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: (Điền tên thành viên 1) – (MSSV)
(Điền tên thành viên 2) – (MSSV)
(Điền tên thành viên 3) – (MSSV)

Giảng viên hướng dẫn: (Điền tên GVHD)

TP. Hồ Chí Minh, 2025

Mục lục

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 $q = 10$–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:

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;) {
    intr_on();
    for(p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if(p->state == RUNNABLE) {
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
      }
      release(&p->lock);
    }
  }
}
```

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.

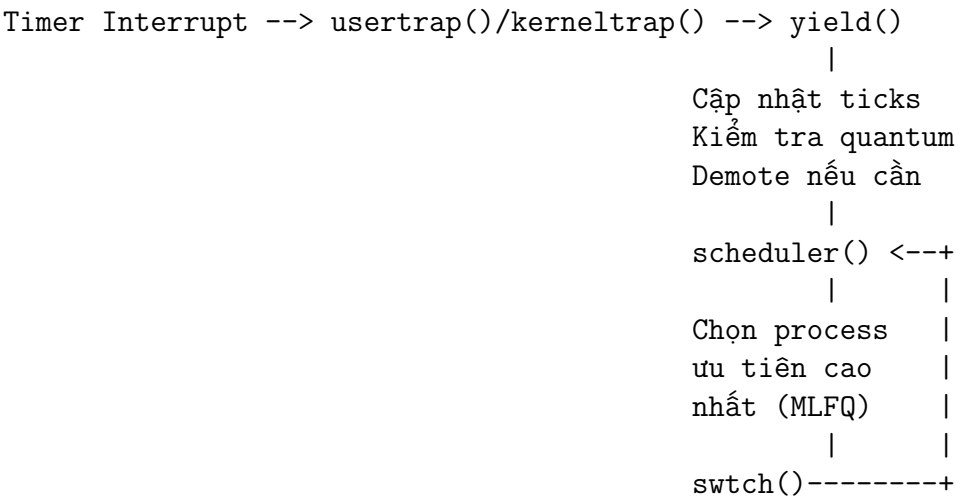
3 Thiết kế hệ thống

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 (`MLFQ_TICKS_0`)	Tiến trình mới / I/O-bound
Queue 1	Trung bình	2 ticks (`MLFQ_TICKS_1`)	Tiến trình dùng hết quantum Q0
Queue 2	Thấp nhất	4 ticks (`MLFQ_TICKS_2`)	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:

```
// MLFQ scheduler fields
int priority;        // Queue hi n t i (0=cao n h t , NMLFQ-1= t h p
    n h t )
int ticks_used;      //  S  ticks  d ng trong time slice h i n t i
int ticks_total;     //  T ng ticks  d ng ( t h ng k )
uint64 last_run_time; //  T h i  im  c h y l n c u i
```

Listing 2: Các trường MLFQ trong struct proc

3.3.2 Biến toàn cục MLFQ

```
uint64 mlfq_ticks = 0;      //  B  m  tick to n c c cho priority
    boost
struct spinlock mlfq_lock;   // Lock b o v mlfq_ticks
```

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()`:

```
void yield(void) {
  struct proc *p = myproc();
  acquire(&p->lock);

  p->ticks_used++;
  p->ticks_total++;

  //  C p  n h t  b  m  to n  c c  cho priority boost
  acquire(&mlfq_lock);
  mlfq_ticks++;
  release(&mlfq_lock);

```

```
  // Kiểm tra đến hạt time slice chưa
  int time_slice = get_time_slice(p->priority);
  if(p->ticks_used >= time_slice) {
    // Demote xuống queue thấp hơn
    if(p->priority < NMLFQ - 1)
      p->priority++;
    p->ticks_used = 0;
  }

  p->state = RUNNABLE;
  sched();
  release(&p->lock);
}
```

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:

```
void sleep(void *chan, struct spinlock *lk) {
  struct proc *p = myproc();
  acquire(&p->lock);
  release(lk);

  // I/O-bound: yield sớm, reset ticks (không bị demote)
  p->ticks_used = 0;

  p->chan = chan;
  p->state = SLEEPING;
  sched();
  // ...
}
```

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()`):

```
void wakeup(void *chan) {
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state == SLEEPING && p->chan == chan) {
      p->state = RUNNABLE;
      // I/O-bound: boost lên 1 mức
      if(p->priority > 0)
        p->priority--;
    }
    release(&p->lock);
  }
}
```

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`:

```
static void priority_boost(void) {
  struct proc *p;
  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    if(p->state != UNUSED) {
      p->priority = 0;      //   a    v   queue cao  n h t
      p->ticks_used = 0;    // Reset ticks
    }
    release(&p->lock);
  }
}
```

Listing 7: Cơ chế Priority Boost

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

```
//  K i m  tra priority boost
acquire(&mlfq_lock);
if(mlfq_ticks >= BOOST_INTERVAL) {
  mlfq_ticks = 0;
  release(&mlfq_lock);
  priority_boost();
} else {
  release(&mlfq_lock);
}
```

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()`
`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()`).

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  int priority;
  struct proc *selected;
  c->proc = 0;

  for(;;) {
    intr_on();

    // 1. Priority boost check
```

```
    acquire(&mlfq_lock);
    if(mlfq_ticks >= BOOST_INTERVAL) {
      mlfq_ticks = 0;
      release(&mlfq_lock);
      priority_boost();
    } else {
      release(&mlfq_lock);
    }

    // 2. MLFQ: Tim process uu tien cao nhat
    selected = 0;
    for(priority = 0; priority < NMLFQ && selected == 0; priority++) {
      for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == RUNNABLE && p->priority == priority) {
          selected = p;
          break;
        }
        release(&p->lock);
      }
    }

    if(selected) {
      selected->state = RUNNING;
      c->proc = selected;
      swtch(&c->context, &selected->context);
      c->proc = 0;
      release(&selected->lock);
    }
  }
}
```

Listing 9: Hàm scheduler() mới với MLFQ

4.2.2 Hàm `get_time_slice()` – Quantum theo priority

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

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:

```
struct {
  int inuse;           // Process  ang     c     s    d ng ?
  int pid;             // Process ID
  int priority;        // Queue  h i n  t i  (0-2)
  int state;           //  T r ng  th  i  (RUNNABLE, RUNNING, SLEEPING,...)
  int ticks_used;      // Ticks     d ng trong quantum  h i n  t i
  int ticks_total;     //  T ng  ticks ( t h ng  k  )
  char name[16];       // T n  t i n  tr nh
};
```

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.

```
        MLFQ SCHEDULER MONITOR (Refresh #5)
Time: 234 ticks

QUEUE STATUS:
Queue 0 [HIGH  ] (3) [######              ]
Queue 1 [MEDIUM] (1) [====                ]
Queue 2 [LOW   ] (2) [--------            ]
------------------------------------------------------------

PROCESS TABLE:
PID   PRIO  STATE   TICKS   TOTAL   NAME
------------------------------------------------------------
1     0     SLEEP   0       12      init
2     0     SLEEP   0       8       sh
*5    0     RUN     0       45      io_bound*
6     1     RUNBLE  1       89      cpu_bound
7     2     RUNBLE  3       156     cpu_bound
Total: 5 | Running: 1 | Sleeping: 2
```

Listing 11: Ví dụ output của mlfqmon

5.3 Web-based UI

(Phần này đang trong quá trình phát triển trên nhánh `feature/web-UI`. Bao gồm TUI monitor với ANSI colors và các chương trình test bổ sung. Sẽ được hoàn thiện trong giai đoạn tiếp theo.)

5.4 Kịch bản Demo

5.4.1 CPU-bound process

Chạy `cpu_bound`: tiến trình thực hiện tính toán nặng, dùng hết quantum ⇒ bị demote từ Queue 0 → Queue 1 → Queue 2.

5.4.2 I/O-bound process

Chạy `io_bound`: tiến trình sleep thường xuyên, yield sớm ⇒ giữ nguyên ở Queue 0.

5.4.3 Mixed workload

Chạy `schedtest` hoặc `demo`: tạo đồng thời cả hai loại, quan sát MLFQ tự động phân biệt và ưu tiên I/O-bound.

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

6.3 Chỉ số đánh giá

- **Response time:** Do 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 scheduler thực tế

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 ĐH KHTN TP.HCM.

10 Link Video Demo

(Thêm link video demo tại đây)