Enhancing xv6: Priority and Lottery Scheduling with Shared Memory and Semaphores

Orestis Theodorou

May 6, 2025

Abstract

This thesis enhances the xv6 operating system by implementing three key features: a priority-based scheduler, a lottery scheduler, and shared memory with semaphores. The priority scheduler assigns processes priority levels (0–10), achieving significant performance improvements over the default round-robin scheduler, such as a 72% runtime reduction in starvation tests (12 ticks vs. 43.5 ticks). The lottery scheduler introduces probabilistic scheduling based on ticket counts, achieving proportional fairness with deviations within 2–3% of expected proportions (e.g., 51.1%, 31.8%, 16.9% for tickets 30, 20, 10). The shared memory and semaphore mechanisms enable efficient inter-process communication (IPC) and synchronization, validated through a producer-consumer application under diverse conditions. These enhancements, evaluated in a single-core environment (`CPUS := 1`), improve xv6's functionality, making it a more comprehensive platform for operating systems education. Performance comparisons with the round-robin baseline, detailed test results, and future optimization opportunities are presented, highlighting the practical and educational value of this work.

Contents

1 Introduction

The xv6 operating system, a pedagogical platform derived from UNIX v6, serves as an ideal environment for exploring operating system concepts due to its simplicity and clarity [1]. However, its default design—featuring a round-robin scheduler and lacking advanced inter-process communication (IPC) mechanisms—limits its ability to support complex scheduling and synchronization scenarios. This thesis addresses these limitations by implementing three enhancements: a priority-based scheduler, a lottery scheduler, and shared memory with semaphores. These additions transform xv6 into a more versatile teaching tool, enabling students to explore advanced scheduling techniques and IPC mechanisms through practical implementation and evaluation.

The priority scheduler replaces the default round-robin scheduler, allowing processes to be scheduled based on priority levels (0–10, with 0 being the highest), incorporating mechanisms like aging and preemption to balance prioritization and fairness. The lottery scheduler introduces a probabilistic approach, where processes are assigned tickets, and scheduling probability is proportional to ticket counts (e.g., tickets 30, 20, 10 yield expected proportions of 50%, 33%, 16.67%). The shared memory and semaphore mechanisms enable efficient IPC and synchronization, supporting concurrent programming scenarios like the producer-consumer problem.

Each implementation is evaluated in a single-core environment (`CPUS := 1`) to ensure a direct comparison with the round-robin baseline, deferring multi-core analysis (`CPUS := 2`) to future work due to the additional complexity of parallelism. Performance is assessed using custom benchmarking utilities (`timingtests.c`, `lotterytest.c`, `shmtest.c`, `prodcons.c`), with results demonstrating significant improvements over the baseline. This thesis documents the design, implementation, and evaluation of these enhancements, highlighting their contributions to xv6 and operating systems education.

2 Background and Related Work

The xv6 operating system, developed by MIT [1], is designed for educational purposes, providing a lightweight platform to study 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 supporting proportional scheduling. Additionally, xv6's original design does not include advanced IPC mechanisms, limiting its ability to handle concurrent programming scenarios.

Priority scheduling, a well-established technique in operating systems, assigns pro-

cesses priority levels to ensure timely execution of critical tasks [2]. However, it risks starvation for low-priority processes without mitigation strategies like aging. Lottery scheduling, proposed by Waldspurger and Weihl [3], offers a probabilistic approach, allocating CPU time proportionally based on ticket counts, providing flexibility and fairness. Shared memory and semaphores are fundamental for IPC and synchronization [4]. Shared memory allows processes to share data efficiently, while semaphores, introduced by Dijkstra [5], coordinate access to shared resources, preventing race conditions in concurrent environments.

This thesis builds on these concepts, implementing a priority scheduler with aging, a lottery scheduler with optimizations for fairness, and shared memory with semaphores in xv6. These enhancements address gaps in xv6's original design, enabling students to explore advanced scheduling and IPC concepts through practical implementation and evaluation.

3 Priority Scheduler Implementation

3.1 Implementation 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 implementation was designed and tested under a single-core configuration (`CPUS := 1`) to ensure a direct comparison with the round-robin baseline. Key modifications include:

- `proc.h`: Extended the `proc` structure with fields like `priority`, `wait_ticks`, `cpu_time`, and others to support priority scheduling and performance tracking.
- `proc.c`: Modified `scheduler()` to use a priority queue (array of linked lists for each priority level), implemented `update_priorities` for aging (reduces priority by 1 every 50 ticks), and added a short-lived FIFO queue for I/O-bound processes at priority 5. Added `context_switches` counter and `sched_log` for performance analysis.
- `sysproc.c`: Implemented `sys_setpriority` and `sys_getcontextswitches` system calls for priority adjustment and performance monitoring.
- `trap.c`: Enabled preemption by modifying `trap()` to check for higher-priority runnable processes on timer interrupts, invoking `yield()` if necessary.
- `timingtests.c`: Developed a test suite with seven workloads (CPU-intensive, I/O-bound, mixed, etc.), measuring runtime and context switches, with a `sleep(5)`

delay between runs for test independence.

3.2 Performance Evaluation

The priority scheduler was evaluated using `timingtests.c` across seven workloads, compared against the round-robin baseline (`CPUS := 1`):

- **Round-Robin Baseline Output:**

```
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 Output:**

```
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)
```

- **Key Results:**
  - **Test 1 (CPU-heavy)**: Reduced runtime from 87 to 14 ticks, with fewer context switches (19–21 vs. 101.95), due to efficient priority queue scheduling.
  - **Test 3 (I/O-bound)**: Improved runtime from 52.5 to 25 ticks by prioritizing half the processes at priority 0, though high context switches (557–558) indicate a need for optimization.
  - **Test 7 (Starvation check)**: Achieved a 72% runtime reduction (12 vs. 43.5 ticks), demonstrating effective prioritization of a lightweight process (priority 0) over heavyweight ones (priority 5).

  - **Tests 4, 5, 6**: Matched the round-robin baseline, showing balanced handling of mixed and uniform-priority workloads.

3.3 Future Enhancements

- Reduce context switches in Test 3 (557–558 to 100–150) by enforcing a minimum time slice for short-lived processes.
- Evaluate performance in a multi-core environment (`CPUS := 2`) to assess scalability.

4 Lottery Scheduler Implementation

4.1 Implementation Overview

The lottery scheduler introduces probabilistic scheduling, where processes are assigned tickets, and scheduling probability is proportional to ticket counts. The implementation was tested under `CPUS := 1`. Key modifications include:

- `proc.c`: Modified `scheduler()` to collect runnable processes, compute total tickets, and select a winner randomly using an Xorshift-based generator. Added Fisher-Yates shuffling to reduce ordering bias. Extended `proc` structure with `tickets` and `ticks_scheduled`.
- `lotterytest.c`: Developed a test suite with three workloads (CPU-intensive, high-contention, starvation check), measuring scheduling proportions and runtime, with a `sleep(5)` delay between runs.
- **System Calls**: Added `settickets` to set ticket counts and another to retrieve scheduling counts.

4.2 Performance Evaluation

The lottery scheduler was evaluated using `lotterytest.c` across three workloads, compared against the round-robin baseline (`CPUS := 1`):

- **Round-Robin Baseline Output:**

```
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 Output:**

```
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%
```

- **Key Results:**
  - **Test 1 (CPU-heavy)**: Achieved proportions of 51.1%, 31.8%, 16.9% (expected: 50%, 33%, 16.67%), with deviations within 2–3%, compared to round-robin's equal 33% each. Higher runtime (1810.6 ticks) due to increased iterations (500M).
  - **Test 2 (Switch overhead)**: Proportions of 50.5%, 32.2%, 17.2% (expected: 50%, 33%, 16.67%), with low variability due to 13211 scheduling events per run.
  - **Test 3 (Starvation check)**: Proportions of 79.3%, 18.6%, 1.9% (expected: 81.97%, 16.39%, 1.64%), preventing starvation of the low-ticket process.

4.3 Future Enhancements

- Reduce variability in Test 1 (e.g., Process A: 45% to 55%) by increasing scheduling events or introducing hybrid time-slicing.
- Optimize runtime overhead in Test 1 (1810.6 ticks) by improving the random number generator.
- Evaluate multi-core performance (`CPUS := 2`).

5 Shared Memory and Semaphore Implementation

5.1 Implementation Overview

The shared memory and semaphore mechanisms enable efficient IPC and synchronization in xv6. Key modifications include:

- **Shared Memory**:

  - `proc.h`: Added `shm` structure for a global `shmtable` and extended `proc` with `shm_mappings`, `shm_objects`, and `shm_count`.
  - `proc.c`: Implemented `sys_shm_open` to create/access shared memory regions and `sys_shm_close` to unmap them. Modified `fork` and `exit` to handle inheritance and cleanup.

- **Semaphores**:
  - `proc.h`: Added `sem` structure for a global `semtable` and extended `proc` with `sem_ids` and `sem_count`.
  - `proc.c`: Implemented `sys_sem_init`, `sys_sem_wait`, and `sys_sem_post` for semaphore operations. Modified `fork` and `exit` for inheritance and cleanup.
- **Testing**: Developed `shmtest.c` to validate shared memory and `prodcons.c` for a producer-consumer application using semaphores.

5.2 Performance Evaluation

The shared memory system was validated using `shmtest.c`, confirming correct operation across scenarios like multiple regions, maximum mappings, and invalid inputs:

```
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

Test 2: Maximum shared memory mappings
Opened /shm_max0 at address 0x60000000
[...]

Test 3: Reusing shared memory names
Child: /shm_reuse = 300
[...]
```

The semaphore implementation was tested using `prodcons.c` under standard (buffer size 5, 10 items) and stress (buffer size 2, 20 items) conditions:

```
Producer: produced 0 at index 0
Consumer: consumed 0 from index 0
[...]
```

```
Producer: produced 19 at index 1
Consumer: consumed 19 from index 1
```

The tests confirmed correct synchronization, with no deadlocks or race conditions, even under high contention.

5.3 Future Enhancements

- Stress test with multiple producer-consumer pairs.
- Implement `sem_destroy` for explicit semaphore cleanup.
- Add advanced synchronization primitives like condition variables.

6 Discussion

The priority scheduler excels in scenarios requiring strict prioritization (e.g., Test 7: 12 vs. 43.5 ticks), but its high context switches in I/O-bound workloads (Test 3: 557–558) indicate a need for optimization. The lottery scheduler achieves proportional fairness across diverse workloads, with deviations within 2–3%, though its runtime overhead (Test 1: 1810.6 ticks) suggests opportunities for efficiency improvements. The shared memory and semaphore mechanisms enable robust IPC and synchronization, handling both standard and stress conditions effectively, as seen in the producer-consumer application.

Comparing the schedulers, the priority scheduler is ideal for workloads with clear priority hierarchies, while the lottery scheduler offers flexibility for proportional scheduling. The shared memory and semaphore implementations complement both by enabling concurrent programming, a critical aspect of modern operating systems. Together, these enhancements make xv6 a more comprehensive platform for studying operating system concepts.

7 Contributions

This thesis makes the following contributions to the xv6 operating system and the field of operating systems education:

- **Priority Scheduler**: Introduces a priority-based scheduling mechanism to xv6, enabling efficient execution of critical tasks. The implementation includes dynamic priority adjustments via aging and preemption, achieving significant performance improvements (e.g., 72% runtime reduction in starvation tests) over

the round-robin scheduler, enhancing xv6's utility for studying scheduling algorithms.

- **Lottery Scheduler**: Implements a probabilistic lottery scheduler, providing proportional fairness based on ticket counts (deviations within 2–3% of expected proportions). This addition offers students a practical exploration of probabilistic scheduling, a flexible alternative to traditional methods.
- **Shared Memory and Semaphores**: Adds efficient IPC and synchronization mechanisms to xv6, enabling concurrent programming scenarios like the producer-consumer problem. The robust implementation, validated under stress conditions, fills a gap in xv6's original design, making it a more comprehensive platform for learning IPC concepts.
- **Educational Value**: Provides a comprehensive set of tools and test programs (`timingtests.c`, `lotterytest.c`, `shmtest.c`, `prodcons.c`) for students to experiment with scheduling and IPC, supported by detailed performance analyses and comparisons with the baseline round-robin scheduler.

These enhancements collectively transform xv6 into a more versatile platform, enabling students to explore advanced operating system concepts through practical implementation and evaluation.

8 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. They enable students to explore advanced scheduling techniques (priority and lottery) and IPC concepts (shared memory and semaphores), supported by practical implementations and detailed performance evaluations. The work also contributes to the broader field by providing a well-documented framework for studying operating system design, with test programs and analyses that can be extended in future research.

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, further enhancing xv6's utility as a teaching platform.

9 Repositories

The complete implementations for each component are available in the following repositories:

- Priority Scheduler: https://github.com/Orestouio/Xv-6-Project
- Lottery Scheduler: https://github.com/Orestouio/Xv6_LotteryExtension
- Shared Memory and Semaphores: https://github.com/Orestouio/Xv6_SharedMemorySemaph

Acknowledgments

I would like to thank my academic supervisors for their guidance throughout this thesis. Additionally, I thank the open-source community for providing the xv6 operating system as a foundation for this work.

Appendix

Priority Scheduler: `scheduler()` Function

```
// Excerpt from proc.c (priority scheduler)
void scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;
    for(;;){
        sti();
        update_priorities();
        acquire(&ptable.lock);
        int min_priority = 11;
        struct proc *min_p = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

```
            if(p->state != RUNNABLE)
                continue;
            if(p->priority < min_priority){
                min_priority = p->priority;
                min_p = p;
            }
        }
        if(min_p){
            c->proc = min_p;
            switchuvm(min_p);
            min_p->state = RUNNING;
            min_p->cpu_time++;
            if(min_p->has_run == 0){
                min_p->first_run_time = ticks;
                min_p->has_run = 1;
            }
            context_switches++;
            sched_log(ticks, min_p->pid, min_p->priority, context_switches);
            swtch(&(c->scheduler), min_p->context);
            switchkvm();
            c->proc = 0;
        }
        release(&ptable.lock);
    }
}
```

Lottery Scheduler: `scheduler()` Function

```
// Excerpt from proc.c (lottery scheduler)
void scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    struct proc *runnable_procs[NPROC];
    int runnable_count;
    int total_tickets;
    int winner;
    int current_tickets;
    static int sched_count = 0;
    c->proc = 0;
    for(;;){
        sti();
```

```
        total_tickets = 0;
        runnable_count = 0;
        acquire(&ptable.lock);
        for(p = ptable; p < &ptable[NPROC]; p++){
            if(p->state == RUNNABLE){
                runnable_procs[runnable_count++] = p;
                total_tickets += p->tickets;
            }
        }
        if(total_tickets == 0){
            release(&ptable.lock);
            continue;
        }
        for(int i = runnable_count - 1; i > 0; i--){
            srand(ticks + lapicid() + randstate + i);
            int j = rand_range(i + 1);
            struct proc *temp = runnable_procs[i];
            runnable_procs[i] = runnable_procs[j];
            runnable_procs[j] = temp;
        }
        srand(ticks + lapicid() + randstate + runnable_count + sched_count);
        winner = rand_range(total_tickets);
        current_tickets = 0;
        for(int i = 0; i < runnable_count; i++){
            p = runnable_procs[i];
            int effective_tickets = p->tickets;
            if(winner < effective_tickets + current_tickets){
                c->proc = p;
                switchuvm(p);
                p->state = RUNNING;
                p->ticks_scheduled++;
                swtch(&(c->scheduler), p->context);
                switchkvm();
                c->proc = 0;
                break;
            }
            current_tickets += effective_tickets;
        }
        release(&ptable.lock);
        sched_count++;
    }
}
```

Shared Memory: `sys_shm_open` Function

```
// Excerpt from proc.c (shared memory)
int sys_shm_open(void)
{
    char *name;
    int size;
    struct proc *curproc = myproc();
    void *va = (void *)(0x60000000 + curproc->shm_count * PGSIZE);
    if(argstr(0, &name) < 0 || argint(1, &size) < 0)
        return -1;
    if(size <= 0 || size > PGSIZE)
        return -1;
    acquire(&ptable.lock);
    int shm_idx = -1;
    for(int i = 0; i < NSHM; i++){
        if(shmtable[i].in_use && strncmp(shmtable[i].name, name, sizeof(
            shmtable[i].name)) == 0){
            shm_idx = i;
            break;
        }
    }
    if(shm_idx == -1){
        for(int i = 0; i < NSHM; i++){
            if(!shmtable[i].in_use){
                shm_idx = i;
                break;
            }
        }
        if(shm_idx == -1){
            release(&ptable.lock);
            return -1;
        }
        acquire(&shmtable[shm_idx].lock);
        shmtable[shm_idx].in_use = 1;
        strncpy(shmtable[shm_idx].name, name, sizeof(shmtable[shm_idx].name)
             - 1);
        shmtable[shm_idx].name[sizeof(shmtable[shm_idx].name) - 1] = '\0';
        shmtable[shm_idx].size = size;
        shmtable[shm_idx].ref_count = 0;
        shmtable[shm_idx].phys_addr = kalloc();
        if(shmtable[shm_idx].phys_addr == 0){
            shmtable[shm_idx].in_use = 0;
```

```
            release(&shmtable[shm_idx].lock);
            release(&ptable.lock);
            return -1;
        }
        memset(shmtable[shm_idx].phys_addr, 0, PGSIZE);
        release(&shmtable[shm_idx].lock);
    }
    if(curproc->shm_count >= MAX_SHM_MAPPINGS){
        release(&ptable.lock);
        return -1;
    }
    acquire(&shmtable[shm_idx].lock);
    if(mappages(curproc->pgdir, va, PGSIZE, V2P(shmtable[shm_idx].phys_addr
        ), PTE_W | PTE_U) < 0){
        if(shmtable[shm_idx].ref_count == 0){
            kfree(shmtable[shm_idx].phys_addr);
            shmtable[shm_idx].in_use = 0;
        }
        release(&shmtable[shm_idx].lock);
        release(&ptable.lock);
        return -1;
    }
    curproc->shm_mappings[curproc->shm_count] = va;
    curproc->shm_objects[curproc->shm_count] = &shmtable[shm_idx];
    curproc->shm_count++;
    shmtable[shm_idx].ref_count++;
    release(&shmtable[shm_idx].lock);
    release(&ptable.lock);
    return (int)va;
}
```

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 Sym-*

posium 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.