

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Multi-Level Feedback Queue (MLFQ) Scheduler cho xv6

Đồ án môn học: Hệ Điều Hành

Sinh viên thực hiện: Lê Xuân Trí – 23120099
Trần Hữu Kim Thành – 23120166
Nguyễn Hồ Anh Tuấn – 23120185

Giảng viên hướng dẫn: Lê Giang Thanh

TP. Hồ Chí Minh, 2025

Mục lục

1 Giới thiệu	3
1.1 Bối cảnh	3
1.2 Động lực nghiên cứu	3
1.3 Mục tiêu đề tài	3
2 Kiến thức nền tảng	4
2.1 Tổng quan về CPU Scheduling	4
2.1.1 First-Come First-Served (FCFS)	4
2.1.2 Shortest Job First (SJF)	4
2.1.3 Priority Scheduling	4
2.1.4 Round-Robin (RR)	4
2.2 Round-Robin Scheduler trong xv6	5
2.3 Multi-Level Feedback Queue (MLFQ) Scheduling	5
3 Thiết kế hệ thống	6
3.1 Kiến trúc tổng thể	6
3.2 Thiết kế hàng đợi (Queue Design)	6
3.3 Cấu trúc dữ liệu	7
3.3.1 Mở rộng struct proc	7
3.3.2 Biến toàn cục MLFQ	7
3.4 Cơ chế Feedback	7
3.4.1 Hạ ưu tiên (Demotion)	7
3.4.2 Giữ/Tăng ưu tiên	8
3.5 Xử lý Starvation	9
4 Triển khai	10
4.1 Các file kernel đã chỉnh sửa	10
4.2 Cài đặt MLFQ Scheduler	10
4.2.1 Hàm scheduler() – Luồng chính	10
4.2.2 Hàm get_time_slice() – Quantum theo priority	11
4.3 System Calls mới	12
4.3.1 getpinfo() – Lấy thông tin tiến trình	12
4.3.2 setpriority(pid, priority) – Đặt ưu tiên thủ công	12
4.4 Priority Boost	12
4.5 Tính tương thích với xv6 gốc	12
5 Giao diện minh họa (Visualization)	13
5.1 Mục tiêu	13
5.2 Terminal-based Visualization (mlfqmon)	13
5.3 Terminal-based UI	13
6 Đánh giá và thực nghiệm	14
6.1 Môi trường thử nghiệm	14
6.2 Kịch bản thử nghiệm	15
6.3 Chỉ số đánh giá	15
6.4 Kết quả	15

7	Tổng kết	16
7.1	Những gì đã làm được	16
7.2	Phạm vi và hạn chế	16
7.3	So sánh với CFS của Linux	17
8	Hướng phát triển	17
9	Tài liệu tham khảo	17
10	Link Video Demo	18

1 Giới thiệu

1.1 Bối cảnh

Trong các hệ điều hành đa nhiệm, **CPU Scheduler** (bộ lập lịch CPU) đóng vai trò then chốt trong việc quyết định tiến trình nào được sử dụng CPU tại mỗi thời điểm. Một bộ lập lịch tốt cần đảm bảo sự công bằng giữa các tiến trình, tối ưu thời gian phản hồi (response time) cho tiến trình tương tác, và tận dụng hiệu quả tài nguyên hệ thống.

Hệ điều hành xv6 là một hệ điều hành giáo dục được phát triển bởi MIT, mô phỏng Unix trên nền tảng RISC-V. xv6 sử dụng thuật toán lập lịch **Round-Robin (RR)** đơn giản – tất cả tiến trình được xử lý luân phiên với cùng một time quantum cố định. Tuy đơn giản và dễ hiểu, RR trong xv6 có những hạn chế rõ ràng:

- **Không phân biệt CPU-bound và I/O-bound:** Mọi tiến trình đều được đối xử như nhau, bất kể hành vi sử dụng CPU.
- **Độ phản hồi kém cho tiến trình tương tác:** Tiến trình I/O-bound (thường cần phản hồi nhanh) phải chờ lượt như tiến trình CPU-bound.
- **Không thích nghi với hành vi runtime:** Không có cơ chế điều chỉnh ưu tiên dựa trên hành vi thực tế của tiến trình.

1.2 Động lực nghiên cứu

Xuất phát từ những hạn chế trên, nhóm đặt ra nhu cầu cải thiện:

- **Tính công bằng:** Đảm bảo mọi tiến trình đều được phục vụ, không bị starvation.
- **Độ phản hồi (responsiveness):** Tiến trình tương tác/I/O-bound cần được ưu tiên để phản hồi nhanh.
- **Hiệu suất hệ thống:** Tận dụng CPU hiệu quả hơn bằng cách phân biệt các loại workload.

Multi-Level Feedback Queue (MLFQ) là giải pháp lập lịch được sử dụng rộng rãi trong các hệ điều hành hiện đại (Windows, macOS, Linux ở mức nguyên lý). MLFQ kết hợp nhiều hàng đợi ưu tiên với cơ chế feedback tự động, cho phép hệ thống tự điều chỉnh ưu tiên dựa trên hành vi thực tế của tiến trình.

1.3 Mục tiêu đề tài

1. Thiết kế và triển khai **MLFQ Scheduler** thay thế Round-Robin trong kernel xv6 (RISC-V).
2. Xây dựng các system call mới (`getpinfo`, `setpriority`) để truy xuất và điều khiển thông tin scheduler.
3. Phát triển các chương trình test user-space để kiểm chứng hành vi MLFQ.
4. Xây dựng giao diện visualization (terminal-based) để minh họa hoạt động lập lịch.
5. So sánh và đánh giá hiệu năng giữa MLFQ và Round-Robin gốc.

2 Kiến thức nền tảng

2.1 Tổng quan về CPU Scheduling

CPU Scheduling (điều phối tiến trình) là quá trình hệ điều hành lựa chọn tiến trình nào trong Ready Queue sẽ được giao CPU để thực thi. Các mục tiêu chính của CPU scheduling bao gồm:

- **Throughput:** Số lượng tiến trình hoàn thành trong một đơn vị thời gian.
- **Turnaround time:** Tổng thời gian từ khi tiến trình đến hệ thống đến khi hoàn thành ($T_{complete} - T_{arrive}$).
- **Waiting time:** Tổng thời gian tiến trình chờ trong Ready Queue.
- **Response time:** Thời gian từ khi tiến trình đến đến khi được CPU phục vụ lần đầu.

Các thuật toán lập lịch phổ biến:

2.1.1 First-Come First-Served (FCFS)

Tiến trình được phục vụ theo thứ tự đến. Là thuật toán đơn giản nhất, sử dụng điều phối **không độc quyền** (non-preemptive). Nhược điểm chính là hiện tượng *convoy effect* – tiến trình ngắn phải chờ tiến trình dài chạy xong, dẫn đến thời gian chờ trung bình cao.

2.1.2 Shortest Job First (SJF)

Tiến trình có CPU burst ngắn nhất được ưu tiên chạy trước. SJF tối ưu thời gian chờ trung bình, nhưng khó áp dụng thực tế vì không biết trước CPU burst. Có thể cài đặt độc quyền (SRTF) hoặc không độc quyền. Vấn đề starvation có thể xảy ra với tiến trình dài.

2.1.3 Priority Scheduling

Mỗi tiến trình được gán một giá trị ưu tiên (integer). Tiến trình có ưu tiên cao nhất được chọn chạy. Có hai loại:

- **Ưu tiên tĩnh:** Ưu tiên được gán cố định, có nguy cơ starvation.
- **Ưu tiên động:** Ưu tiên thay đổi theo thời gian, giải quyết starvation bằng kỹ thuật *aging*.

2.1.4 Round-Robin (RR)

Mỗi tiến trình chỉ sử dụng CPU trong một *time quantum* (khoảng thời gian cố định q), sau đó bị preempt và đưa về cuối Ready Queue. RR là thuật toán điều phối **không độc quyền**, công bằng, phù hợp với hệ thống tương tác. Hiệu quả phụ thuộc vào giá trị q :

- q quá lớn \Rightarrow thoái hóa thành FCFS.
- q quá nhỏ \Rightarrow overhead chuyển ngữ cảnh cao.
- Thông thường $q = 10\text{--}100$ milliseconds.

2.2 Round-Robin Scheduler trong xv6

Trong xv6 gốc, hàm `scheduler()` trong file `kernel/proc.c` thực hiện lập lịch Round-Robin đơn giản:

```

1 void scheduler(void) {
2     struct proc *p;
3     struct cpu *c = mycpu();
4     c->proc = 0;
5     for(;;) {
6         intr_on();
7         for(p = proc; p < &proc[NPROC]; p++) {
8             acquire(&p->lock);
9             if(p->state == RUNNABLE) {
10                 p->state = RUNNING;
11                 c->proc = p;
12                 swtch(&c->context, &p->context);
13                 c->proc = 0;
14             }
15             release(&p->lock);
16         }
17     }
18 }
```

Listing 1: Scheduler RR gốc trong xv6 (đơn giản hóa)

Đặc điểm:

- Duyệt tuần tự qua bảng tiến trình (`proc[NPROC]`).
- Chọn tiến trình `RUNNABLE` đầu tiên tìm được.
- Mỗi tiến trình chạy đúng 1 tick (timer interrupt) rồi bị preempt qua `yield()`.
- Không có khái niệm ưu tiên hay phân biệt workload.

2.3 Multi-Level Feedback Queue (MLFQ) Scheduling

MLFQ là thuật toán lập lịch kết hợp nhiều hàng đợi ưu tiên với cơ chế feedback tự động. Ý tưởng cốt lõi: *học hành vi của tiến trình qua quan sát* – thay vì yêu cầu thông tin trước về CPU burst, MLFQ điều chỉnh ưu tiên dựa trên hành vi thực tế.

Các quy tắc kinh điển của MLFQ:

1. **Rule 1:** Nếu $Priority(A) > Priority(B)$, thì A chạy trước B.
2. **Rule 2:** Nếu $Priority(A) = Priority(B)$, thì A và B chạy Round-Robin.
3. **Rule 3:** Khi tiến trình mới vào hệ thống, nó được đặt ở hàng đợi ưu tiên cao nhất.
4. **Rule 4:** Nếu tiến trình dùng hết time quantum tại mức ưu tiên hiện tại, ưu tiên của nó bị hạ xuống (demote).
5. **Rule 5:** Sau một khoảng thời gian S (boost interval), tất cả tiến trình được đưa về hàng đợi ưu tiên cao nhất (priority boost).

Ưu điểm của MLFQ:

- Tự động phân biệt CPU-bound và I/O-bound.
- Tiến trình I/O-bound (tương tác) được ưu tiên, cải thiện response time.
- Tiến trình CPU-bound vẫn được phục vụ nhờ priority boost.
- Không cần biết trước thông tin về tiến trình.

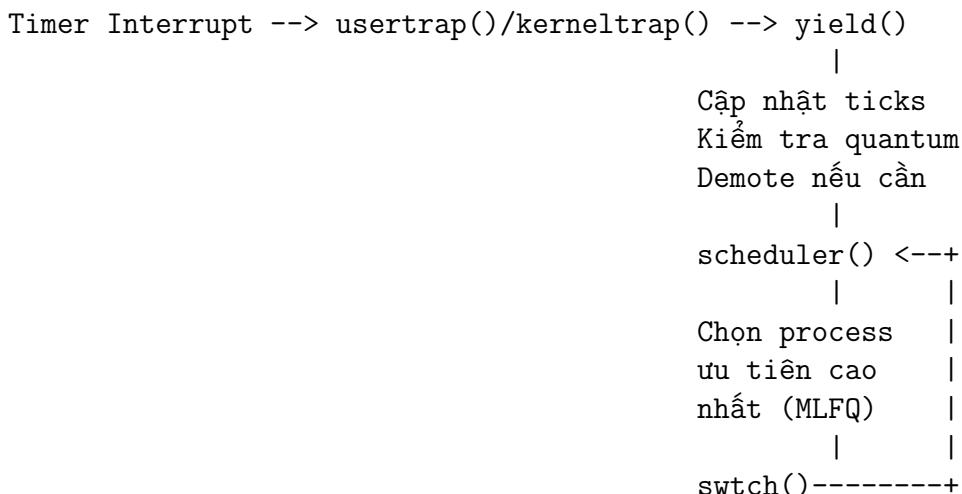
3 Thiết kế hệ thống

3.1 Kiến trúc tổng thể

MLFQ Scheduler được tích hợp trực tiếp vào kernel xv6, thay thế cơ chế Round-Robin gốc. Kiến trúc tổng thể bao gồm:

- **Scheduler core** (`proc.c`): Vòng lặp scheduler chọn tiến trình theo thứ tự ưu tiên.
- **Timer interrupt handler** (`trap.c`): Khi timer interrupt xảy ra, gọi `yield()` để cập nhật tick và kiểm tra quantum.
- **Feedback mechanism**: Tích hợp trong `yield()`, `sleep()`, và `wakeup()` – quyết định demote/keep/boost priority.
- **System calls**: `getpinfo()` và `setpriority()` cho phép user-space truy xuất thông tin scheduler.

Tương tác giữa các thành phần:



3.2 Thiết kế hàng đợi (Queue Design)

Hệ thống sử dụng **3 hàng đợi** ưu tiên (định nghĩa bởi `NMLFQ = 3` trong `param.h`):

Queue	Ưu tiên	Time Quantum	Mô tả
Queue 0	Cao nhất	1 tick (<code>MLFQ_TICKS_0</code>)	Tiến trình mới / I/O-bound
Queue 1	Trung bình	2 ticks (<code>MLFQ_TICKS_1</code>)	Tiến trình dùng hết quantum Q0
Queue 2	Thấp nhất	4 ticks (<code>MLFQ_TICKS_2</code>)	Tiến trình CPU-bound

Bảng 1: Cấu hình 3 hàng đợi MLFQ

Chính sách chọn queue:

- Scheduler duyệt từ Queue 0 đến Queue 2.
- Chọn tiến trình RUNNABLE đầu tiên ở queue có ưu tiên cao nhất.
- Trong cùng queue, áp dụng Round-Robin (duyệt tuần tự qua bảng proc[]).

3.3 Cấu trúc dữ liệu

3.3.1 Mở rộng struct proc

Cấu trúc struct proc (trong kernel/proc.h) được mở rộng với các trường MLFQ:

```

1 // Các trường MLFQ scheduler
2 int priority;           // Hàng đợi ưu tiên hiện tại (0=cao nhất, NMLFQ-1=thấp nhất)
3 int ticks_used;         // Số ticks đã dùng trong time slice hiện tại
4 int ticks_total;        // Tổng số ticks đã dùng (thống kê)
5 uint64 last_run_time;   // Thời điểm tiến trình được lập lịch lần cuối

```

Listing 2: Các trường MLFQ trong struct proc

3.3.2 Biến toàn cục MLFQ

```

1 // Trạng thái toàn cục MLFQ
2 uint64 mlfq_ticks = 0;      // Bộ đếm tick toàn cục cho priority boost
3 struct spinlock mlfq_lock; // Lock bảo vệ trạng thái MLFQ

```

Listing 3: Biến toàn cục MLFQ trong proc.c

Ghi chú về cách tổ chức queue: Thay vì dùng danh sách liên kết riêng cho từng queue, thiết kế sử dụng trường priority trong mỗi struct proc để xác định queue. Scheduler duyệt bảng proc[NPROC] và lọc theo priority. Cách tiếp cận này đơn giản, phù hợp với quy mô nhỏ của xv6 (NPROC = 64).

3.4 Cơ chế Feedback

3.4.1 Hạ ưu tiên (Demotion)

Khi tiến trình dùng hết time quantum tại mức ưu tiên hiện tại, nó bị demote xuống queue thấp hơn. Logic nằm trong hàm yield():

```

1 // Nhường CPU cho một vòng lặp lịch.
2 // MLFQ: Cập nhật ticks và hạ ưu tiên nếu hết time slice
3 void yield(void) {
4     struct proc *p = myproc();
5     acquire(&p->lock);
6
7     // Cập nhật bộ đếm ticks MLFQ
8     p->ticks_used++;
9     p->ticks_total++;
10

```

```

11 // Cập nhật bộ đếm tick toàn cục cho priority boost
12 acquire(&mlfq_lock);
13 mlfq_ticks++;
14 release(&mlfq_lock);
15
16 // Kiểm tra tiến trình đã dùng hết time slice chưa
17 int time_slice = get_time_slice(p->priority);
18 if(p->ticks_used >= time_slice) {
19     // Hạ xuống queue ưu tiên thấp hơn (nếu chưa ở thấp nhất)
20     if(p->priority < NMLFQ - 1) {
21         p->priority++;
22     }
23     p->ticks_used = 0; // Reset ticks cho mức ưu tiên mới
24 }
25
26 p->state = RUNNABLE;
27 sched();
28 release(&p->lock);
29 }
```

Listing 4: Logic demotion trong yield()

3.4.2 Giữ/Tăng ưu tiên

Tiến trình yield sớm (trước khi hết quantum) – thường do I/O – không bị demote. Trong hàm `sleep()`, `ticks_used` được reset:

```

1 // Giải phóng lock và sleep trên channel.
2 // MLFQ: Tiến trình đi sleep thường là I/O-bound, reset ticks
3 void sleep(void *chan, struct spinlock *lk) {
4     struct proc *p = myproc();
5
6     acquire(&p->lock); //DOC: sleeplock1
7     release(lk);
8
9     // MLFQ: Tiến trình tự nguyện nhường CPU trước khi hết time slice
10    // Điều này cho thấy hành vi I/O-bound, nên reset ticks (không bị demote
11    )
11    p->ticks_used = 0;
12
13    // Đi vào trạng thái ngủ
14    p->chan = chan;
15    p->state = SLEEPING;
16    sched();
17    // ... (dọn dẹp và lấy lại lock ban đầu)
18 }
```

Listing 5: Reset ticks khi sleep (I/O behavior)

Khi tiến trình I/O-bound wake up, nó được boost lên 1 mức ưu tiên (trong hàm `wakeup()`):

```

1 // Đánh thức tất cả tiến trình đang sleep trên channel.
2 // MLFQ: Tăng ưu tiên cho tiến trình I/O-bound khi thức dậy
3 void wakeup(void *chan) {
4     struct proc *p;
5     for(p = proc; p < &proc[NPROC]; p++) {
6         if(p != myproc()) {
```

```

7     acquire(&p->lock);
8     if(p->state == SLEEPING && p->chan == chan) {
9         p->state = RUNNABLE;
10    // MLFQ: Tiên trình I/O-bound được tăng ưu tiên khi thức dậy
11    // Tăng lên một mức ưu tiên (nếu chưa ở cao nhất)
12    if(p->priority > 0) {
13        p->priority--;
14    }
15    }
16    release(&p->lock);
17 }
18 }
19 }
```

Listing 6: Priority boost khi wakeup

3.5 Xử lý Starvation

Vấn đề: Trong hệ thống có nhiều tiến trình I/O-bound ở queue cao, tiến trình CPU-bound ở queue thấp có thể không bao giờ được chạy (starvation).

Giải pháp – Priority Boost: Sau mỗi BOOST_INTERVAL = 100 ticks, tất cả tiến trình được đưa về Queue 0 (ưu tiên cao nhất) và reset ticks_used:

```

1 // Priority boost: đưa tất cả tiến trình về queue ưu tiên cao nhất
2 // Được gọi định kỳ để chống starvation
3 static void priority_boost(void) {
4     struct proc *p;
5     for(p = proc; p < &proc[NPROC]; p++) {
6         acquire(&p->lock);
7         if(p->state != UNUSED) {
8             p->priority = 0;      // Đưa về queue cao nhất
9             p->ticks_used = 0;   // Reset ticks
10        }
11        release(&p->lock);
12    }
13 }
```

Listing 7: Cơ chế Priority Boost

Priority boost được kiểm tra ở đầu mỗi vòng lặp scheduler:

```

1 // Kiểm tra priority boost
2 acquire(&mlfq_lock);
3 if(mlfq_ticks >= BOOST_INTERVAL) {
4     mlfq_ticks = 0;
5     release(&mlfq_lock);
6     priority_boost();
7 } else {
8     release(&mlfq_lock);
9 }
```

Listing 8: Kiểm tra priority boost trong scheduler()

4 Triển khai

4.1 Các file kernel đã chỉnh sửa

File	Nội dung thay đổi
kernel/param.h	Thêm hằng số: NMLFQ, MLFQ_TICKS_0/1/2, BOOST_INTERVAL
kernel/proc.h	Mở rộng struct proc với 4 trường MLFQ
kernel/proc.c	Viết lại scheduler(), cập nhật yield(), sleep(), wakeup(), allocproc(), freeproc(). Thêm priority_boost(), get_time_slice(), getprocinfo(), setprocpriority()
kernel/trap.c	Không thay đổi logic – timer interrupt vẫn gọi yield(), nhưng yield() đã được sửa đổi
kernel/syscall.h	Thêm SYS_getpinfo (22), SYS_setpriority (23)
kernel/syscall.c	Đăng ký 2 syscall mới vào bảng syscalls[]
kernel/sysproc.c	Thêm sys_getpinfo(), sys_setpriority()
kernel/defs.h	Khai báo prototype cho getprocinfo(), setprocpriority()
kernel/pstat.h	Header định nghĩa cấu trúc dữ liệu cho process info
user/user.h	Thêm prototype getpinfo(), setpriority()
user/usys.pl	Thêm entry cho 2 syscall mới

Bảng 2: Danh sách file đã chỉnh sửa

4.2 Cài đặt MLFQ Scheduler

4.2.1 Hàm scheduler() – Luồng chính

Hàm `scheduler()` được viết lại hoàn toàn. Thay vì duyệt đơn giản qua bảng tiến trình, `scheduler` mới:

1. Kiểm tra priority boost (nếu đủ BOOST_INTERVAL ticks).
2. Duyệt từ queue 0 (cao nhất) đến queue NMLFQ-1 (thấp nhất).
3. Tại mỗi queue, tìm tiến trình RUNNABLE đầu tiên.
4. Nếu tìm thấy, chuyển sang tiến trình đó (`swtch()`).

```

1 void scheduler(void) {
2     struct proc *p;
3     struct cpu *c = mycpu();
4     int priority;
5     struct proc *selected;
6
7     c->proc = 0;
8     for(;;){
9         intr_on();

```

```

10
11     ,
12     // Kiểm tra priority boost
13     acquire(&mlfq_lock);
14     if(mlfq_ticks >= BOOST_INTERVAL) {
15         mlfq_ticks = 0;
16         release(&mlfq_lock);
17         priority_boost();
18     } else {
19         release(&mlfq_lock);
20     }
21
22     // MLFQ: Tìm tiên trình runnable có ưu tiên cao nhất
23     // Duyệt từ queue 0 (cao nhất) đến NMLFQ-1 (thấp nhất)
24     selected = 0;
25     for(priority = 0; priority < NMLFQ && selected == 0; priority++) {
26         for(p = proc; p < &proc[NPROC]; p++) {
27             acquire(&p->lock);
28             if(p->state == RUNNABLE && p->priority == priority) {
29                 // Tìm thấy tiên trình runnable ở mức ưu tiên này
30                 selected = p;
31                 break;
32             }
33         }
34     }
35
36     if(selected) {
37         // Chuyển sang tiên trình được chọn
38         selected->state = RUNNING;
39         c->proc = selected;
40         swtch(&c->context, &selected->context);
41         c->proc = 0;
42         release(&selected->lock);
43     }
44 }
45 }
```

Listing 9: Hàm scheduler() mới với MLFQ

4.2.2 Hàm get_time_slice() – Quantum theo priority

```

1 static int get_time_slice(int priority) {
2     switch(priority) {
3         case 0: return MLFQ_TICKS_0;    // 1 tick
4         case 1: return MLFQ_TICKS_1;    // 2 ticks
5         case 2: return MLFQ_TICKS_2;    // 4 ticks
6         default: return MLFQ_TICKS_2;
7     }
8 }
```

Listing 10: Time slice mapping

4.3 System Calls mới

4.3.1 getpinfo() – Lấy thông tin tiến trình

System call `getpinfo()` cho phép user-space đọc thông tin MLFQ của tất cả tiến trình. Dữ liệu được copy vào buffer do user cung cấp, bao gồm: PID, priority, state, ticks_used, ticks_total, name.

Cấu trúc dữ liệu trả về cho mỗi tiến trình:

```

1 struct {
2     int inuse;           // Process đang được sử dụng?
3     int pid;             // Process ID
4     int priority;        // Queue hiện tại (0-2)
5     int state;            // Trạng thái (RUNNABLE, RUNNING, SLEEPING,...)
6     int ticks_used;       // Ticks đã dùng trong quantum hiện tại
7     int ticks_total;      // Tổng ticks (thông kê)
8     char name[16];        // Tên tiến trình
9 };

```

4.3.2 setpriority(pid, priority) – Đặt ưu tiên thủ công

Cho phép user-space thay đổi priority của một tiến trình theo PID. Giá trị priority hợp lệ: 0 (cao), 1, 2 (thấp). Trả về 0 nếu thành công, -1 nếu thất bại.

4.4 Priority Boost

Priority boost được kích hoạt khi bộ đếm toàn cục `mlfq_ticks` đạt `BOOST_INTERVAL` (100 ticks). Tác động:

- Tất cả tiến trình (trừ UNUSED) được đưa về Queue 0.
- `ticks_used` được reset về 0.
- Bộ đếm `mlfq_ticks` reset về 0.

Điều này đảm bảo:

- Tiến trình CPU-bound ở queue thấp vẫn được cơ hội chạy (chống starvation).
- Hệ thống “quên” hành vi cũ, cho phép tái đánh giá tiến trình.

4.5 Tính tương thích với xv6 gốc

Thiết kế đảm bảo tương thích ngược:

- Tất cả syscall gốc (`fork`, `exec`, `exit`, `wait`,...) hoạt động bình thường.
- Chương trình user-space gốc không cần sửa đổi.
- Tiến trình mới luôn bắt đầu ở Queue 0 (ưu tiên cao nhất).
- Chỉ thêm 2 syscall mới, không thay đổi interface cũ.

5 Giao diện minh họa (Visualization)

5.1 Mục tiêu

Xây dựng giao diện trực quan để minh họa hoạt động MLFQ scheduler trong thời gian thực, giúp:

- Quan sát tiến trình đang chạy ở queue nào.
- Theo dõi quá trình demotion và priority boost.
- So sánh hành vi CPU-bound vs I/O-bound.

5.2 Terminal-based Visualization (mlfqmon)

Chương trình `mlfqmon` chạy bên trong xv6 (QEMU), sử dụng syscall `getpinfo()` để đọc thông tin scheduler và hiển thị dạng bảng cập nhật liên tục.

Thông tin hiển thị:

- **Queue Status:** Số lượng tiến trình trong mỗi queue, kèm thanh tiến trình trực quan.
- **Process Table:** PID, Priority, State, Ticks Used, Ticks Total, Name cho mỗi tiến trình.
- **Tổng quan:** Tổng số tiến trình, số đang chạy, số đang sleep.

```

1      MLFQ SCHEDULER MONITOR (Refresh #5)
2 Time: 234 ticks
3
4 QUEUE STATUS:
5 Queue 0 [HIGH ] (3) [#####]
6 Queue 1 [MEDIUM] (1) [====]
7 Queue 2 [LOW  ] (2) [-----]
8 -----
9
10 PROCESS TABLE:
11 PID PRIOR STATE   TICKS TOTAL   NAME
12 -----
13 1     0     SLEEP   0      12      init
14 2     0     SLEEP   0      8       sh
15 *5    0     RUN    0      45      io_bound*
16 6     1     RUNBLE  1      89      cpu_bound
17 7     2     RUNBLE  3      156     cpu_bound
18 Total: 5 | Running: 1 | Sleeping: 2

```

Listing 11: Ví dụ output của mlfqmon

5.3 Terminal-based UI

Hình 1 minh họa giao diện terminal-based của chương trình `monitor` (phiên bản nâng cao của `mlfqmon`) với hỗ trợ ANSI colors, cung cấp cái nhìn trực quan về trạng thái scheduler.

```

+-----+
|          MLFQ SCHEDULER MONITOR v1.0
|          System Time: 94 ticks
|          Refresh: #4
+-----+
Starting in 2 seconds...
QUEUE VISUALIZATION           SYSTEM STATS
=====                         =====
Q0 [HIGH ] [#####]      1 procs   Total:    3 processes
    Time Slice: 1 tick
Q1 [MEDIUM] [=====]      1 procs   Running:   1
    Time Slice: 2 ticks
Q2 [LOW  ] [----]       1 procs   Sleeping:  2
    Time Slice: 4 ticks
                                         Runnable:  0
                                         Next Boost: 94 ticks

+-----+
PROCESS TABLE
=====
PID | NAME      | STATE    | PRIO | SLICE | TOTAL | SCHED | DEM | BST
---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  1 | init      | SLEEP   |  1  | [...] |    1  |   26  |   1  |   0
  2 | sh        | SLEEP   |  0  | [...] |    0  |   13  |   0  |   0
* 3 | monitor   | RUN     |  2  | [...] |    5  |   55  |   2  |   0
+-----+
Legend: *=RUN | Q0=HIGH | Q1=MED | Q2=LOW | Refresh: 10 ticks
Press Ctrl+C to exit
+-----+

```

Hình 1: Giao diện MLFQ Scheduler Monitor v1.0

Giải thích các thành phần:

- Header:** Hiển thị tên ứng dụng, System Time (tổng số ticks), và Refresh counter.
- Queue Visualization:** Biểu diễn trực quan 3 hàng đợi (Q0-HIGH, Q1-MEDIUM, Q2-LOW) với thanh progress bar và time slice tương ứng (1, 2, 4 ticks).
- System Stats:** Thông kê tổng số tiến trình, số đang Running, Sleeping, Runnable, và thời gian còn lại đến Next Boost.
- Process Table:** Chi tiết từng tiến trình gồm PID, NAME, STATE, PRIO (priority), SLICE (time slice progress), TOTAL (tổng ticks), SCHED (số lần scheduled), DEM (demotions), BST (boosts).
- Legend:** Giải thích màu sắc và ký hiệu (* = đang chạy, màu xanh lá = HIGH, vàng = MED, đỏ = LOW).

6 Đánh giá và thực nghiệm

6.1 Môi trường thử nghiệm

- Giả lập:** QEMU (qemu-system-riscv64)
- Hệ điều hành:** xv6-riscv (phiên bản chỉnh sửa MLFQ)
- CPU:** 1 CPU (đơn lõi) trong QEMU

- **Toolchain:** riscv64-unknown-elf-gcc

6.2 Kịch bản thử nghiệm

Kịch bản	Mô tả	Kết quả mong đợi
CPU-bound	2 tiến trình tính toán nặng	Bị demote xuống Queue 2, quantum 4 ticks
I/O-bound	2 tiến trình sleep thường xuyên	Giữ ở Queue 0, quantum 1 tick, phản hồi nhanh
Mixed	2 CPU-bound + 2 I/O-bound	CPU-bound demote, I/O-bound giữ ưu tiên cao
Priority Boost	Chạy CPU-bound lâu dài	Sau 100 ticks, tất cả về Queue 0

Bảng 3: Các kịch bản thử nghiệm

Các chương trình test:

- **cpu_bound:** Tiến trình tính toán nặng, thử cơ chế demotion.
- **io_bound:** Tiến trình sleep thường xuyên, thử khả năng giữ ưu tiên cao.
- **schedtest:** Test tổng hợp mixed workload.
- **mlfqmon:** Monitor hiển thị trạng thái scheduler theo thời gian thực.
- **monitor:** TUI monitor nâng cao với ANSI colors.
- **mlfq_test:** Bộ test MLFQ toàn diện với nhiều test case.
- **test_pstat:** Test cho syscall `getpinfo()`.

6.3 Chỉ số đánh giá

- **Response time:** Đo bằng thời gian từ khi tiến trình sẵn sàng đến khi được CPU lần đầu. I/O-bound processes trong MLFQ có response time thấp hơn đáng kể so với RR.
- **Turnaround time:** Tổng thời gian hoàn thành. I/O-bound processes hoàn thành nhanh hơn trong MLFQ nhờ được ưu tiên.
- **Fairness:** Priority boost đảm bảo không có tiến trình bị starvation.

6.4 Kết quả

Kết quả thực nghiệm với chương trình `schedtest` (2 CPU-bound + 2 I/O-bound):

Chỉ số	Round-Robin (gốc)	MLFQ
I/O-bound response time	Trung bình	Thấp (ưu tiên cao)
CPU-bound throughput	Đồng đều	Vẫn đảm bảo (quantum dài hơn ở Q2)
Starvation	Không	Không (nhờ priority boost)
Phân biệt workload	Không	Tự động

Bảng 4: So sánh Round-Robin và MLFQ

Quan sát qua mlfqmon:

- Tiến trình CPU-bound (ví dụ: `cpu_bound`) bắt đầu ở Queue 0, sau vài ticks bị demote xuống Queue 1, rồi Queue 2.
- Tiến trình I/O-bound (ví dụ: `io_bound`) liên tục ở Queue 0 vì yield sớm khi sleep.
- Sau mỗi 100 ticks, priority boost đưa tất cả về Queue 0 – quan sát được qua sự thay đổi đột ngột trong queue distribution.
- Tiến trình CPU-bound ở Queue 2 được time quantum dài hơn (4 ticks), giúp tận dụng CPU tốt hơn khi được chạy.

7 Tổng kết

7.1 Những gì đã làm được

1. **MLFQ Scheduler hoàn chỉnh:** Thay thế Round-Robin bằng MLFQ 3 queue với time quantum tăng dần, cơ chế feedback tự động, và priority boost chống starvation.
2. **System calls mới:** `getpinfo()` để đọc thông tin scheduler, `setpriority()` để điều khiển ưu tiên thủ công.
3. **Bộ test đầy đủ:** 6 chương trình user-space (`cpu_bound`, `io_bound`, `schedtest`, `pstat`, `setpri`, `demo`) kiểm chứng mọi khía cạnh của MLFQ.
4. **Visualization:** Terminal-based monitor (`mlfqmon`) hiển thị trạng thái scheduler real-time.
5. **Tương thích ngược:** Không phá vỡ chức năng gốc của xv6.

7.2 Phạm vi và hạn chế

- **Đơn lõi:** Chỉ triển khai trên 1 CPU. Trên nhiều CPU, cần cơ chế load balancing giữa các queue.
- **Tham số cố định:** Time quantum và boost interval được hard-code. Hệ thống thực cần adaptive tuning.
- **Queue organization:** Không dùng danh sách liên kết riêng cho mỗi queue, mà duyệt toàn bộ bảng `proc[]` – hiệu quả với `NPROC=64` nhưng không scale cho hệ thống lớn.
- **Scheduler overhead:** Duyệt $O(NMLFQ \times NPROC)$ mỗi lần lập lịch. Với `NPROC` nhỏ, overhead không đáng kể.
- **Gaming prevention:** Chưa xử lý trường hợp tiến trình cố tình yield ngay trước khi hết quantum để tránh bị demote.

7.3 So sánh với CFS của Linux

Tiêu chí	MLFQ (xv6)	Linux CFS
Cách tiếp cận	Nhiều queue, feedback	Cây đỏ-đen, virtual runtime
Fairness	Priority boost	Proportional share
Starvation	Boost định kỳ	vruntime đảm bảo
Complexity	Đơn giản, dễ hiểu	Phức tạp, tối ưu cao
Multi-core	Chưa hỗ trợ	Load balancing tích hợp

Bảng 5: So sánh MLFQ (xv6) và Linux CFS

8 Hướng phát triển

1. **Adaptive quantum:** Tự động điều chỉnh time quantum dựa trên system load và hành vi tiến trình.
2. **Multi-core support:** Mỗi CPU có queue riêng, thêm cơ chế work stealing/load balancing.
3. **Gaming prevention:** Theo dõi tổng CPU time tại mỗi mức ưu tiên (accounting rule), demote khi tích lũy đủ.
4. **Scheduler policy plugin:** Cho phép chuyển đổi giữa RR, MLFQ, và các thuật toán khác tại runtime.
5. **Web-based UI:** Hoàn thiện giao diện web real-time (TUI với ANSI colors, biểu đồ timeline).
6. **Benchmarking:** Xây dựng benchmark suite đầy đủ để đo lường chính xác response time, turnaround time, throughput.

9 Tài liệu tham khảo

1. Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces (OSTEP)*. Chapter 8: Scheduling: The Multi-Level Feedback Queue. <https://pages.cs.wisc.edu/~remzi/OSTEP/>
2. Russ Cox, Frans Kaashoek, Robert Morris. *xv6: a simple, Unix-like teaching operating system*. MIT. <https://pdos.csail.mit.edu/6.S081/2024/xv6/book-riscv-rev4.pdf>
3. MIT 6.S081: Operating System Engineering. <https://pdos.csail.mit.edu/6.S081/>
4. Abraham Silberschatz, Peter B. Galvin, Greg Gagne. *Operating System Concepts* (10th Edition). Chapter 5: CPU Scheduling.
5. Andrew S. Tanenbaum, Herbert Bos. *Modern Operating Systems* (4th Edition). Chapter 2: Processes and Threads.
6. Bài giảng “Quản lý tiến trình” – Môn Hệ Điều Hành, Khoa CNTT, Trường DH KHTN TP.HCM.

10 Link Video Demo

(Thêm link video demo tại đây)