Enhancing xv6: Priority Scheduler, Lottery Scheduler, and Shared Memory with Semaphores

Orestis Theodorou

May 6, 2025

Abstract

This thesis presents the enhancement of the xv6 operating system through the implementation of three distinct mechanisms: a priority-based scheduler, a lottery scheduler, and shared memory with semaphores. The priority scheduler assigns processes priority levels (0–10) to ensure critical tasks are executed promptly, achieving significant runtime improvements (e.g., 12 ticks vs. 43.5 ticks in starvation tests) compared to the default round-robin scheduler. The lottery scheduler introduces probabilistic scheduling based on ticket counts, achieving proportional fairness with deviations within 2–3% of expected values across diverse workloads. The shared memory and semaphore mechanisms enable efficient inter-process communication (IPC) and synchronization, validated through a producer-consumer application under standard and stress conditions. These implementations are rigorously tested in a single-core environment, demonstrating their effectiveness in improving scheduling flexibility and IPC capabilities in xv6. This work provides a comprehensive foundation for educational purposes and future operating system enhancements.

1 Introduction

The xv6 operating system, a pedagogical platform derived from UNIX, provides a lightweight environment for exploring operating system concepts. Originally equipped with a round-robin scheduler and lacking advanced IPC mechanisms, xv6 offers an ideal foundation for implementing and evaluating new scheduling and communication primitives. This thesis enhances xv6 through three implementations:

- **Priority Scheduler**: Replaces the round-robin scheduler with a priority-based mechanism, assigning processes priority levels (0–10, with 0 being the highest) to ensure critical tasks are executed efficiently.

- **Lottery Scheduler**: Introduces a probabilistic scheduling approach where processes are assigned tickets, and scheduling probability is proportional to ticket counts, ensuring fairness across workloads.
- **Shared Memory and Semaphores**: Implements shared memory for efficient IPC and semaphores for synchronization, enabling coordinated access to shared resources in a producer-consumer application.

Each implementation is designed and tested in a single-core configuration (`CPUS := 1`) to ensure direct comparisons with the baseline round-robin scheduler and to isolate the effects of the new mechanisms. This report details the design, implementation, testing, and performance analysis of each component, followed by a discussion of their collective impact on xv6 and potential future enhancements.

2 Background and Related Work

The xv6 operating system, developed by MIT [1], is a teaching-oriented system that mirrors UNIX v6, providing a simple yet functional platform for studying operating system concepts. Its default round-robin scheduler ensures fairness by allocating equal time slices to all processes, but it lacks mechanisms for prioritizing tasks or allocating CPU time proportionally.

Scheduling algorithms have been extensively studied in operating system design. Priority scheduling, as discussed in [2], assigns processes priority levels to ensure critical tasks are executed promptly, though it risks starvation without mitigation strategies like aging. Lottery scheduling, proposed by Waldspurger and Weihl [3], introduces a probabilistic approach where processes are assigned tickets, and the scheduler selects processes based on ticket proportions, offering fairness and flexibility.

Inter-process communication (IPC) and synchronization are fundamental to operating systems. Shared memory, as described in [4], allows processes to share data efficiently by mapping a common memory region into their address spaces. Semaphores, introduced by Dijkstra [5], provide a synchronization primitive to coordinate access to shared resources, preventing race conditions in concurrent environments.

This work builds on these concepts, adapting priority and lottery scheduling to xv6's single-core environment and introducing shared memory and semaphores to enable efficient IPC and synchronization, addressing gaps in xv6's original design.

3 Priority Scheduler Implementation

3.1 Design Overview

The priority scheduler replaces xv6's round-robin scheduler, introducing a mechanism where processes are scheduled based on priority levels (0–10, with 0 being the highest). The design includes:

- **Priority Levels**: Processes are assigned priorities from 0 (highest) to 10 (lowest), with a default priority of 5.
- **Priority Queue**: A scheduling queue organizes processes by priority, ensuring high-priority processes are executed first.
- **Dynamic Adjustment**: An aging mechanism adjusts priorities to prevent starvation, increasing priority (lowering the priority number) for waiting processes.
- **Fairness for I/O-Bound Processes**: A short-lived FIFO queue at priority 5 ensures fairness for I/O-bound processes.
- **Preemption**: Timer interrupts trigger preemption if a higher-priority process is runnable.

3.2 Implementation Details

The priority scheduler was implemented under the single-core configuration (`CPUS := 1`) to ensure a direct comparison with the round-robin scheduler. The following modifications were made:

- `proc.h`: Extended `struct proc` with fields: `int priority`, `int wait_ticks`, `int creation_time`, `int completion_time`, `int waiting_time`, `int last_runnable_tick`, `int first_run_time`, `int has_run`, `int cpu_time`, and `struct proc *next` for priority queues. In `allocproc`, new processes are initialized with priority 5.
- `proc.c`:
  - **Priority Queue and Scheduling**: The `scheduler()` function uses a priority queue (array of linked lists for priorities 0–10). Processes are scheduled from the highest-priority queue, with a FIFO queue at priority 5 for I/O-bound processes (PIDs ¿ 100).
  - **Dynamic Adjustment**: The `update_priorities` function forces processes with PIDs ¿ 100 to priority 5 and uses aging to reduce priority by 1

every 50 ticks of waiting, preventing starvation. Processes (except PID 1) are terminated after 10,000 ticks.

- **Context Switch Tracking**: A global `context_switches` counter is incremented in `scheduler()` during `swtch`.
- **Scheduling Log**: A `sched_log` function logs tick, PID, priority, and context switch count.

- `sysproc.c`: Added `sys_setpriority` and `sys_getcontextswitches` system calls, ensuring thread safety with `ptable.lock`.
- `trap.c`: Modified `trap()` to enable preemption on timer interrupts, yielding if a higher-priority process is runnable, with optimization via caching the minimum runnable priority.
- `timingtests.c`: Enhanced to measure runtime with `uptime()` and context switches. Test 2 performs 200 switches, Test 4 uses pipes for accurate timing, and a `sleep(5)` delay ensures test independence.

3.3 Testing and Validation

A baseline was established using xv6's round-robin scheduler under `CPUS := 1`, augmented with a `context_switches` counter and `sys_getcontextswitches` system call. The `timingtests.c` utility evaluated performance across seven workloads, each run 10 times over two runs (1 tick = 10ms):

1. **CPU-intensive**: 10 processes, 20M iterations each, concurrent.
2. **Context-switching overhead**: 200 sequential fork-and-exit operations.
3. **I/O-bound**: 100 processes, 100ms sleep each, 50 concurrent.
4. **Mixed workload**: 5 CPU-intensive (50M iterations, priority 0) and 5 I/O-bound (500ms sleep, priority 10), concurrent.
5. **Process creation**: 50 sequential fork operations, children exit immediately.
6. **Short-duration tasks**: 200 processes, 10,000 iterations each, 50 concurrent.
7. **Starvation evaluation**: 1 lightweight process (50K iterations, priority 0) vs. 5 heavyweight processes (20M iterations each, priority 5), concurrent.

Round-Robin Results:

```
Test 1: CPU-heavy: Avg 87 ticks (Range: 83-100), Avg Context Switches:
    101.95
```

```
Test 2: Switch overhead: Avg 70.5 ticks (Range: 67-83)
Test 3: I/O-bound: Avg 52.5 ticks (Range: 50-61)
Test 4: Mixed load: Avg 50 ticks (Range: 50-52)
Test 5: Process creation: Avg 17 ticks (Range: 17-18)
Test 6: Short tasks: Avg 70 ticks (Range: 68-73)
Test 7: Starvation check: Avg 43.5 ticks (Range: 42-46)
```

Priority Scheduler Results:

```
Test 1: CPU-heavy: Avg 14 ticks (Range: 13-16), Avg Context Switches: 19-21
Test 2: Switch overhead: Avg 68 ticks (Range: 68-69)
Test 3: I/O-bound: Avg 25 ticks (Range: 25-27), Avg Context Switches:
   557-558
Test 4: Mixed load: Avg 50 ticks (Range: 50-50)
Test 5: Process creation: Avg 17 ticks (Range: 17-20)
Test 6: Short tasks: Avg 70 ticks (Range: 69-71)
Test 7: Starvation check: Avg 12 ticks (Range: 11-13)
```

3.4 Final Results and Comparison

- **Test 1 (CPU-heavy)**: The priority scheduler achieves 14 ticks (range: 13–16 ticks, 19–21 context switches) vs. round-robin's 87 ticks (range: 83–100 ticks, 101.95 context switches). The priority queue minimizes overhead, and preemption ensures fair CPU sharing at priority 5.
- **Test 2 (Switch overhead)**: The priority scheduler averages 68 ticks (range: 68–69 ticks) vs. round-robin's 70.5 ticks (range: 67–83 ticks), with better consistency due to the FIFO queue at priority 5 and test isolation via `sleep(5)`.
- **Test 3 (I/O-bound)**: The priority scheduler averages 25 ticks (range: 25–27 ticks, 557–558 context switches) vs. round-robin's 52.5 ticks (range: 50–61 ticks), prioritizing half the processes at priority 0. High context switches indicate excessive preemption by PID 3.
- **Test 4 (Mixed load)**: Both schedulers average 50 ticks, but the priority scheduler prioritizes CPU-bound processes (priority 0) while aging prevents starvation of I/O-bound processes (priority 10).
- **Test 5 (Process creation)**: Both schedulers average 17 ticks, with the priority scheduler showing slight variability (range: 17–20 ticks) due to queue overhead.
- **Test 6 (Short tasks)**: Both schedulers average 70 ticks, with the priority scheduler showing tighter consistency (range: 69–71 ticks) due to test isolation.

- **Test 7 (Starvation check)**: The priority scheduler averages 12 ticks (range: 11–13 ticks) vs. round-robin's 43.5 ticks (range: 42–46 ticks), prioritizing the lightweight process (priority 0) effectively.

The priority scheduler outperforms round-robin in Tests 1, 2, 3, and 7, matches it in Tests 4, 5, and 6, and demonstrates effective prioritization with fairness via aging. High context switches in Test 3 are a noted limitation.

4 Lottery Scheduler Implementation

4.1 Design Overview

The lottery scheduler introduces probabilistic scheduling where processes are assigned tickets, and the probability of scheduling is proportional to ticket counts (e.g., tickets 30, 20, 10 yield expected proportions of 50%, 33%, 16.67%). The design includes:

- **Ticket-Based Scheduling**: Processes are selected randomly based on ticket proportions.
- **Random Number Generation**: An Xorshift-based generator ensures high-quality randomness.
- **Shuffling**: Fisher-Yates shuffling reduces ordering bias in process selection.
- **Scheduling Events Tracking**: A counter tracks scheduling events for performance analysis.

4.2 Implementation Details

The lottery scheduler was implemented under `CPUS := 1` to compare with the round-robin scheduler. Key modifications include:

- `proc.c`:
  - **Lottery Scheduling**: The `scheduler()` collects runnable processes into `runnable_procs`, computes total tickets, selects a winner via random number, and shuffles the array using Fisher-Yates to reduce bias.
  - **Random Number Generator**: Implements `srand`, `rand`, and `rand_range` using Xorshift, seeded with `ticks`, CPU ID, and other variables to ensure fairness.
  - **Scheduling Log**: The `ticks_scheduled` field tracks scheduling events.

  - **Process Management**: `fork()`, `allocproc()`, and `userinit()` initialize `tickets` (default 1) and `ticks_scheduled`.

- `lotterytest.c`:
  - **Test Functions**: `run_workload_test` (Tests 1, 3) forks 3 processes with specified tickets, performing iterations and yielding periodically. `run_switch_test` (Test 2) forks 50 processes in C-A-B order.
  - **Average Calculation**: `print_average_results` computes average proportions over 5 runs.
  - **Test Independence**: A `sleep(5)` delay ensures system stabilization between runs.
- **Optimizations**: Increased iterations (e.g., 500M in Test 1), reduced yield intervals (5,000 iterations), and added shuffling to reduce variability (e.g., Process A in Test 1: 45% to 55%).

4.3 Testing and Validation

The round-robin baseline was established under `CPUS := 1`, with `ticks_scheduled` added to track scheduling events. The `lotterytest.c` utility evaluated three workloads over 5 runs (1 tick = 10ms):

1. **CPU-intensive**: 3 processes, 500M iterations each, tickets 30, 20, 10 (expected: 50%, 33%, 16.67%).
2. **Context-switching overhead**: 50 processes (17 with 30 tickets, 17 with 20 tickets, 16 with 10 tickets, expected: 50%, 33%, 16.67%), 100M iterations each, forked in C-A-B order.
3. **Starvation check**: 3 processes, 10M iterations each, tickets 50, 10, 1 (expected: 81.97%, 16.39%, 1.64%).

Round-Robin Results:

```
Test 1: CPU-heavy: Avg 87 ticks (Range: 83-100), Schedules: A=33%, B=33%, C=33%
Test 2: Switch overhead: Avg 70.5 ticks (Range: 67-83), Schedules: A=33%, B=33%, C=33%
```

Lottery Scheduler Results:

```
Test 1: CPU-heavy: Avg 1810.6 ticks (Range: 1787-1851), Schedules: A=51.1%, B=31.8%, C=16.9%
```

```
Test 2: Switch overhead: Avg not measured, Schedules: A=50.5%, B=32.2%, C
    =17.2%
Test 3: Starvation check: Avg 42 ticks (Range: 42-42), Schedules: A=79.3%,
    B=18.6%, C=1.9%
```

4.4 Final Results and Comparison

- **Test 1 (CPU-heavy)**: The lottery scheduler achieves proportions of A: 51.1%, B: 31.8%, C: 16.9% (expected: 50%, 33%, 16.67%) with 1810.6 ticks (range: 1787–1851 ticks), vs. round-robin's equal proportions (33% each) and 87 ticks. Deviations are within 2–3%, with 753 scheduling events per run reducing variability.
- **Test 2 (Switch overhead)**: Proportions are A: 50.5%, B: 32.2%, C: 17.2% (expected: 50%, 33%, 16.67%) vs. round-robin's 33% each. Deviations are within 2–3%, with 13211 events per run ensuring low variability.
- **Test 3 (Starvation check)**: Proportions are A: 79.3%, B: 18.6%, C: 1.9% (expected: 81.97%, 16.39%, 1.64%) with 42 ticks, vs. round-robin's 33% each and 87 ticks. Process C (1 ticket) is scheduled consistently, preventing starvation.

The lottery scheduler achieves proportional fairness, outperforming round-robin in ticket-based scheduling, though runtime overhead (e.g., Test 1: 1810.6 ticks) is a noted area for optimization.

5 Shared Memory and Semaphore Implementation

5.1 Shared Memory Implementation

5.1.1 Design Overview

The shared memory system enables processes to create, access, and manage shared memory regions identified by unique string identifiers, with features including named regions, multiple mappings, inheritance across `fork`, and reference counting for automatic deallocation.

- **Data Structures**:
  - `shmtable`: A global array of `NSHM` (10) `shm` structures (`name`, `in_use`, `phys_addr`, `size`, `ref_count`, `lock`).

  - Per-process mappings in `proc`: `shm_mappings[MAX_SHM_MAPPINGS]` (4), `shm_objects`, `shm_count`.

- **System Calls**: `shm_open(const char *name, int size)` and `shm_close(int addr)`.

5.1.2 Implementation Details

Modifications include:

- `proc.h`, `proc.c`: Define `shm` structure, implement `sys_shm_open` and `sys_shm_close`, and modify `fork` and `exit`.
- `syscall.h`, `syscall.c`, `usys.S`, `user.h`: Add system call support.

`sys_shm_open` allocates memory, maps it at `0x60000000` (incremented by `PGSIZE`), and manages references. `sys_shm_close` unmaps regions and deallocates memory when `ref_count` reaches zero.

5.1.3 Testing and Validation

The `shmtest.c` program validated functionality:

- **Basic Test**: Parent and child share `/shm1`, with the child updating the value to 42.
- **Extended Tests**: Multiple regions (`/shm1`, `/shm2`), maximum mappings (4), reusing names (`/shm_reuse`), and invalid inputs.

Output:

```
Test 1: Opening two shared memory regions
Parent: Set /shm1 to 100, /shm2 to 200
Child: /shm1 = 100, /shm2 = 200
Child: Set /shm1 to 101, /shm2 to 201
Parent: /shm1 = 101, /shm2 = 201
[Kernel Debug] Shared memory /shm1 freed
[Kernel Debug] Shared memory /shm2 freed

Test 2: Maximum shared memory mappings
Opened /shm_max0 at address 0x60000000
Opened /shm_max1 at address 0x60001000
Opened /shm_max2 at address 0x60002000
Opened /shm_max3 at address 0x60003000
shm_open failed for /shm_max4 (expected for i=4)
```

```
[Kernel Debug] Shared memory /shm_max0 freed
[Kernel Debug] Shared memory /shm_max1 freed
[Kernel Debug] Shared memory /shm_max2 freed
[Kernel Debug] Shared memory /shm_max3 freed

Test 3: Reusing shared memory names
Child: /shm_reuse = 300
Child: Set /shm_reuse to 301
Parent: /shm_reuse = 301
[Kernel Debug] Shared memory /shm_reuse freed

Test 4: Invalid inputs
shm_open with negative size failed (expected)
shm_open with zero size failed (expected)
shm_open with size > PGSIZE failed (expected)
shm_close with invalid address failed (expected)

All tests completed
```

5.1.4 Challenges and Solutions

Challenges included implicit declarations (`mappages`), undefined functions (`strcmp`), user-space errors, multiple definitions, page faults in `fork`, and missing `snprintf`, all resolved through appropriate kernel modifications and manual string handling.

5.2 Semaphore Implementation

5.2.1 Design Overview

The semaphore system provides synchronization with counting semaphores, blocking/waking, inheritance across `fork`, automatic cleanup, and error handling.

- **Data Structures**:
  - `semtable`: Array of `NSEM` (10) `sem` structures (`in_use`, `value`, `lock`, `queue` with `NPROC` (64) entries).
  - Per-process in `proc`: `sem_ids[MAX_SEM]` (4), `sem_count`.
- **System Calls**: `sem_init(int value)`, `sem_wait(int sem_id)`, `sem_post(int sem_id)`.

5.2.2 Implementation Details

Modifications include:

- `proc.h`, `proc.c`: Define `sem` structure, implement `sys_sem_init`, `sys_sem_wait`, `sys_sem_post`, and modify `fork` and `exit`.
- `syscall.h`, `syscall.c`, `usys.S`, `user.h`: Add system call support.

`sys_sem_init` validates inputs, allocates a semaphore, and initializes the queue. `sys_sem_wait` decrements `value`, blocking if negative, and `sys_sem_post` increments `value`, waking a process if waiting.

5.2.3 Testing and Validation

The `prodcons.c` program tested semaphores using a circular buffer with semaphores `empty`, `full`, and `print_sem`:

- **Standard Test (Buffer Size 5, 10 Items):**

```
Producer: produced 0 at index 0
Consumer: consumed 0 from index 0
Producer: produced 1 at index 1
Consumer: consumed 1 from index 1
Producer: produced 2 at index 2
Consumer: consumed 2 from index 2
Producer: produced 3 at index 3
Consumer: consumed 3 from index 3
Producer: produced 4 at index 4
Consumer: consumed 4 from index 4
Producer: produced 5 at index 0
Consumer: consumed 5 from index 0
Producer: produced 6 at index 1
Consumer: consumed 6 from index 1
Producer: produced 7 at index 2
Consumer: consumed 7 from index 2
Producer: produced 8 at index 3
Consumer: consumed 8 from index 3
Producer: produced 9 at index 4
Consumer: consumed 9 from index 4
[Kernel Debug] Semaphore 0 freed
[Kernel Debug] Semaphore 1 freed
[Kernel Debug] Semaphore 2 freed
[Kernel Debug] Shared memory /buffer freed
```

- **Stress Test (Buffer Size 2, 20 Items):**

```
Producer: produced 0 at index 0
Consumer: consumed 0 from index 0
Producer: produced 1 at index 1
Consumer: consumed 1 from index 1
Producer: produced 2 at index 0
Consumer: consumed 2 from index 0
[...]
Producer: produced 19 at index 1
Consumer: consumed 19 from index 1
[Kernel Debug] Semaphore 0 freed
[Kernel Debug] Semaphore 1 freed
[Kernel Debug] Semaphore 2 freed
[Kernel Debug] Shared memory /buffer freed
```

- **Error Handling Test (Invalid Semaphore ID):**

```
prodcons: invalid semaphore ID -1, exiting
```

The tests confirm synchronized access, robustness under contention, and proper error handling.

5.2.4 Challenges and Solutions

Challenges included linker errors, kernel panics due to lock conflicts, interleaved output, and unbalanced execution, resolved through implementation fixes, lock restructuring, adding `print_sem`, and introducing a producer delay.

6 Discussion

The three implementations collectively enhance xv6's capabilities:

- **Priority Scheduler**: Excels in prioritizing critical tasks (e.g., Test 7: 12 ticks vs. 43.5 ticks), but high context switches in Test 3 (557–558) suggest preemption overhead.
- **Lottery Scheduler**: Achieves proportional fairness (e.g., Test 1 deviations within 2–3%), though runtime overhead (1810.6 ticks in Test 1) indicates optimization potential.
- **Shared Memory and Semaphores**: Enables efficient IPC and synchronization, with robust performance under stress (e.g., buffer size 2, 20 items).

Challenges across implementations included kernel-level modifications (e.g., system call integration, lock management) and test design (e.g., ensuring sufficient scheduling events, test independence). The combined impact of these enhancements makes xv6 a more versatile platform for teaching and experimenting with operating system concepts, supporting advanced scheduling and IPC scenarios.

7 Conclusion

This thesis successfully enhances the xv6 operating system through three implementations: a priority scheduler, a lottery scheduler, and shared memory with semaphores. The priority scheduler ensures efficient execution of critical tasks, outperforming the round-robin scheduler in key scenarios (e.g., 12 ticks vs. 43.5 ticks in starvation tests). The lottery scheduler achieves proportional fairness, with scheduling proportions within 2–3% of expected values, offering a flexible alternative to round-robin scheduling. The shared memory and semaphore mechanisms enable robust IPC and synchronization, validated through a producer-consumer application under diverse conditions.

These enhancements collectively improve xv6's functionality, making it a more comprehensive educational tool. Future work includes optimizing the priority scheduler's context switches, reducing the lottery scheduler's runtime overhead, and extending the shared memory system with advanced synchronization primitives like condition variables. Additionally, evaluating all implementations in a multi-core environment (`CPUS := 2`) would provide insights into their scalability and performance under parallelism.

References

[1] Cox, R., Kaashoek, F., & Morris, R. (2019). Xv6: A Simple, Unix-like Teaching Operating System. *MIT CSAIL*.

[2] Tanenbaum, A. S., & Bos, H. (2008). *Modern Operating Systems*. Pearson Education.

[3] Waldspurger, C. A., & Weihl, W. E. (1994). Lottery Scheduling: Flexible Proportional-Share Resource Management. *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[4] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts*. Wiley.

[5] Dijkstra, E. W. (1965). Cooperating Sequential Processes. *Programming Languages*, Academic Press.