xv6 Lottery Scheduler: Implementation and Performance Analysis

Orestis Theodorou

April 28, 2025

Abstract

This report presents the implementation and evaluation of a lottery scheduler in the xv6 operating system, replacing the default round-robin scheduler with a probabilistic mechanism. 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%). Performance is assessed using the `lotterytest.c` utility across three workloads under a single-core configuration (`CPUS := 1`). The lottery scheduler achieves proportional fairness, with deviations within 2–3% of expected proportions (e.g., Test 1: 51.1%, 31.8%, 16.9%), significantly improving over the round-robin scheduler's equal scheduling (33% each). Optimizations like Fisher-Yates shuffling and increased scheduling events reduce variability, though runtime overhead (e.g., 1810.6 ticks in Test 1) suggests areas for improvement. This implementation enhances xv6's scheduling flexibility, making it a valuable educational tool.

1 Introduction

This report presents the implementation and evaluation of a lottery scheduler for the xv6 operating system, a teaching-oriented system derived from UNIX, as part of a dissertation project. The xv6 system originally employs a round-robin scheduler, which allocates equal time slices to all processes. In contrast, the lottery scheduler introduced in this work assigns processes a number of tickets (e.g., 30, 20, 10), and the probability of a process being scheduled is proportional to its ticket count (e.g., expected proportions of 50%, 33%, 16.67% for tickets 30, 20, 10, respectively). The performance of this scheduler is evaluated using a custom benchmarking utility, `lotterytest.c`, under the default single-core configuration (`CPUS := 1`). This configuration ensures a direct comparison with the round-robin scheduler, as multi-core setups (`CPUS := 2`) introduce parallelism and scheduling dynamics that require separate analysis, which is deferred to future work. This report details the implementation, evaluates the performance across three distinct tests, compares the lottery scheduler with the round-robin baseline, and demonstrates its effectiveness in achieving proportional fairness in a single-core environment.

2 Related Work

Lottery scheduling, proposed by Waldspurger and Weihl [1], provides a probabilistic approach to resource allocation, known for its fairness and flexibility in operating systems. The xv6 operating system, designed for educational purposes [2], offers a lightweight platform for experimenting with scheduling algorithms. This work adapts lottery scheduling to xv6's single-core environment, introducing mechanisms to enhance fairness and reduce variability, building on prior research in probabilistic scheduling.

3 Baseline: Establishing the Round-Robin Foundation

To evaluate the lottery scheduler, a baseline was established using xv6's default round-robin scheduler under the single-core configuration (`CPUS := 1`). The round-robin scheduler allocates equal time slices to each process in a cyclic manner, ensuring fairness but lacking the ability to allocate CPU time proportionally based on ticket counts. The `lotterytest.c` utility was used to measure performance across three workloads, each designed to assess different aspects of the scheduler. Each test was executed multiple times, with results averaged to determine scheduling proportions and runtime (1 tick = 10ms). The tests are defined as follows:

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

3.1 Baseline Observations

The round-robin scheduler's performance provides a reference for comparison. For Test 1 (CPU-heavy), it averaged 87 ticks, with each of the 3 processes receiving equal scheduling (33% each), ignoring ticket counts. In Test 2 (Switch overhead), it averaged 70.5 ticks for a smaller process set, with equal scheduling (33% each). These results, derived from prior evaluations, serve as a benchmark for assessing the lottery scheduler's performance under similar single-core conditions.

3.2 Enhancing Round-Robin with Scheduling Counts

To enable detailed comparison, the round-robin scheduler was augmented to track scheduling events, particularly for Tests 1 and 2. This involved adding a scheduling counter field, `ticks_scheduled`, to the process structure in `proc.c`, which is incremented in `scheduler()` during each `swtch` call. Additionally, a system call was introduced to retrieve process information, including scheduling counts, allowing the `lotterytest.c` utility to measure scheduling events per process before and after each test. This enhancement confirmed that the round-robin scheduler distributes schedules equally (approximately 33% per process) in both Tests 1 and 2, providing a benchmark for evaluating the lottery scheduler's proportional fairness under single-core conditions.

4 Round-Robin Output

4.1 With `CPUS := 1`

```
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%
```

5 Lottery Scheduler: Implementation Overview

The lottery scheduler was implemented to replace the round-robin scheduler in xv6, introducing a probabilistic scheduling mechanism where processes are selected based on their ticket counts. The implementation was designed and tested under the default single-core configuration (`CPUS := 1`) to ensure a direct comparison with the round-robin scheduler. The following modifications were made to the xv6 system, as reflected in the provided `proc.c` and `lotterytest.c` files:

- `proc.c`:
  - **Lottery Scheduling**: The `scheduler()` function was modified to implement lottery scheduling. It collects all runnable processes into an array (`runnable_procs`), computes the total tickets, and uses a random number generator to select a winner based on ticket proportions. The winner is scheduled by calling `swtch`, and its `ticks_scheduled` counter is incremented. To reduce ordering bias, the `runnable_procs` array is shuffled before selection using the Fisher-Yates algorithm, seeded with a combination of `ticks`, CPU ID (`lapicid()`), `randstate`, and iteration indices.
  - **Random Number Generator**: An Xorshift-based random number generator was implemented (`srand`, `rand`, `rand_range`) to ensure high-quality randomness. The generator avoids modulo bias in `rand_range` by rejecting values that would skew the distribution, and it is seeded dynamically with `ticks`, CPU ID, and other variables to minimize predictability and ensure fairness across scheduling events.
  - **Scheduling Log**: The `ticks_scheduled` field tracks scheduling events, which are used for performance analysis in `lotterytest.c`.
  - **Process Management**: Functions like `fork()`, `allocproc()`, and `userinit()` were updated to initialize `tickets` and `ticks_scheduled`, ensuring new processes inherit ticket counts appropriately (e.g., in `fork`, `np->tickets = curproc->tickets`). In `allocproc`, new processes are initialized with a default ticket count of 1.
- `lotterytest.c`:
  - **Test Functions**: The `run_workload_test` function is used for Tests 1 and 3, forking 3 processes with specified ticket counts (30-20-10 for Test 1, 50-10-1 for Test 3), performing iterations (500M for Test 1, 10M for Test 3), and yielding periodically (every 5,000 iterations). The `run_switch_test` function for Test 2 forks 50 processes (17 with 30 tickets, 17 with 20 tickets, 16 with 10 tickets) in a C-A-B order, each performing 100M iterations and yielding every 100,000 iterations. Both functions use `uptime()` to measure runtime and collect scheduling statistics.
  - **Average Calculation**: The `print_average_results` function calculates and prints the average scheduling proportions over 5 runs, providing a clear summary of performance.
  - **Test Independence**: A 50ms delay (`sleep(5)`) is added between each test run in `main()` to ensure system stabilization and minimize interference between tests.
- **System Call Support**: The implementation relies on system calls such as `settickets` to set ticket counts for processes and another system call to retrieve process information, including scheduling counts, which are used by `lotterytest.c` to analyze performance. These system calls ensure thread safety by synchronizing access to the process table.

5.1 Optimization Steps

To achieve the current performance, several optimizations were applied based on the improvements discussed throughout the project:

- **Variability Reduction**: Initial tests for Test 1 showed high variability (e.g., Process A: 43% to 62%) due to a low number of scheduling events (191 per run) and ordering bias in process selection.
- **Shuffling Runnable Processes**: Fisher-Yates shuffling was added in `scheduler()` to eliminate ordering bias, reducing variability.
- **Improved Randomness**: The Xorshift-based generator, seeded with `ticks`, CPU ID, `randstate`, and iteration indices, ensures better distribution of scheduling events. The generator was further enhanced by incorporating entropy from multiple sources, reducing deviations in average proportions (from 6.5% to 1%) and tightening variability (from 22% to 10% for Process A in Test 1).
- **Increased Scheduling Events**: Test 1 iterations were increased from 50M to 500M, raising the number of scheduling events from 191 to 753 per run (based on 1925 + 1201 + 639 schedules over 5 runs), reducing variability (e.g., Process A: 45% to 55%).
- **Yield Frequency**: The yield interval was reduced to 5,000 iterations in Tests 1 and 3 (from a higher value), increasing scheduling opportunities from 605 to 792–816 per run in Test 1, stabilizing proportions.
- **Interleaved Forking in Test 2**: A C-A-B forking order was adopted to balance initial scheduling, ensuring low variability (e.g., Process A: 50% to 50%).

5.2 Challenges and Solutions

Several challenges were encountered during the implementation of the lottery scheduler:

- **High Variability in Initial Tests**: Early results for Test 1 showed high variability (e.g., Process A: 43% to 62%) due to a low number of scheduling events (191 per run) and ordering bias. This was resolved by increasing iterations to 500M (raising events to 753 per run) and adding Fisher-Yates shuffling to eliminate ordering bias, reducing variability to 45%–55%.
- **Poor Randomness Quality**: Initial use of a simple random number generator led to skewed distributions (deviations up to 6.5%). An Xorshift-based generator was implemented, seeded with multiple sources (`ticks`, CPU ID, `randstate`, iteration indices), reducing deviations to 1%.
- **Test Interference**: Early test runs showed interference between workloads (e.g., Test 1 affecting Test 2). A 50ms delay (`sleep(5)`) was added between test runs in `lotterytest.c`, ensuring system stabilization and consistent results.
- **Expected Proportions Display in Test 3**: The output displayed rounded proportions (81%-16%-1%) due to integer division in `lotterytest.c`. While the actual expected values are 81.97%-16.39%-1.64%, this display issue does not affect the scheduler's performance and can be addressed by adjusting the output format in future iterations.

This implementation enables the lottery scheduler to achieve proportional fairness, reduce variability through increased scheduling events and shuffling, and provide detailed performance metrics for evaluation in a single-core environment.

6 Lottery Scheduler Output

6.1 With `CPUS := 1`

The lottery scheduler was evaluated over 5 runs for each test, with the following results:

```
Test 1: CPU-heavy: Schedules: A=51.1%, B=31.8%, C=16.9%
Test 2: Switch overhead: Schedules: A=50.5%, B=32.2%, C=17.2%
Test 3: Starvation check: Schedules: A=79.3%, B=18.6%, C=1.9%
```

7 Final Results and Comparison

The lottery scheduler's performance was evaluated using `lotterytest.c` and compared against the round-robin scheduler's baseline with `CPUS := 1`, focusing on runtime (ticks) and scheduling proportions. Each test's results are analyzed to explain the observed performance, justify the representativeness of the tests, and compare the lottery scheduler with the round-robin scheduler, demonstrating the successful implementation of lottery-based scheduling in a single-core environment.

- **Test 1 (CPU-heavy)**: The lottery scheduler achieves an average runtime of 1810.6 ticks per run (range: 1787–1851 ticks), with scheduling proportions of A: 51.1%, B: 31.8%, C: 16.9% (expected: 50%, 33%, 16.67%), compared to the round-robin scheduler's 87 ticks (range: 83–100 ticks) and equal proportions (A: 33%, B: 33%, C: 33%). In this test, 3 processes each perform 500 million iterations concurrently, with tickets 30, 20, and 10. The lottery scheduler's `scheduler()` function uses random selection proportional to ticket counts, achieving deviations of 1.1%, 1.2%, and 0.2% for Processes A, B, and C, respectively, all within a target threshold of 2–3%. The increased iterations (500M) and reduced yield interval (every 5,000 iterations) provide 753 scheduling events per run (1925 + 1201 + 639 over 5 runs), reducing variability (e.g., Process A: 45% to 55%, standard deviation $\sigma \approx 3.8\%$). The round-robin scheduler, ignoring tickets, schedules all processes equally, which is unfair in this context (e.g., Process A should receive 50% of the schedules, not 33%). The significant increase in runtime (1810.6 vs. 87 ticks) is primarily due to the higher number of iterations (500M vs. 100M in the baseline), as well as the overhead of the lottery mechanism (random number generation, shuffling). However, the achieved proportional fairness validates the scheduler's core functionality.
- **Test 2 (Switch overhead)**: The lottery scheduler achieves scheduling proportions of A: 50.5%, B: 32.2%, C: 17.2% (expected: 50%, 33%, 16.67%), compared to the round-robin scheduler's equal proportions (A: 33%, B: 33%, C: 33%) and 70.5 ticks (range: 67–83 ticks). This test involves 50 processes (17 with 30 tickets, 17 with 20 tickets, 16 with 10 tickets), each performing 100 million iterations. The lottery scheduler achieves deviations of 0.5%, 0.8%, and 0.5% for Processes A, B, and C, well within a 2–3% target. The low variability (e.g., Process A: 50% to 50%, $\sigma \approx 0.2\%$) reflects the large number of scheduling events (13211 per run, based on 33375 + 21312 + 11369 over 5 runs), and the interleaved C-A-B forking order ensures balanced initial scheduling. Runtime was not measured in the provided output, but the significant

number of processes (50 vs. the baseline's smaller set) and lottery scheduling overhead (random selection, shuffling) likely result in a higher runtime compared to the baseline's 70.5 ticks. The lottery scheduler's ability to closely match the expected proportions (unlike the round-robin's equal scheduling) demonstrates its effectiveness in handling rapid process turnover with ticket-based fairness.

- **Test 3 (Starvation check)**: The lottery scheduler averages 42 ticks per run (range: 42–42 ticks), with scheduling proportions of A: 79.3%, B: 18.6%, C: 1.9% (expected: 81.97%, 16.39%, 1.64%), compared to the round-robin scheduler's 87 ticks (range: 83–100 ticks) and equal proportions (A: 33%, B: 33%, C: 33%) for a similar workload. This test involves 3 processes with tickets 50, 10, and 1, each performing 10 million iterations. The lottery scheduler achieves deviations of 2.7%, 2.2%, and 0.3%, with Processes A and C within a 2–3% threshold and Process B slightly outside. The output displays expected proportions as 81%-16%-1% due to integer division rounding in `run_workload_test` (e.g., (50 * 100) / 61 = 81), but the actual expected values are 81.97%-16.39%-1.64%. Variability is reasonable (e.g., Process A: 77% to 80%, $\sigma \approx 1.3\%$), with 834 scheduling events per run (3310 + 779 + 80 over 5 runs). The lottery scheduler successfully prevents starvation of Process C (1 ticket), which is scheduled 10–19 times per run, confirming fairness. The runtime decrease (42 vs. 87 ticks) reflects the lower number of iterations (10M vs. 100M in the baseline). The round-robin scheduler's equal scheduling fails to reflect the intended proportional allocation, underscoring the lottery scheduler's advantage in such scenarios.

7.1 Representativeness of the Tests

The tests are representative of real-world scheduling scenarios and suitable for evaluating the lottery scheduler's performance:

- **Test 1 (CPU-heavy)**: This test simulates compute-intensive workloads common in scientific computing or data processing, where processes require significant CPU time. The use of 500M iterations ensures a sufficient number of scheduling events (753 per run), allowing the lottery scheduler to demonstrate proportional fairness over a reasonable duration (1810.6 ticks, or 18 seconds). The test's design aligns with typical operating system benchmarks for CPU scheduling, making it representative of scenarios where proportional CPU allocation is desired.
- **Test 2 (Switch overhead)**: This test evaluates the scheduler's performance under high contention, with 50 processes competing for CPU time, a scenario typical in multi-user systems or servers with many concurrent tasks (e.g., web servers). The large number of scheduling events (13211 per run) ensures statistical stability, and the C-A-B forking order minimizes initial bias, making the test representative of workloads with frequent context switches.
- **Test 3 (Starvation check)**: This test assesses the scheduler's ability to prevent starvation in the presence of highly skewed ticket distributions (50-10-1), a critical requirement for fairness in operating systems. The test simulates scenarios where a low-priority process (e.g., a background task) must not be indefinitely delayed by high-priority processes (e.g., foreground applications). The consistent scheduling of Process C (1 ticket) validates the scheduler's fairness, and the test's design is representative of priority-based workloads in real-world systems.
- **Test Independence**: The 50ms delay (`sleep(5)`) between test runs ensures system stabilization, preventing interference between tests (e.g., residual process states affecting subsequent runs). This design choice enhances the reliability and representativeness of the results.

7.2 Key Observations

- The lottery scheduler achieves proportional fairness in all tests, with deviations within 2–3% for most processes, except for Process B in Test 3 (2.2% deviation), which is still acceptable for a lottery scheduler.
- Test 1 shows moderate variability (e.g., Process A: 45% to 55%) due to the limited number of scheduling events (753 per run), a known characteristic of lottery scheduling with few processes.
- Test 2 demonstrates low variability (e.g., Process A: 50% to 50%) due to the large number of scheduling events (13211 per run), confirming the scheduler's stability under high contention.
- Test 3 confirms that the scheduler prevents starvation, with Process C (1 ticket) consistently scheduled, and the proportions are close to expected, despite the minor display discrepancy in expected values (81%-16%-1% vs. 81.97%-16.39%-1.64%).
- Compared to the round-robin scheduler, the lottery scheduler provides ticket-based fairness, ensuring processes are scheduled according to their ticket proportions rather than equally, a significant improvement for workloads where proportional scheduling is desired.

7.3 Conclusion

The lottery scheduler successfully implements its intended functionality: scheduling processes probabilistically based on ticket counts, with mechanisms like Fisher-Yates shuffling and an Xorshift-based random number generator ensuring fairness and reducing variability. The performance results are robust—Tests 1, 2, and 3 achieve deviations within 2–3% for most processes, validating the scheduler's design. Compared to the round-robin scheduler, the lottery scheduler offers significant improvements in scenarios requiring proportional scheduling, as demonstrated by all three tests. The tests are representative of real-world workloads (compute-intensive, high-contention, and priority-based), and the `sleep(5)` delay ensures result consistency by minimizing interference. This implementation enhances xv6's utility as an educational tool, enabling students to explore probabilistic scheduling concepts and their practical implications. The project is ongoing, with further refinements planned to address variability and runtime overhead, as noted in the future enhancements.

8 Future Enhancements

While the lottery scheduler demonstrates robust results in the single-core configuration, areas for improvement remain:

- **Test 1 Variability**: The variability in Test 1 (e.g., Process A: 45% to 55%) could be reduced by further increasing the number of scheduling events (e.g., to 1B iterations) or introducing a hybrid mechanism, such as time-slicing within lottery scheduling, to ensure more consistent short-term fairness.
- **Runtime Overhead**: The increased runtime in Test 1 (1810.6 vs. 87 ticks) suggests overhead from the lottery mechanism (random number generation, shuffling). Optimizing these operations (e.g., using a faster random number generator) could reduce runtime while maintaining fairness.

- **Multi-core Optimization**: Evaluate performance in a multi-core environment (`CPUS := 2`) to ensure the lottery scheduler scales effectively, addressing potential scheduling dynamics introduced by parallelism, which were not assessed in this phase.

9 Repository and Final Remarks

- The complete implementation—including `proc.c`, `lotterytest.c`, and supporting files—is available at `https://github.com/Orestouio/Xv6_LotteryExtension`.
- This report documents the current state of the lottery scheduler under the single-core configuration (`CPUS := 1`), demonstrating its superiority over the round-robin scheduler in achieving proportional fairness across diverse workloads. The project is ongoing, with further refinements planned to address variability and runtime overhead, as noted in the future enhancements.

References

[1] 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).*

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