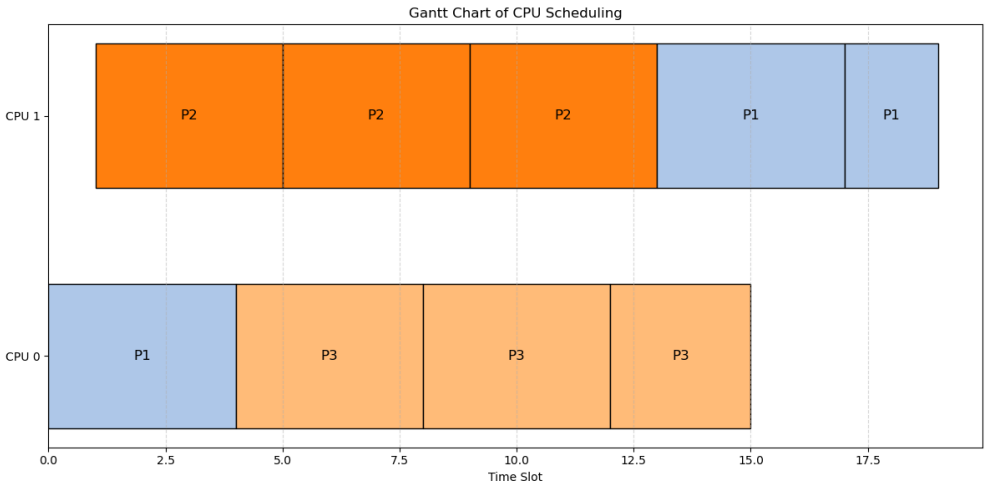

Figure 3: Biểu đồ Gantt của MLQ.

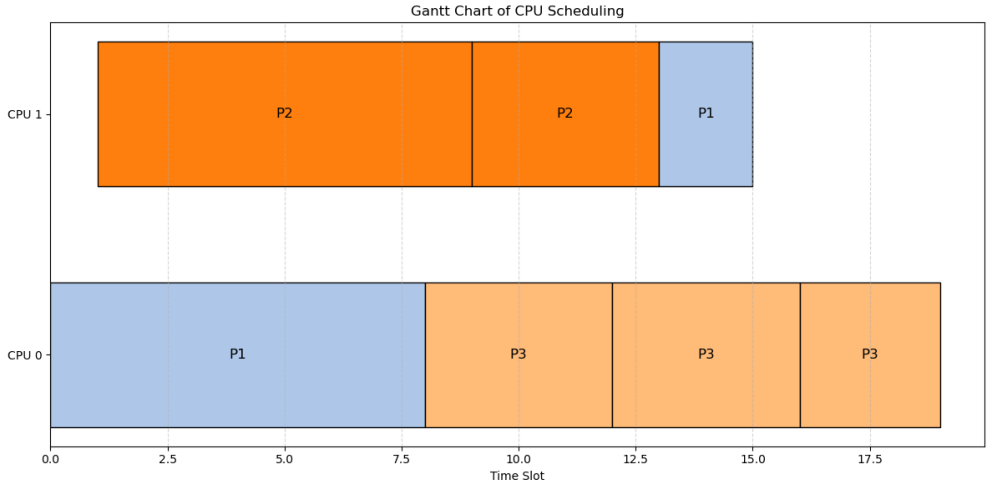

Figure 4: Biểu đồ Gantt của CFS.

Vì test trên chỉ để thấy được sự khác biệt cơ bản giữa 2 mô hình nên được thiết kế khá đơn giản, nhóm sẽ thiết kế một test khác cần thời gian chạy lâu hơn để thấy rõ được sự khác biệt trong cách phân bổ waiting time:

```
20 3 7
2048 16777216 0 0 0
0 10 -15
0 10 -10
0 10 10
0 11 -18
0 11 -5
0 11 8
0 11 17
```

Với `10` và `11` là 2 process chạy 120 và 160 lệnh. Trong test này nhóm để `time_slot` và `target_latency` đều là 20, giá trị prio sẽ được điều chỉnh tương ứng với niceness. Kết quả chạy như sau:

```
====================CFS======================
Turnaround time: 376
Process 1: 36
Process 2: 76
Process 3: 177
Process 4: 13
Process 5: 107
Process 6: 167
Process 7: 216
Average waiting time: 113.142857
Switch count: 138
===============================================
====================MLQ======================
Turnaround time: 417
Process 1: 0
Process 2: 0
Process 3: 157
```

```
Process 4: 107
Process 5: 123
Process 6: 140
Process 7: 257
Average waiting time: 112.000000
Switch count: 54
==========================================
```

Có thể thấy rằng, ở mô hình CFS, thời gian chờ tăng dần theo thứ tự của niceness, điều này thể hiện rằng CFS có ưu tiên các tiến trình có độ ưu tiên cao. Mặt khác, mô hình MLQ cũng đã có slot để làm giảm độ starvation cho các tiến trình prio cao. Đồng thời, số lần context switch của CFS lớn hơn MLQ rất nhiều, điều này có thể gây overhead khi hệ điều hành được hiện thực phức tạp hơn.

Tuy nhiên, hệ điều hành được nhóm hiện thực lại tương đối đơn giản, không có chi phí khi tiến hành context-switch. Do đó, nếu so sánh về hiệu suất, chỉ có thể so sánh xem Turn Around Time của mô hình nào thấp hơn. Nhưng nhóm sẽ không tiến hành so sánh về thời gian chạy (TAT) vì theo góc nhìn của nhóm, việc này là không có ý nghĩa. CFS sử dụng `target latency` để tính thời gian chạy, còn MLQ thì sử dụng `time slot` cố định; hai chỉ số này không tương quan với nhau và vì vậy, không thể chọn chỉ số hợp lý để so sánh xem mô hình nào định thời tốt hơn, cho ra TAT tốt hơn. Nếu muốn TAT thấp nhất có thể, ta chỉ cần chọn giá trị target latency và time slot nhỏ để mô hình trở thành định thời cho từng dòng lệnh (mỗi lần process chỉ được phép chạy 1 lệnh).

Dựa trên những phân tích ở trên, có thể nhận xét về ưu nhược điểm giữa MLQ và CFS như sau:

- **Completely Fair Scheduler:**
  - **Ưu điểm**:
    - **Công bằng**: Các process được sử dụng CPU theo trọng số, giảm starvation.
    - **Thời gian chạy không cố định**: Thời gian chạy của mỗi process được tính toán riêng, vì vậy không cần phải quan tâm đến việc process chiếm CPU quá lâu.
    - **Phù hợp cho hệ thống tương tác**: Vì tất cả process ít hoạt động đều được ưu tiên quay trở lại CPU nên phù hợp với hệ thống yêu cầu phản hồi nhanh.
  - **Nhược điểm**:
    - **Khó hiện thực**: Logic của CFS phức tạp và khó hiện thực hơn các thuật toán khác (nhất là phần hiện thực cây đỏ đen).
    - **Context Switch nhiều**: Vì khi có nhiều process đang đợi, lượng time slot được cung cấp cho process sẽ ít đi, gây tăng số lần switch.
- **Multi Level Queue Scheduler:**
  - **Ưu điểm**:
    - **Dễ hiện thực**: Mô hình nhiều hàng đợi, mỗi hàng đợi chạy Round-Robin tương đối dễ hiện thực.
    - **Tính ưu tiên cao**: Các process có độ ưu tiên cao sẽ được ưu tiên thực thi trước các process khác. Điều này giúp MLQ phù hợp với các hệ thống real-time, nhúng.
  - **Nhược điểm**:
    - **Thời gian chạy cố định**: Vì quantumn và slot của mỗi queue đều là cố định trong quá trình chạy nên phải tùy chỉnh dựa vào độ phức tạp của process. Chẳng hạn process gồm 1000 lệnh thì phải nâng quantumn và slot lên.

* **Starvation**: Các process có độ ưu tiên thấp phải chờ các hàng đợi có độ ưu tiên cao hết slot, gây starvation lớn. Ngược lại, nếu chỉnh slot quá thấp thì sẽ khiến các process cần ưu tiên không được thực hiện.

Phía trên là những ưu nhược điểm giữa CFS và MLQ mà nhóm nhận thấy trong quá trình hiện thực. Không thể chỉ dựa vào Turn Around Time hay Waiting Time để đánh giá xem mô hình nào mạnh hơn bởi vì TAT nhỏ sẽ làm Waiting Time và Context Switch lớn. Việc lựa chọn mô hình nào là tùy vào nhu cầu của người hiện thực muốn hệ thống trở nên công bằng hay muốn hoàn thành những công việc cần thiết trước.