NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS
SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

UNDERGRADUATE THESIS

Priority and Lottery Scheduling in xv6: Implementation and Performance Analysis

2025

UNDERGRADUATE THESIS

Priority and Lottery Scheduling in xv6: Implementation and Performance Analysis

ABSTRACT

This thesis presents the implementation and evaluation of two scheduling algorithms—priority scheduling and lottery scheduling—in the `xv6` operating system. Priority scheduling ensures high-priority processes are executed first, while lottery scheduling allocates CPU time probabilistically based on ticket assignments. We detail the design, implementation, and testing of both schedulers, analyze their performance under different CPU configurations (1 to 5 CPUs), compare their results, and propose future improvements.

SUBJECT AREA: Operating Systems
KEYWORDS: scheduling, priority, lottery, `xv6`, performance

Contents

List of Figures

List of Tables

1 INTRODUCTION

Process scheduling is a fundamental pillar of operating system design, determining how CPU resources are allocated among competing processes to balance performance, responsiveness, and fairness. The xv6 operating system, a lightweight Unix-like teaching system developed by MIT, provides an ideal platform for exploring scheduling algorithms due to its simplicity and C-based implementation. Motivated by a passion for operating systems and low-level C programming, this thesis implements two contrasting schedulers in xv6: priority scheduling, which prioritizes processes based on fixed priority levels, and lottery scheduling, which allocates CPU time probabilistically based on ticket assignments. Inspired by discussions with my academic advisor, this work aims to deepen the understanding of scheduling algorithms through their implementation and performance analysis in a real operating system.

The priority scheduler was designed to replace xv6's default round-robin scheduler, selecting processes with the highest priority for execution, which is critical for systems requiring predictable task prioritization. In contrast, the lottery scheduler promotes fairness by allocating CPU time proportionally to the number of tickets, making it suitable for general-purpose systems. To evaluate these schedulers, comprehensive test suites were developed. For priority scheduling, tests measure completion time and context switch overhead across CPU-intensive, I/O-bound, mixed, and short-lived workloads, comparing performance to the default round-robin scheduler. For lottery scheduling, tests focus on fairness, measuring CPU time distribution across processes with varying ticket counts and workloads. Both schedulers were tested on configurations with 1 to 5 CPUs to assess scalability and performance under different system loads.

The objectives of this thesis are threefold: (1) to design and implement robust priority and lottery schedulers in xv6, (2) to evaluate their performance and fairness using customized test suites, with metrics such as completion time, context switch overhead, and CPU time proportionality, and (3) to compare their behavior across different CPU configurations and propose improvements. By implementing and testing these schedulers, this work not only explores their theoretical trade-offs but also demonstrates practical skills in modifying a real operating system, providing insights into the strengths and limitations of deterministic and probabilistic scheduling approaches.

2 BACKGROUND AND RELATED WORK

2.1 xv6 Operating System

The xv6 operating system, developed by MIT, is a teaching operating system based on Unix Version 6. Its simplicity and clean C-based implementation make it an ideal platform for experimenting with operating system concepts, such as process scheduling. xv6 supports a multi-CPU environment and manages processes through a process table protected by a spinlock.

2.2 Round-Robin Scheduler

The default scheduler in xv6 uses a round-robin algorithm, allocating equal time slices to all runnable processes in a cyclic order. As implemented in the `scheduler` function in `proc.c`, the system iterates through the process table, selecting the first process in the RUNNABLE state for execution. This ensures fairness but does not prioritize processes based on their importance or workload. To support performance evaluation, the base implementation was extended with system calls (`getcontextswitches`, `yield`) and process fields (`start_ticks`, `run_ticks`, `wait_ticks`, `end_ticks`) to track timing metrics. The `timingtests.c` test suite evaluates the scheduler across various workloads, providing a baseline for comparing the priority and lottery schedulers. The simplicity and fairness of the round-robin scheduler make it a suitable benchmark for assessing improvements introduced by priority and lottery scheduling.

2.3 Priority Scheduling

Priority scheduling assigns fixed priorities to processes, ensuring that the process with the highest priority executes first. This deterministic approach is ideal for systems requiring predictable execution of critical tasks but may lead to starvation of low-priority processes. The implementation in this thesis replaces the round-robin scheduler, introducing priority-based process selection and a new system call for setting priorities.

2.4 Lottery Scheduling

Lottery scheduling, proposed by Waldspurger and Weihl, allocates CPU time probabilistically based on ticket assignments. Each process is assigned a number of tickets, and the scheduler selects the next process by drawing a random ticket, with the selection probability proportional to the number of tickets. This approach ensures fairness over time, making it suitable for general-purpose systems. The implementation introduces ticket management and random selection mechanisms, supported by new system calls.

3 DESIGN AND IMPLEMENTATION

3.1 Priority Scheduling

The priority scheduler replaces `xv6`'s round-robin scheduler with a deterministic priority-based approach, assigning priorities from 0 to 10 (0 being the highest). It leverages per-CPU runqueues for scalability across multiple CPUs [6]. Key modifications include:

- **Process Structure**: The `struct proc` in `proc.h` adds fields: `priority` (0–10), `next`, `wait_ticks`, `creation_time`, `completion_time`, `waiting_time`, `last_runnable_tick`, `first_run_time`, `has_run`, `cpu_time`, and `cpu`. The default priority is 5 in `allocproc`.
- **Per-CPU Runqueues**: Each CPU maintains a `struct runqueue` with 11 priority queues (0–10) and a FIFO queue for priority 5 (short-lived processes), managed by `rq_init`, `rq_add`, `rq_remove`, and `rq_select` in `runqueue.c`. Processes are added to their priority queue, and `rq_select` chooses the highest-priority process.
- **Scheduler Logic**: The `scheduler` function in `proc.c` selects the highest-priority process via `rq_select`:

Κώδικας 1: Process Selection for Priority Scheduling

```
struct proc *p = rq_select(&c->rq);
if (!p) {
    release(&ptable_lock);
    sti();
    continue;
}
c->proc = p;
switchuvm(p);
p->state = RUNNING;
p->waiting_time += ticks - p->last_runnable_tick;
p->last_runnable_tick = ticks;
if (!p->has_run) {
    p->first_run_time = ticks;
    p->has_run = 1;
}
context_switches++;
swtch(&(c->scheduler), p->context);
```

- **Priority Selection Mechanism**: The `rq_select` function ensures deterministic scheduling by first checking the priority 5 FIFO queue for short-lived processes. If empty, it scans priority queues from 0 to 10, selecting the first process from the highest non-empty queue. This guarantees that higher-priority processes are always scheduled first. The pseudocode is:

Κώδικας 2: Pseudocode for Process Selection in Priority Scheduling

```
function rq_select(rq):
    acquire(rq.lock)
    if rq.count == 0:
        release(rq.lock)
        return null
    if rq.short_lived_head != null:
        p = rq.short_lived_head
        rq.short_lived_head = p.next
        if p.next == null:
            rq.short_lived_tail = null
        p.next = null
        rq.count -= 1
        release(rq.lock)
        return p
    for prio in 0 to 10:
        if rq.priority_head[prio] != null:
            p = rq.priority_head[prio]
            rq.priority_head[prio] = p.next
            if p.next == null:
                rq.priority_tail[prio] = null
            p.next = null
            rq.count -= 1
            release(rq.lock)
            return p
    release(rq.lock)
    return null
```

- **Priority Aging**: The `update_priorities` function, called every 50 ticks, increases the priority of waiting processes (reducing the priority number) to prevent starvation, except for processes with PID 1 and 2 (critical system processes). Processes with PID > 100 are reset to priority 5, and those running for over 10,000 ticks are terminated, except for PID 1 and 2.
- **Load Balancing**: In `fork`, new processes are assigned to the CPU with the fewest processes, reducing contention.
- **System Calls**: The `setpriority` system call sets priorities (0–10). A `sched_log` records events for debugging, disabled during tests.
- **Integration with xv6**: Processes become runnable (e.g., via `wakeup`) and are added to the runqueue using `rq_add`, placing them in the appropriate priority queue. Selected processes are removed via `rq_remove`. This per-CPU approach replaces global process table scans, enhancing scalability [6].

The priority scheduler was tested using `prioritytest.c`, which extends the round-robin test suite by assigning priorities to specific tests (I/O-bound, Mixed Load, Starvation Check). For example, in the starvation check, a lightweight process is given priority 0, while heavy processes receive priority 10, ensuring rapid completion of the lightweight process. The implementation balances responsiveness for high-priority tasks with fairness via aging, making it suitable for systems requiring predictable prioritization.

3.2 Lottery Scheduling

The lottery scheduler employs a probabilistic ticket-based approach, using per-CPU runqueues for scalability [6]. Key modifications include:

- **Process Structure**: The `struct proc` adds fields: `tickets` (default 1), `ticks_scheduled`, `recent_schedules`, `last_scheduled`, and `cpu`, initialized in `allocproc`.
- **Per-CPU Runqueues**: Each CPU's `struct runqueue` can hold up to 64 processes, managed by `rq_init`, `rq_add`, `rq_remove`, and `rq_select` in `runqueue.c`.
- **Scheduler Logic**: The `scheduler` selects a process via `rq_select`:

Κώδικας 3: Process Selection for Lottery Scheduling

```
p = rq_select(&c->rq, sched_count);
if (p == 0) {
    acquire(&ptable_lock);
    release(&ptable_lock);
    sti();
    continue;
}
acquire(&ptable_lock);
rq_remove(&c->rq, p);
c->proc = p;
switchuvm(p);
p->state = RUNNING;
p->ticks_scheduled++;
p->recent_schedules++;
p->last_scheduled = ticks;
swtch(&(c->scheduler), p->context);
```

- **Lottery Selection Mechanism**: The `rq_select` function computes the total tickets, shuffles the process array to avoid bias, and selects a process by generating a random number within the ticket range. The pseudocode is:

Κώδικας 4: Pseudocode for Process Selection in Lottery Scheduling

```
function rq_select(rq, sched_count):
    acquire(rq.lock)
    if rq.count == 0:
        release(rq.lock)
        return null
    total_tickets = 0
    for each process p in rq.procs:
        if p != null:
            effective_tickets = max(p.tickets, 1)
            total_tickets += effective_tickets
    if total_tickets == 0:
        release(rq.lock)
        return null
    temp_procs = list of non-null processes in rq.procs
    shuffle(temp_procs)
    winner = random(0, total_tickets - 1)
    current_tickets = 0
```

```
    for each p in temp_procs:
        effective_tickets = max(p.tickets, 1)
        if winner < current_tickets + effective_tickets:
            release(rq.lock)
            return p
        current_tickets += effective_tickets
    release(rq.lock)
    return null
```

- **Random Number Generation**: A linear congruential generator in `rand.c` provides random numbers for ticket selection [2].
- **Starvation Prevention**: Every 100 scheduling decisions, `recent_schedules` decays by multiplying by 3/4, reducing bias and ensuring fairness.
- **Load Balancing**: The `fork` function assigns processes to the CPU with the least total tickets.
- **System Calls**: The `settickets`, `settickets_pid`, `getpinfo`, and `yield` system calls support lottery scheduling.
- **Integration with xv6**: Processes are added to the runqueue via `rq_add` when runnable and removed via `rq_remove` when scheduled, replacing global process table scans with per-CPU runqueues [6].

The lottery scheduler was tested using `lotterytest.c`, evaluating fairness across six tests with varying ticket counts and workloads. The implementation ensures probabilistic fairness, scalability, and responsiveness, making it suitable for general-purpose systems prioritizing equitable CPU allocation.

4 TEST METHODOLOGY

The evaluation of the round-robin, priority, and lottery schedulers relies on custom test suites executed across configurations with 1 to 5 CPUs. The round-robin and priority schedulers are tested using `timingtests.c` and `prioritytest.c`, respectively, measuring completion time and context switch overhead across various workloads. The lottery scheduler is evaluated with `lotterytest.c`, focusing on fairness through CPU time proportionality.

4.1 Round-Robin Tests

The `timingtests.c` suite includes seven tests, each run five times to compute average metrics:

- **CPU-heavy**: 10 processes, each performing 20 million iterations, measuring completion time and context switches.
- **Switch Overhead**: 200 processes that terminate immediately, testing the overhead of `fork` and context switches.
- **I/O-bound**: 100 processes with 10-tick sleeps, evaluating I/O handling.
- **Mixed Load**: 5 CPU-intensive processes (50 million iterations) and 5 I/O-bound processes (50-tick sleeps), reporting the minimum completion time.
- **Process Creation**: 50 processes that terminate immediately, testing `fork` overhead.
- **Low Process Count**: 200 processes with 10,000 iterations, evaluating short-lived task performance.
- **Starvation Check**: 1 lightweight process (50,000 iterations) versus 5 heavy processes (20 million iterations), testing for starvation.

Metrics include completion time (measured in ticks via `uptime`) and context switches (via `getcontextswitches`). These tests provide a baseline for round-robin scheduler performance.

4.2 Priority Tests

The `prioritytest.c` suite extends `timingtests.c` with priority assignments to evaluate the priority scheduler's performance. It includes seven tests, each run five times to compute average metrics:

- **CPU-heavy**: 10 processes, each performing 20 million iterations, using the default priority (5), measuring completion time and context switches.
- **Switch Overhead**: 200 processes that terminate immediately, using the default priority (5), testing `fork` and context switch overhead.

- **I/O-bound**: 50 processes with short CPU bursts and 1-tick sleeps; the first 25 processes receive priority 5, and the last 25 receive priority 0 (highest), evaluating priority-based I/O handling.
- **Mixed Load**: 5 CPU-intensive processes (50 million iterations) with priority 0 and 5 I/O-bound processes (50-tick sleeps) with priority 10 (lowest), reporting the minimum completion time.
- **Process Creation**: 50 processes that terminate immediately, using the default priority (5), testing `fork` overhead.
- **Low Process Count**: 200 processes with 10,000 iterations, using the default priority (5), evaluating short-lived task performance.
- **Starvation Check**: 1 lightweight process (50,000 iterations) with priority 0 versus 5 heavy processes (20 million iterations) with priority 10, ensuring high-priority processes complete quickly.

Metrics include completion time (measured in ticks via `uptime`) and context switches (via `getcontextswitches`). The tests verify that high-priority processes complete faster, particularly in prioritization scenarios.

4.3 Lottery Tests

The `lotterytest.c` suite evaluates the lottery scheduler's fairness across six tests, each run five times to compute average CPU time distribution:

- **Low Process Count**: 3 processes with tickets (30, 20, 10), expecting CPU time distribution of 50%, 33%, 16%.
- **Basic Fairness**: 8 processes with tickets (30, 30, 20, 20, 10, 10, 5, 5), expecting grouped proportions (46%, 31%, 15%, 8%).
- **Switch Overhead**: 50 processes in three groups (17 with 30 tickets, 17 with 20, 16 with 10), expecting CPU time distribution of 50%, 33%, 16%.
- **Starvation Check**: 8 processes with tickets (50, 50, 10, 10, 1, 1, 1, 1), expecting grouped proportions (81%, 16%, 3%).
- **Grouped Ticket Levels**: 30 processes in three groups (10 with 1 ticket, 10 with 5, 10 with 10), expecting CPU time distribution of 6%, 31%, 62%.
- **Mixed Load**: 20 processes (5 CPU-intensive with 20 tickets, 5 I/O-bound with 15, 5 short-lived with 10, 5 mixed with 5), expecting CPU time distribution of 40%, 30%, 20%, 10%.

The primary metric is CPU time proportionality, measured as the percentage of scheduling events (`ticks_scheduled`) retrieved via the `getpinfo` system call. Each test spawns child processes, sets ticket counts using `settickets`, and collects statistics after a 50-tick sleep. Results are reported with one decimal place accuracy (e.g., 50.3%) to account for randomness. These tests verify the lottery scheduler's ability to allocate CPU time proportionally to ticket counts across diverse workloads, including CPU-heavy, I/O-bound, and mixed scenarios.

5 RESULTS

This section presents the performance metrics for the round-robin, priority, and lottery schedulers across configurations with 1 to 5 CPUs, based on the test suites described in Section 6. Results for the round-robin and priority schedulers include average completion time (in ticks) and average context switches (CS) per execution for each test. The lottery scheduler results focus on average CPU time proportions across five executions for each test, reflecting fairness in ticket-based allocation.

5.1 Round-Robin Scheduler Results

The `timingtests.c` suite measures completion time and context switches for seven tests. Table 1 summarizes the average metrics across 1 to 5 CPUs, with context switches calculated as the average of five executions per test.

Table 1: Round-Robin Scheduler: Average Completion Time and Context Switches per Execution

Test	**1 CPU**	**2 CPUs**	**3 CPUs**	**4 CPUs**	**5 CPUs**
Completion Time (Ticks/Execution) – Context Switches (CS/Execution)					
CPU-heavy	86 – 100	46 – 108	33 – 115	28 – 129	23 – 131
Switch Overhead	72 – 471	82 – 505	102 – 540	110 – 554	116 – 557
I/O-bound	51 – 1151	54 – 1176	59 – 1240	64 – 1256	66 – 1257
Mixed Load	50 – 185	50 – 259	50 – 365	50 – 398	50 – 421
Process Creation	19 – 69	19 – 77	23 – 92	25 – 101	27 – 105
Low Process Count	77 – 277	79 – 308	95 – 372	102 – 406	109 – 423
Starvation Check	46 – 55	23 – 55	17 – 60	14 – 63	14 – 73

5.2 Priority Scheduler Results

The `prioritytest.c` suite evaluates the priority scheduler with the same tests, incorporating priority assignments. Table 2 summarizes the average completion time and context switches per execution across 1 to 5 CPUs.

Table 2: Priority Scheduler: Average Completion Time and Context Switches per Execution

Test	1 CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs
Completion Time (Ticks/Execution) – Context Switches (CS/Execution)					
CPU-heavy	84 – 94	54 – 108	41 – 118	35 – 127	28 – 131
Switch Overhead	71 – 470	92 – 490	114 – 509	126 – 521	127 – 521
I/O-bound	27 – 577	30 – 586	38 – 605	44 – 595	44 – 609
Mixed Load	50 – 242	50 – 291	50 – 347	50 – 354	50 – 396
Process Creation	18 – 68	21 – 72	24 – 74	26 – 77	31 – 82
Low Process Count	74 – 274	85 – 288	95 – 296	102 – 304	121 – 321
Starvation Check	44 – 50	27 – 53	29 – 55	28 – 56	33 – 60

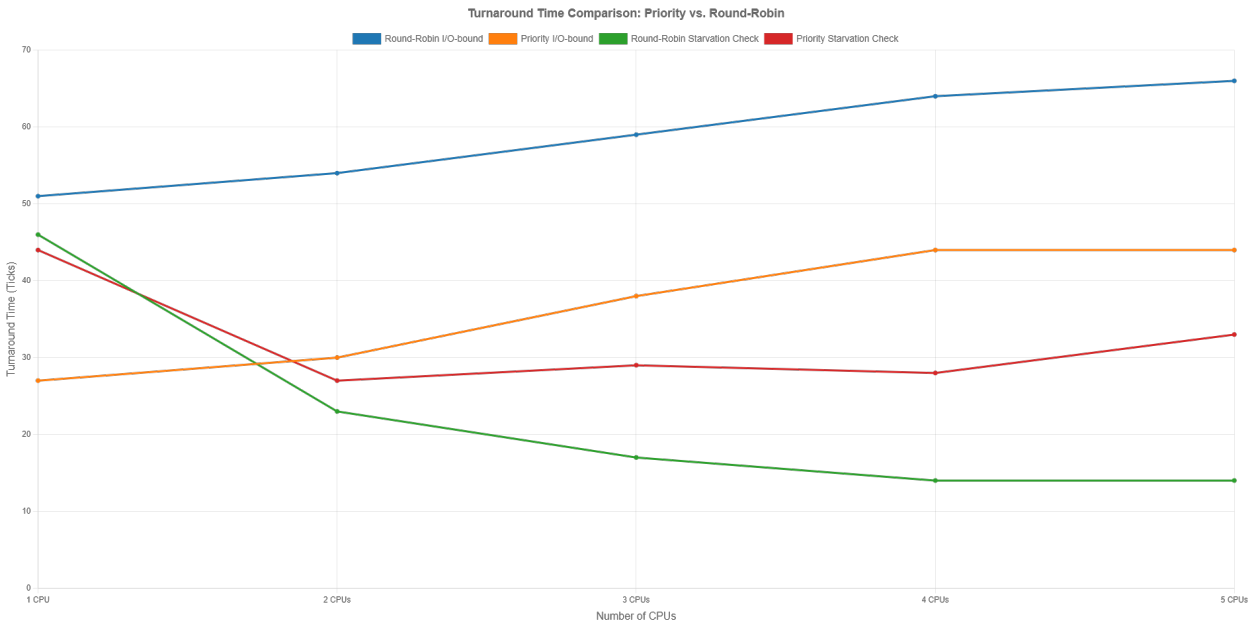

Figure 1: Completion Time Comparison: Priority vs. Round-Robin Schedulers

Note: Blue lines represent the Round-Robin I/O-bound test, orange lines the Priority I/O-bound test, green lines the Round-Robin Starvation Check test, and red lines the Priority Starvation Check test.

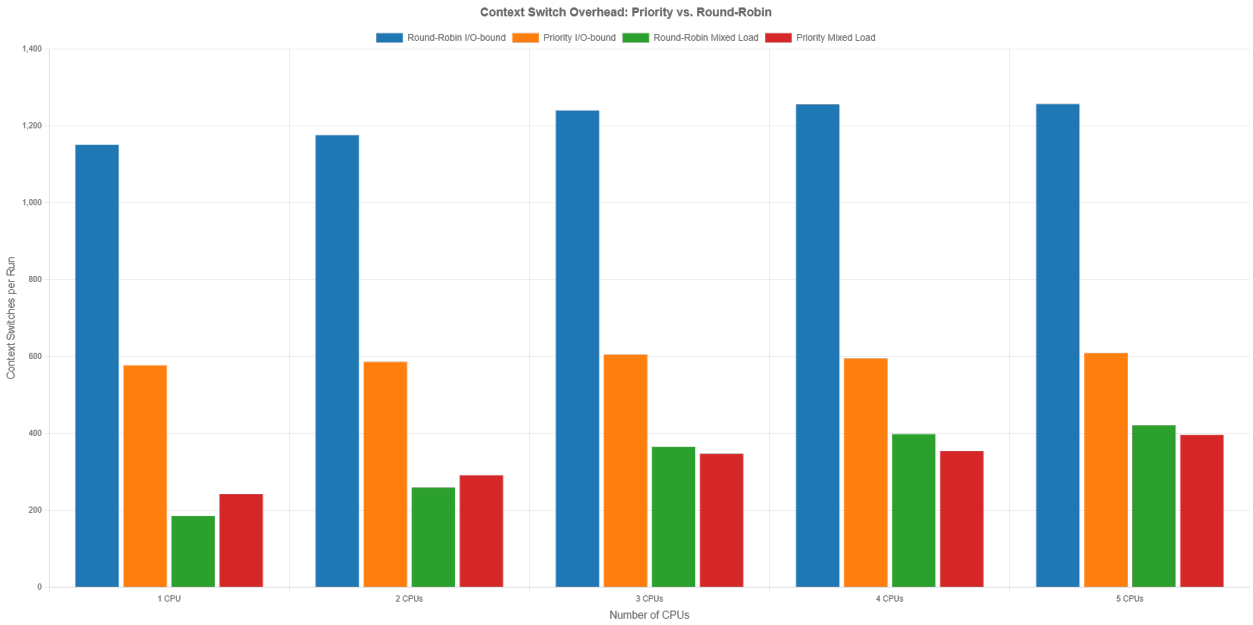

Figure 2: Context Switch Comparison: Priority vs. Round-Robin Schedulers

Note: Blue lines represent the Round-Robin I/O-bound test, orange lines the Priority I/O-bound test, green lines the Round-Robin Mixed Load test, and red lines the Priority Mixed Load test.

5.3 Lottery Scheduler Results

The `lotterytest.c` suite measures CPU time proportionality for six tests, reported as percentages of scheduling events. Tables 3 to 8 summarize the average proportions across five executions for each test across 1 to 5 CPUs, compared to expected proportions.

Table 3: Lottery Scheduler: Low Process Count (Expected: 50%, 33%, 16%)

Process	1 CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs
A (30 tickets)	50.4	37.6	38.7	41.7	43.8
B (20 tickets)	33.0	43.3	35.7	34.3	34.9
C (10 tickets)	16.4	12.2	25.5	23.9	21.1

Table 4: Lottery Scheduler: Basic Fairness (Expected: 46%, 31%, 15%, 8%)

Group	1 CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs
A+B (30+30)	46.9	47.8	42.3	41.7	32.7
C+D (20+20)	30.8	30.9	33.5	26.4	24.7
E+F (10+10)	15.0	13.6	16.9	20.7	21.2
G+H (5+5)	7.0	7.4	7.2	11.0	21.2

Table 5: Lottery Scheduler: Switch Overhead (Expected: 50%, 33%, 16%)

Group	1 CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs
A (30 tickets)	50.1	50.9	50.2	48.7	49.2
B (20 tickets)	32.2	31.7	32.0	33.2	33.0
C (10 tickets)	17.5	17.3	17.6	18.0	17.7

Table 6: Lottery Scheduler: Starvation Check (Expected: 81%, 16%, 3%)

Group	1 CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs
A+B (50+50)	66.3	44.7	37.7	43.8	39.8
C+D (10+10)	28.1	37.2	36.6	27.4	23.2
E+F+G+H (1+1+1+1)	5.5	17.9	25.5	28.7	36.8

Table 7: Lottery Scheduler: Grouped Ticket Levels (Expected: 6%, 31%, 62%)

Group	1 CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs
1 (10×1)	7.0	10.6	12.4	13.4	13.6
2 (10×5)	33.2	34.5	34.3	35.3	34.7
3 (10×10)	59.7	54.7	53.1	51.1	51.6

Table 8: Lottery Scheduler: Mixed Load (Expected: 40%, 30%, 20%, 10%)

Group	1 CPU	2 CPUs	3 CPUs	4 CPUs	5 CPUs
CPU-heavy (5×20)	64.5	63.9	65.2	63.5	63.8
I/O-bound (5×15)	18.7	18.1	17.8	18.2	18.9
Short-lived (5×10)	0.3	0.3	0.3	0.3	0.3
Mixed (5×5)	16.3	17.5	16.5	17.9	16.8

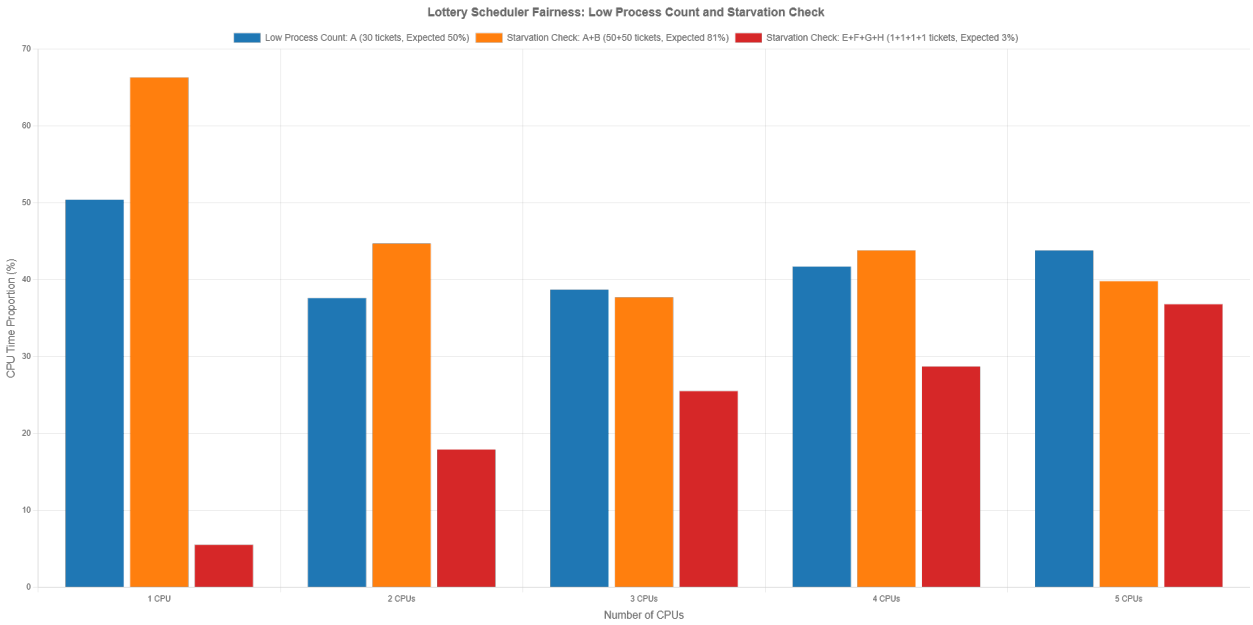

Figure 3: CPU Time Proportionality: Lottery Scheduler across 1 to 5 CPUs

Note: Blue lines represent the Low Process Count test: A (30 tickets, Expected 50%), orange lines the Starvation Check test: A+B (50+50 tickets, Expected 81%), and red lines the Starvation Check test: E+F+G+H (1+1+1+1 tickets, Expected 3%).

6 ANALYSIS

This section provides a detailed interpretation of the performance results for the round-robin, priority, and lottery schedulers, as presented in Section 5. It examines completion time, context switches, and CPU time proportionally across 1 to 5 CPUs, linking observed trends to specific design choices from Section 3. The analysis evaluates each scheduler's effectiveness in terms of responsiveness, fairness, and scalability, highlighting how implementation details shape performance and identifying limitations.

6.1 Round-Robin Scheduler Analysis

The round-robin scheduler results reflect its design as a fair, non-prioritizing algorithm that allocates equal time slices to all runnable processes via cyclic selection in the `scheduler` function. In the CPU-heavy test, completion time decreases from 86 ticks (1 CPU) to 23 ticks (5 CPUs), as 10 processes execute in parallel across multiple CPUs. However, context switches (CS) increase from 100 to 131 due to the spinlock-protected process table, which introduces contention as CPUs coordinate access. The Switch Overhead test, with 200 processes terminating immediately, shows an increase in ticks (72 to 116) and context switches (471 to 557), as frequent `fork` and `exit` operations amplify scheduling overhead due to the lack of prioritization.

The I/O-bound test, with 100 processes sleeping for 10 ticks, exhibits stable context switches (1151–1257) but increasing ticks (51 to 66) with more CPUs, indicating that I/O-induced context switches (via `yield` calls) are less parallel due to sleep-wake cycles. The Mixed Load test maintains a constant 50 ticks across all CPUs, as it measures the minimum completion time (dominated by I/O-bound processes with 50-tick sleeps), but context switches rise from 185 to 421, reflecting the scheduler's need to handle diverse workloads. The Process Creation and Low Process Count tests show moderate increases in ticks (19 to 27, 77 to 109) and context switches (69 to 105, 277 to 423), consistent with `fork` overhead and frequent scheduling. The Starvation Check test reduces ticks from 46 to 14 with more CPUs, as the lightweight process (50,000 iterations) benefits from parallel execution, but context switches increase slightly (55 to 73) due to contention with heavy processes.

The round-robin scheduler's strength lies in its guaranteed fairness, as all processes receive equal CPU time, evident in the balanced context switch distribution. However, the lack of prioritization leads to inefficiencies for critical tasks. For example, in the Starvation Check test, the lightweight process waits 46 ticks on 1 CPU because it is scheduled only after heavy processes, highlighting the design's inability to prioritize high-priority tasks. The increase in context switches with more CPUs underscores the scalability limitations of the centralized process table and spinlock, which could be mitigated with per-CPU runqueues, as implemented in the other schedulers.

6.2 Priority Scheduler Analysis

The priority scheduler results, derived from the `prioritytest.c` suite, demonstrate its ability to optimize high-priority processes through deterministic prioritization, as implemented in `rq_select` with per-CPU priority queues. In the CPU-heavy test, completion time de-

creases from 84 ticks (1 CPU) to 28 ticks (5 CPUs), slightly worse than round-robin (86 to 23) due to the overhead of managing priority queues, despite load balancing in `fork`. Context switches are comparable (94–131 vs. 100–131 for round-robin), reflecting the added complexity of priority-based selection.

The I/O-bound test, with 50 processes (first 25 at priority 5, last 25 at priority 0), achieves lower ticks (27–44 vs. 51–66 for round-robin), as high-priority processes (priority 0) execute immediately, reducing wait time. Context switches are lower (577–609 vs. 1151–1257) because the test uses 50 processes instead of 100, and prioritized scheduling reduces unnecessary switches. The Mixed Load test, with 5 CPU-heavy processes at priority 0 and 5 I/O-bound processes at priority 10, shows 50 ticks across all CPUs, identical to round-robin, as the minimum completion time is dominated by I/O-bound processes. However, context switches are lower (242–396 vs. 185–421) due to prioritized scheduling reducing contention. The Starvation Check test improves ticks (44 to 33 vs. 46 to 14 for round-robin), with context switches slightly lower (50–60 vs. 55–73), as the lightweight process (priority 0) is prioritized over heavy processes (priority 10).

The priority scheduler's responsiveness for high-priority tasks, particularly in the I/O-bound and Starvation Check tests, stems from its 11 priority queues and the `update_priorities` function, which increases priority every 50 ticks to prevent starvation. In the Starvation Check test, low-priority processes do not excessively block the high-priority process due to aging. However, tests with uniform priorities (e.g., CPU-heavy, Switch Overhead) show performance similar to round-robin, indicating workload dependency. The increase in context switches with more CPUs reflects the overhead of queue management, but per-CPU runqueues ensure scalability. The 50-tick aging interval may be too slow for dynamic workloads, delaying low-priority processes.

The effectiveness of the priority scheduler's aging mechanism in preventing starvation is evident in the Starvation Check test, where the lightweight process (priority 0) completes in 44 ticks on 1 CPU and 33 ticks on 5 CPUs, significantly faster than round-robin's 46 to 14 ticks, despite competition with five heavy processes at priority 10. This indicates that aging successfully promotes low-priority processes over time, preventing indefinite delays. However, in the I/O-bound test, the 25 processes at priority 5 experience slightly higher completion times (part of the 27–44 ticks) compared to priority 0 processes, suggesting that the 50-tick aging interval may not escalate priorities quickly enough for dynamic workloads with frequent I/O operations. A shorter aging interval could reduce delays for lower-priority processes, aligning with theoretical recommendations for adaptive aging [5].

6.3 Lottery Scheduler Analysis

The lottery scheduler results (Tables 3 to 8), derived from the `lotterytest.c` suite, highlight its probabilistic fairness, driven by random ticket-based selection in `rq_select` and per-CPU runqueues (Section 3) [2]. The Low Process Count test achieves proportions close to expected (50.4%, 33.0%, 16.4% vs. 50%, 33%, 16%) on 1 CPU but deviates on 5 CPUs (43.8%, 34.9%, 21.1%) as per-CPU runqueues fragment ticket distributions, and the linear congruential generator in `rand.c` may introduce variability for small process counts. The Basic Fairness test maintains stable proportions (A+B: 46–47% on 1–2 CPUs) but drops to 32.7% on 5 CPUs, with low-ticket groups (G+H) gaining excessive time (21.2% vs. 8%) due to runqueue fragmentation.

The Switch Overhead test, with 50 processes, is consistent (50–51%, 31–33%, 17–18%)

across all CPUs, as the larger number of scheduling events smooths out randomness, and `rq_select` randomization performs well with balanced ticket distributions. The Starvation Check test shows high-ticket groups (A+B) receiving 39.8–66.3% vs. expected 81%, with low-ticket groups (E+F+G+H) gaining excessive time (5.5–36.8% vs. 3%), indicating starvation risks due to infrequent `recent_schedules` decay (every 100 scheduling decisions). The Grouped Ticket Levels test shows the low-ticket group 1 over-allocated (7–13.6% vs. 6%) and the high-ticket group 3 under-allocated (59.7–51.6% vs. 62%), reflecting load balancing issues in `fork`. The Mixed Workload test shows CPU-heavy processes dominating (64–65% vs. 40%) and short-lived tasks receiving negligible time (0.3% vs. 20%) as they terminate quickly before being scheduled, limiting ticket-based allocation.

The lottery scheduler's performance on a single CPU is fair for large process counts due to effective randomization. Fairness on multiple CPUs degrades due to runqueue fragmentation and load balancing in `fork`, which assigns processes based on static ticket counts without runtime adjustments. The decay every 100 decisions is insufficient for low-ticket processes, and the random number generator's quality affects small tests. Scalability is strong, but fairness is undermined for unbalanced workloads, particularly in the Mixed Workload test, where short-lived processes are underserved.

The ticket decay mechanism, reducing `recent_schedules` by 3/4 every 100 scheduling decisions, aims to prevent starvation by limiting bias toward recently scheduled processes [2]. However, its effectiveness is limited, as seen in the Starvation Check test (Table 6), where low-ticket processes (1 ticket each) receive 5.5–36.8% CPU time across 1–5 CPUs, far exceeding the expected 3%, while high-ticket processes (50 tickets each) are underallocated (39.8–66.3% vs. 81%). This suggests that the decay interval is too infrequent to ensure fairness for low-ticket processes, especially in multi-CPU configurations where runqueue fragmentation exacerbates imbalances. In the Mixed Workload test (Table 8), short-lived processes (10 tickets) receive only 0.3% CPU time (vs. 20% expected), partly due to their quick termination but also because the decay mechanism fails to prioritize them over CPU-heavy processes (20 tickets, 64–65% vs. 40%). A more frequent decay (e.g., every 50 decisions) or a minimum scheduling guarantee could enhance fairness, ensuring low-ticket processes receive proportional CPU time in shorter intervals [4].

7 COMPARISON

This section compares the priority and lottery schedulers implemented in `xv6`, based on their performance results (Section 5), design choices (Section 3), and test methodology. The comparison evaluates responsiveness (ability to prioritize critical tasks), fairness (equitable CPU time allocation), scalability (performance across 1 to 5 CPUs), and suitability for different workloads (CPU-intensive, I/O-bound, mixed, and short-lived).

7.1 Responsiveness

The priority scheduler excels in responsiveness for high-priority tasks, as evident in the I/O-bound and Starvation Check tests (Table 2). In the I/O-bound test, completion times are significantly lower (27–44 ticks vs. 51–66 ticks for round-robin) because the scheduler prioritizes the 25 processes with priority 0, executing them immediately. Similarly, in the Starvation Check test, the lightweight process (priority 0) completes faster (44–33 ticks vs. 46–14 ticks for round-robin), with fewer context switches (50–60 vs. 55–73), as heavy processes (priority 10) are deferred. The deterministic selection in `rq_select`, using 11 priority queues, ensures predictable execution for critical tasks, supported by the `setpriority` system call for dynamic priority adjustments. The aging mechanism (`update_priorities`, every 50 ticks) mitigates starvation, ensuring low-priority processes eventually execute, though the 50-tick interval may delay responsiveness in dynamic workloads.

In contrast, the lottery scheduler lacks deterministic prioritization, relying on probabilistic ticket-based selection (Section 3). In the Starvation Check test (Table 6), high-ticket processes (A+B, 50 tickets each) receive only 39.8–66.3% CPU time (vs. expected 81%), while low-ticket processes (1 ticket each) gain excessive time (5.5–36.8% vs. 3

7.2 Fairness

The lottery scheduler achieves better long-term fairness, allocating CPU time proportionally to ticket counts, as seen in the Switch Overhead and Basic Fairness tests (Tables 5 and 4). In the Switch Overhead test, proportions are close to expected (49.2–50.9%, 31.7–33.2%, 17.3–18.0% vs. 50%, 33%, 16%) across 1–5 CPUs, reflecting the effectiveness of ticket-based randomized selection in `rq_select` for large scheduling events. The Basic Fairness test shows stable proportions for high-ticket groups (A+B: 41.7–47.8% vs. 46%) across 1–4 CPUs, though fairness degrades at 5 CPUs (32.7%) due to runqueue fragmentation. The `getpinfo` system call provides transparency into scheduling statistics, and the decay mechanism ensures low-ticket processes are not completely starved, though the 100-decision interval is too slow for small tests (e.g., Starvation Check).

The priority scheduler, by design, sacrifices fairness for responsiveness, as low-priority processes may be delayed indefinitely without aging. In the Mixed Load test (Table 2), CPU-intensive processes (priority 0) complete faster, but I/O-bound processes (priority 10) contribute to the 50-tick completion time, indicating potential delays for low-priority tasks. The aging mechanism mitigates this, but the fixed 50-tick interval is less adaptive than the lottery's probabilistic approach. In uniform-priority tests (e.g., CPU-heavy, Switch Overhead), the priority scheduler behaves like round-robin, with comparable completion times (84–28 vs. 86–23 ticks) and context switches (95–131 vs. 100–131), offering no

fairness advantage. Thus, the lottery scheduler is better suited for general-purpose systems prioritizing equitable CPU allocation over time.

7.3 Scalability

Both schedulers leverage per-CPU runqueues to enhance scalability, but their performance varies with increasing CPU counts. The priority scheduler maintains consistent performance across 1–5 CPUs, with completion times decreasing in CPU-heavy (84 to 28 ticks) and Starvation Check (44 to 28 ticks) tests, though context switches increase (e.g., 95 to 131, 50 to 60) due to queue management overhead. Load balancing in `fork`, assigning new processes to the CPU with the fewest processes, ensures even distribution, but spinlock-protected queues cause contention at higher CPU counts, as seen in the Switch Overhead test (CS: 470 to 521). The priority scheduler's scalability is robust for prioritized workloads but limited by aging and queue overhead in uniform-priority scenarios.

The lottery scheduler scales well for large process counts, as seen in the Switch Overhead test, where proportions remain stable across CPUs. However, fairness degrades with more CPUs in small tests (e.g., Low Process Count: 50.4–43.8% for A across 1–5 CPUs) due to runqueue fragmentation, where processes are distributed across CPUs, reducing ticket proportionality. Load balancing in `fork`, based on total tickets, is static and does not adapt to dynamic changes, exacerbating the issue. The random number generator (`rand.c`) adds variability, particularly in small tests, but does not significantly impact scalability. Both schedulers are scalable, but the priority scheduler is more predictable, while the lottery scheduler's fairness is sensitive to CPU count and process distribution.

7.4 Suitability for Workloads

The priority scheduler is ideal for workloads requiring strict prioritization, such as I/O-bound or mixed workloads with critical tasks. In the I/O-bound test, it reduces completion times (27–44 ticks) by prioritizing high-priority processes, and in the Mixed Load test, it ensures efficient completion of CPU-heavy tasks. However, for CPU-heavy or short-lived workloads with uniform priorities, it offers no significant advantage over round-robin (e.g., CPU-heavy: 84–28 vs. 86–23 ticks). The scheduler struggles with fairness in scenarios with many low-priority processes, as seen in potential delays for I/O-bound processes in the Mixed Load test.

The lottery scheduler suits general-purpose workloads where fairness is prioritized, such as CPU-intensive or mixed workloads with varied ticket assignments. In the Grouped Ticket Levels test (Table 7), it allocates CPU time close to expected proportions (7–13.6%, 33.2–35.3%, 59.7–51.6% vs. 6%, 31%, 62%) for large process counts but struggles with short-lived tasks in the Mixed Load test (0.3% vs. 20%) due to their rapid termination. Its probabilistic nature makes it less suitable for real-time systems or workloads requiring predictable execution, as seen in the Starvation Check test deviations. For I/O-bound workloads, it performs adequately but lags in responsiveness compared to the priority scheduler.

7.5 Conclusion

The priority scheduler is superior for systems requiring high responsiveness and predictable execution of critical tasks, such as real-time or I/O-intensive applications, but sacrifices fairness for low-priority processes, with aging only partially mitigating this. The lottery scheduler excels in fairness, ensuring proportional CPU allocation over time, making it suitable for general-purpose systems with diverse workloads, but its probabilistic approach limits responsiveness and fairness at higher CPU counts due to runqueue fragmentation. Both schedulers scale well with per-CPU runqueues, but the priority scheduler is more predictable, while the lottery scheduler's performance depends on workload size and ticket distribution. These trade-offs highlight the priority scheduler's strength in deterministic environments and the lottery scheduler's advantage in equitable resource allocation, aligning with their design goals in `xv6`.

8 FUTURE IMPROVEMENTS

This section proposes improvements for the priority and lottery schedulers, based on their performance results (Section 5), analysis, and comparison (Section 7). The suggestions aim to address limitations in responsiveness, fairness, and scalability, enhancing the schedulers' suitability for diverse workloads and multi-CPU environments in `xv6`. Each proposal is grounded in observed performance trends and design choices (Section 3), offering practical modifications to improve efficiency.

8.1 Priority Scheduler Improvements

The priority scheduler exhibits strong responsiveness for high-priority tasks but faces limitations in fairness for low-priority processes and scalability in uniform-priority workloads. The following improvements address these issues:

- **Dynamic Aging Interval**: The current aging mechanism in `update_priorities` increases process priorities every 50 ticks, delaying low-priority processes in dynamic workloads, as seen in the Mixed Load test, where I/O-bound processes (priority 10) contribute to the 50-tick completion time (Table 2). Implementing a dynamic aging interval, adjusted based on workload characteristics (e.g., shorter intervals for I/O-bound processes identified via high `wait_ticks`), would improve fairness. For example, `update_priorities` could reduce the interval to 25 ticks if `wait_ticks` exceeds a threshold (e.g., 100 ticks), ensuring faster priority escalation for stalled processes. This requires modifying `proc.h` to track workload type hints and updating `update_priorities` logic.
- **Workload-Based Priority Assignment**: The priority scheduler performs similarly to round-robin in uniform-priority tests (e.g., CPU-heavy: 84–28 ticks vs. 86–23 ticks, Table 2), indicating inefficiency when priorities are not leveraged. Introducing a heuristic in the kernel to assign initial priorities based on process behavior (e.g., CPU-heavy processes identified via `run_ticks` receive lower priority, I/O-bound processes via `wait_ticks` receive higher) would enhance responsiveness. This could be implemented in `allocproc` by analyzing `run_ticks` and `wait_ticks` after a brief execution period (e.g., 100 ticks), setting `priority` accordingly. The `setpriority` system call would remain available for manual overrides, balancing automation with flexibility.
- **Improved Load Balancing**: The current load balancing in `fork`, assigning new processes to the CPU with the fewest processes, does not consider priority distribution, leading to contention in the Switch Overhead test (CS: 470–521, Table 2). A load balancer that accounts for priorities, assigning processes to CPUs with the lowest priority-weighted load (e.g., sum of `10 - priority` for all processes), would reduce contention. This requires modifying `fork` to compute weighted load for each CPU runqueue and updating `runqueue.c` to provide priority sums via `rq_stats`. This improvement would enhance scalability for prioritized workloads.
- **Queue Overhead Reduction**: The 11 priority queues per CPU increase context switch overhead in uniform-priority tests (e.g., CPU-heavy CS: 95–131, Table 2), as `rq_select` scans multiple queues. Reducing the number of queues (e.g., to 5 levels:

0–2, 3–5, 6–8, 9–10) and using a bitmap to track non-empty queues would optimize selection time. This involves modifying `runqueue.h` to redefine `struct runqueue` with fewer queues and a `nonempty_bitmap` field, updating `rq_select` to check the bitmap first, potentially halving selection overhead while maintaining prioritization granularity.

These improvements would make the priority scheduler more adaptive to dynamic workloads, fairer for low-priority processes, and more scalable, particularly for uniform-priority scenarios, aligning with its goal of deterministic prioritization.

8.2 Lottery Scheduler Improvements

The lottery scheduler achieves probabilistic fairness but faces issues with runqueue fragmentation, short-lived process allocation, and starvation in low-ticket scenarios. The following improvements address these limitations:

- **Dynamic Load Balancing**: The static load balancing in `fork`, assigning processes to the CPU with the fewest total tickets, causes runqueue fragmentation, degrading fairness in small tests (e.g., Low Process Count: process A drops from 50.4% to 43.8% on 5 CPUs, Table 3). A dynamic load balancer, periodically migrating processes to balance ticket sums across CPUs (e.g., every 100 ticks via a new `balance_tickets` function), would improve fairness. This requires modifying `runqueue.c` to support process migration between runqueues and updating `scheduler` to trigger `balance_tickets`, using `ticks_scheduled` to identify under- or over-scheduled processes. This would stabilize proportions in multi-CPU tests.
- **Enhanced Starvation Prevention**: The `recent_schedules` decay every 100 scheduling decisions is too infrequent, leading to starvation in the Starvation Check test (low-ticket processes receive 5.5–36.8% vs. 3%, Table 6). Reducing the decay interval to 50 decisions and introducing a minimum scheduling guarantee (e.g., scheduling low-ticket processes every 200 ticks if `ticks_scheduled` is below a threshold) would ensure fairer allocation. This involves updating `rq_select` to check `last_scheduled` and enforce selection of starved processes, modifying `proc.h` to track starvation metrics, and adjusting decay logic in `runqueue.c`.
- **Short-Lived Process Handling**: The Mixed Load test shows short-lived processes receiving negligible CPU time (0.3% vs. 20%, Table 8) due to rapid termination before scheduling. Introducing a minimum ticket allocation period (e.g., guaranteeing at least one scheduling event within 10 ticks of process creation) would improve allocation. This could be implemented by modifying `rq_add` to flag new processes with a `newly_created` field in `proc.h`, ensuring `rq_select` prioritizes them once before reverting to ticket-based selection. This would enhance fairness for short-lived tasks without compromising long-term proportionality.
- **Improved Random Number Generation**: The linear congruential generator in `rand.c` introduces variability in small tests (e.g., Low Process Count on 2 CPUs: A at 37.6%, C at 12.2%, Table 3), affecting fairness. Replacing it with a higher-quality generator, such as a tailored Mersenne Twister for `xv6`, would reduce variability. This requires rewriting `rand.c` to implement the new algorithm and updating `rq_select` to use `rand_range` with the improved generator, ensuring more uniform ticket selection in small-scale tests.

- **Adaptive Ticket Adjustment**: Static ticket assignments via `settickets` limit responsiveness for dynamic workloads, as seen in the Mixed Load test where CPU-heavy processes dominate (64.5–65.2% vs. 40%, Table 8). An adaptive ticket adjustment mechanism, increasing tickets for I/O-bound processes (identified via high `wait_ticks`) and decreasing for CPU-heavy processes (high `run_ticks`), would balance allocation. This could be implemented in `scheduler` by periodically calling a new `adjust_tickets` function, using `getpinfo` data to guide adjustments, improving responsiveness for interactive workloads.

These improvements would enhance the lottery scheduler's fairness across varying CPU counts, ensure better allocation for short-lived and low-ticket processes, and improve responsiveness for dynamic workloads, aligning with its goal of proportional resource allocation.

8.3 General Improvements

Both schedulers exhibit common limitations in multi-CPU scalability and adaptability to workloads, suggesting the following general improvements:

- **Hybrid Scheduling Approach**: Combining priority and lottery scheduling could leverage both strengths. For example, assigning priority levels to process groups and using lottery scheduling within each level would provide deterministic prioritization for critical tasks (e.g., I/O-bound) and probabilistic fairness within priority groups. This requires redesigning `runqueue.h` to support hybrid runqueues, modifying `rq_select` to select a priority level first and then draw a ticket, and extending system calls (`setpriority`, `settickets`) to support group assignments. This would improve responsiveness and fairness for mixed workloads, as seen in the priority scheduler's I/O-bound test (27–44 ticks, Table 2) and the lottery scheduler's Switch Overhead test (50.1–50.9%, Table 5).
- **Advanced Load Balancing**: Both schedulers rely on static load balancing in `fork`, which is suboptimal for dynamic workloads (e.g., priority CS: 470–521, lottery fairness degradation in Low Process Count, Tables 2, 3). Implementing a global load balancer, running periodically (e.g., every 500 ticks) to redistribute processes based on runtime metrics (`run_ticks`, `tickets`, or `priority`), would reduce contention and improve scalability. This could be implemented as a new `global_balance` function in `proc.c`, using per-CPU statistics from `runqueue.c`, benefiting both schedulers in multi-CPU tests.
- **Performance Monitoring Tools**: The current `getpinfo` and `getcontextswitches` system calls provide limited insights. Developing a comprehensive kernel profiling tool, accessible via a new `getstats` system call, to report per-process and per-CPU metrics (e.g., `run_ticks`, `wait_ticks`, `ticks_scheduled`, queue contention) would aid debugging and optimization. This requires extending `proc.h` to store additional statistics and implementing `sys_getstats` in `sysproc.c`, improving analysis for tests like CPU-heavy and Mixed Load (Tables 2, 8).
- **Optimized Synchronization**: Both schedulers use spinlocks for runqueue access, contributing to contention at high CPU counts (e.g., priority CS: 95–131, lottery fairness degradation at 5 CPUs). Implementing lock-free or fine-grained locking mecha-

nisms (e.g., per-priority-queue locks for the priority scheduler, per-process ticket updates for the lottery scheduler) would reduce contention. This involves redesigning `runqueue.c` to use atomic operations or read-write locks, leveraging `xv6`'s spinlocks, and would enhance scalability for all tests.

These general improvements would make both schedulers more scalable and adaptive, addressing common challenges in multi-CPU environments and dynamic workloads.

8.4 Summary

The proposed improvements address specific weaknesses in the priority scheduler (slow aging, inefficiency in uniform priorities) and lottery scheduler (runqueue fragmentation, short-lived process allocation), while introducing general enhancements (hybrid scheduling, advanced load balancing) to strengthen both. By implementing dynamic aging intervals, workload-based priority assignment, and adaptive ticket adjustments, the schedulers will better handle diverse workloads. Advanced load balancing and optimized synchronization will improve scalability, while performance monitoring tools will facilitate future optimization. These improvements align with the thesis's goals, enhancing responsiveness, fairness, and scalability within `xv6`'s scheduling framework.

9 CONCLUSIONS

This thesis presented the design, implementation, and evaluation of two scheduling algorithms—priority scheduling and lottery scheduling—in the `xv6` operating system, replacing the default round-robin scheduler to explore deterministic and probabilistic approaches to CPU resource allocation. Guided by the goals of developing robust schedulers, evaluating their performance across diverse workloads, and proposing improvements, this work provides a comprehensive analysis of their behavior, trade-offs, and potential enhancements, contributing to a deeper understanding of operating system scheduling.

The priority scheduler, implemented with per-CPU runqueues and 11 priority levels (Section 3), achieves high responsiveness for critical tasks, as demonstrated in the I/O-bound test (completion times of 27–44 ticks vs. 51–66 ticks for round-robin, Table 2) and the Starvation Check test (44–33 ticks vs. 46–14 ticks). Its deterministic selection via `rq_select` and the aging mechanism in `update_priorities` ensure predictable execution and mitigate starvation, making it ideal for real-time or I/O-intensive systems. However, its performance in uniform-priority workloads (e.g., CPU-heavy: 84–28 ticks, Table 2) resembles round-robin, and the fixed 50-tick aging interval delays low-priority processes in dynamic workloads, indicating limitations in fairness.

The lottery scheduler, built with ticket-based randomization and per-CPU runqueues (Section 3), excels in long-term fairness, as shown in the Switch Overhead test (proportions of 50.1–50.9%, 31.7–33.2%, 17.3–18.0% vs. expected 50%, 33%, 16%, Table 5). Its probabilistic approach, supported by the `settickets` and `getpinfo` system calls, ensures proportional CPU allocation for large process counts, suitable for general-purpose systems. However, runqueue fragmentation degrades fairness in small multi-CPU tests (e.g., Low Process Count: 50.4–43.8% for process A, Table 3), and short-lived processes receive negligible time in mixed workloads (0.3% vs. 20%, Table 8), highlighting challenges in responsiveness and scalability.

The comparison of the two schedulers (Section 7) reveals distinct trade-offs. The priority scheduler emphasizes responsiveness and predictability, outperforming the lottery scheduler in I/O-bound and prioritized workloads but sacrificing fairness for low-priority tasks. The lottery scheduler ensures equitable resource allocation, surpassing the priority scheduler in fairness for diverse workloads, but it lacks deterministic prioritization and faces fairness issues in multi-CPU settings due to static load balancing. Both schedulers scale well with per-CPU runqueues, but their performance depends on the workload, with the priority scheduler being more predictable and the lottery scheduler sensitive to process distribution and CPU count.

The proposed improvements (Section 8) address these limitations. For the priority scheduler, dynamic aging intervals, workload-based priority assignment, and priority-weighted load balancing will enhance fairness and scalability. For the lottery scheduler, dynamic load balancing, enhanced starvation prevention, and short-lived process handling will improve fairness and responsiveness. General improvements, such as a hybrid scheduling approach combining priority and lottery mechanisms, advanced load balancing, and performance monitoring tools, will make both schedulers more robust and adaptive, aligning with the thesis's goal of proposing actionable enhancements.

This work contributes to operating systems research by demonstrating the practical implementation of scheduling algorithms in a real system, `xv6`, and providing a detailed performance evaluation across 1 to 5 CPUs (Section 5). The customized test suites

(`prioritytest.c`, `lotterytest.c`) offer a rigorous methodology for assessing responsiveness, fairness, and scalability, while system call extensions (`setpriority`, `settickets`, `getpinfo`) enhance `xv6`'s flexibility for scheduling experiments. The findings underscore the importance of tailoring scheduler design to specific workloads and the challenges of balancing responsiveness, fairness, and scalability in multi-CPU environments.

Future work could implement the proposed improvements and explore additional scheduling paradigms, such as multilevel feedback queues or real-time scheduling, in `xv6`. Expanding test suites to include more complex workloads (e.g., network-bound tasks) and evaluating energy efficiency would further enrich the analysis. This thesis not only achieves its objectives but also lays the foundation for further exploration of scheduling algorithms, offering insights into their practical implications and reinforcing the value of hands-on operating system development for advancing theoretical knowledge.

GLOSSARY

Term	Definition
Context Switch	The process of storing and restoring the state of a process so that execution can be resumed from the same point later.
CPU	Central Processing Unit, the primary component of a computer that performs computations.
CPU-heavy	A process that requires significant computational resources.
Fork	A system call that creates a new process by duplicating the calling process.
I/O-bound	A process that spends most of its time waiting for input/output operations.
Lottery	A scheduling algorithm that allocates CPU time probabilistically based on ticket assignments.
Priority	A scheduling algorithm that executes processes based on assigned priority levels.
Process	An instance of a program in execution.
Round-Robin	A scheduling algorithm that allocates equal time slices to processes in a cyclic order.
Runqueue	A queue of runnable processes waiting for CPU time.
Runnable	A process state indicating it is ready to execute but waiting for CPU allocation.
Scheduling	The method by which CPU time is allocated to processes.
Spinlock	A synchronization mechanism that causes a thread to wait in a loop until a resource is available.
Starvation	A condition where a process is perpetually denied CPU time.
Ticket	A value assigned to a process in lottery scheduling to determine its CPU allocation probability.
Turnaround Time	The total time taken from process submission to completion.
Yield	A system call that allows a process to voluntarily relinquish CPU control.
xv6	A Unix-like teaching operating system developed by MIT.

ABBREVIATIONS

Abbreviation	Expansion
CS	Context Switches
CPU	Central Processing Unit
I/O	Input/Output
MIT	Massachusetts Institute of Technology
NCPU	Number of Central Processing Units
PID	Process Identifier
xv6	Experimental Version 6

APPENDIX I

Execution Instructions

To verify the experiments:

1. Clone the repository: `git clone https://github.com/Orestouio/xv6-thesis-enhancements`
2. Install QEMU: `sudo apt-get install qemu-system-x86`
3. Navigate to one of the directories: `lottery-scheduler`, `priority-scheduler`, or `round-robin`
4. Compile xv6: `make`
5. Run with 1 to 5 CPUs: `make qemu CPUS=x`, where `x` is 1, 2, 3, 4, or 5
6. Execute the corresponding test: `./prioritytest`, `./lotterytest`, or `./timingtests`

Source Code

The priority and lottery schedulers were implemented in the files `runqueue.c` and `proc.c`. Key functions include `rq_select` for process selection and `scheduler` for execution management. The full source code and test suites (`prioritytest.c`, `lotterytest.c`, `timingtests.c`) are available at https://github.com/Orestouio/xv6-thesis-enhancements [1].

Note on Results

The tables in Section 5 present rounded averages from five test iterations. The test suites, available in the repository [1], can be executed to reproduce the results.

References

[1] “Source code repository for priority and lottery scheduling in xv6,” [Online]. Available: https://github.com/Orestouio/xv6-thesis-enhancements. [Accessed: Jun. 20, 2025].

[2] C. A. Waldspurger and W. E. Weihl, “Lottery scheduling: Flexible proportional-share resource management,” in Proc. 1st USENIX Symp. Operating Systems Design and Implementation (OSDI), Monterey, CA, USA, Nov. 1994, pp. 1–12.

[3] Massachusetts Institute of Technology PDOS Group, “xv6 operating system source code (Revision 2020),” [Online]. Available: https://pdos.csail.mit.edu/6.828/xv6. [Accessed: Jun. 20, 2025].

[4] A. Silberschatz, P. B. Galvin, and G. Gagne, Operating System Concepts, 10th ed. Hoboken, NJ, USA: Wiley, 2018.

[5] E. G. Coffman, Jr., and L. Kleinrock, “Computer scheduling methods and their countermeasures,” in Proc. Spring Joint Computer Conf., Atlantic City, NJ, USA, Apr. 1968, pp. 155–162.

[6] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quema, and A. Fedorova, “The Linux scheduler: A decade of wasted cores,” in Proc. 11th European Conf. Computer Systems (EuroSys), London, UK, Apr. 2016, pp. 1–16.

[7] N. Fisher, S. Baruah, and T. Fisher, “Partitioned Scheduling of Sporadic Tasks According to Static Priorities,” in Proc. 18th Euromicro Conf. Real-Time Systems (ECRTS), Dresden, Germany, Jul. 2006, pp. 123–130.

[8] QEMU Project, “QEMU documentation (latest version),” [Online]. Available: https://www.qemu.org/docs/master/. [Accessed: Jun. 20, 2025].