Lab 2 Report on XV6 Operating System Scheduler Modification
Team 16

Xiao Fan, Kaya Gokalp, Ashwin Nellimuttath, and Hemanth Paladugu

November 17, 2023

Contents

1 Introduction

In Lab 2, we focused on enhancing the XV6 operating system's scheduler. Our objective was to implement both lottery and stride scheduling algorithms, requiring extensive modifications to various kernel files and the addition of new system calls. This report details the implementation process, the challenges faced, and the insights gained.

2 Demonstration Video Link

A comprehensive demonstration of the schedulers can be viewed at the following link, showcasing the implemented functionality in a live setting:

`http://example.com/demo-video`

3 List of Files Modified

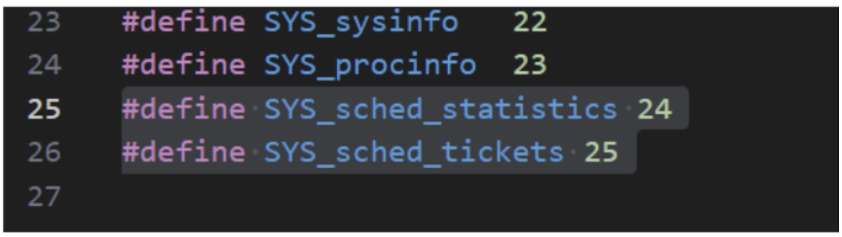
The following files were modified for the implementation:

- kernel/syscall.h
- kernel/syscall.c
- kernel/sysproc.c
- kernel/proc.c
- kernel/proc.h
- kernel/defs.h
- Makefile
- user/usys.pl
- user/user.h
- user/lab2test.c

4 Detailed Explanation of Changes

4.1 kernel/syscall.h

Added new syscall numbers at lines 25 & 26 to incorporate the newly introduced system calls.



```
23  #define SYS_sysinfo   22
24  #define SYS_procinfo  23
25  #define SYS_sched_statistics 24
26  #define SYS_sched_tickets 25
27
```

Changes in Syscall.h

4.2 kernel/syscall.c

Declared syscall handler functions at lines 113, 114, 142, 143 to manage the execution of new system calls.

```
112 extern uint64 sys_procinfo(void);
113 extern uint64 sys_sched_statistics(void);
114 extern uint64 sys_sched_tickets(void);
```

4.3 kernel/sysproc.c

Implemented two syscall functions: `sys_sched_statistics` and `sys_sched_tickets` for retrieving scheduling statistics and setting ticket values for processes.

```
uint64
sys_sched_statistics(void)
{
  sched_statistics();
  return 0;
}

uint64
sys_sched_tickets(void)
{
  uint64 tick;
  argaddr(0,(void *)&tick);
  sched_tickets(tick);
  return 0;
}
```

Changes in sys.proc.c

4.4 kernel/proc.h

Introduced variables for managing tickets in lottery scheduling and for stride calculation in stride scheduling.

```
  int ticket_val;              // Value of the ticket
  int num_ticket;              // Number of tickets used
  int pass;                    // Pass value
  int stride;                  // Stride value
```

Changes in proc.h

4.5 kernel/proc.c

This section details the implementation of code for generating random ticket values and initializing ticket management variables, which are crucial for the lottery scheduler. The code facilitates the generation of random values for the tickets.

```c
void            fill_pinfo(struct proc *curr_proc, struct pinfo *in);
int             sched_statistics(void);
int             sched_tickets(int);
```

We initialized the variables `ticket_val`, `pass`, and `stride` in `allocproc`, which were previously declared in `proc.h`.

```c
// The rand function given in the project file
unsigned short lfsr = 0xACE1u;
unsigned short bit;

unsigned short rand()
{
  bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) & 1;
  return lfsr = (lfsr >> 1) | (bit << 15);
}
```

```c
static struct proc*
allocproc(void)
{
  struct proc *p;

  for(p = proc; p < &proc[NPROC]; p++) {
    acquire(&p->lock);
    p->ticket_val = def_ticket;
    p->pass = 0;
    p->stride =  def_k/ p->ticket_val;
    if(p->state == UNUSED) {
      // p->ticket_val;
      goto found;
    } else {
      release(&p->lock);
    }
  }
```

Furthermore, we set up the initialization of the Process ID (PID), state, syscall count, ticks, and the number of tickets held by the process.

```c

found:
  p->pid = allocpid();
  p->state = USED;
  p->num_syscalls = 0;
  p->ticks = 0;
#if defined(LOTTERY)
  p->tickets = 1;
#else
  p->tickets = 10000;
#endif
```

Additionally, this function iterates through the process table and prints scheduling statistics for each active process. These statistics include the process ID (PID), the process name, the number of tickets assigned to the process, and the total number of ticks the process has accumulated.

```c
*/
int sched_statistics(void)
{
  struct proc *p;
  for (p = proc; p < &proc[NPROC]; p++)
  {
    if (p->pid > 0)
    {
      printf("%d(%s): tickets:%d, ticks:%d\n", p->pid, p->name, p->tickets, p->ticks);
    }
  }
  return 0;
}
```

```c
int sched_tickets(int ticket_val)
{
  // we make sure ticket val does not esceed 10000 as given in the question
  if (ticket_val <= 10000)
  {
    // process
    struct proc *p = myproc();
    // We assign the ticket to the process's tickets
    p->tickets = ticket_val;

    //  Calculate the stride for the Stride Scheduling algorithm
    p->stride = 10000 / ticket_val;
    p->pass = p->stride;
  }
  return 0;
}
```

Another function is implemented to set the number of tickets for the currently running process. This function ensures that the provided ticket value does not exceed 10,000, then assigns the ticket value to the process's tickets. Additionally, it calculates the stride and initializes the pass value for the Stride Scheduling algorithm.

```c
int getTotalTickets(void)
{
  struct proc *p;
  int total = 0;
  for (p = proc; p < &proc[NPROC]; p++)
  {
    acquire(&p->lock);
    if (p->state == RUNNABLE)
    {
      total += p->tickets;
    }
    release(&p->lock);
  }
  return total;
}
```

Furthermore, a function is included that iterates through the process table to calculate the total number of tickets for all runnable processes, considering only those in the RUNNABLE state.

```c
void scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for (;;)
  {
```

Finally, each CPU calls the scheduler() after setting itself up. The scheduler never returns. It continuously loops, performing the following actions: - Choosing a process to run. - Switching to start running that process. - Eventually, that process transfers control back to the scheduler via a context switch.


```c
void scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();

  c->proc = 0;
  for (;;)
  {
```


4.6 Makefile

Included the path for the test file `lab2test` to facilitate the testing of newly defined syscalls.

```make
	$U/_lab1test\
	$U/_lab2test\
```

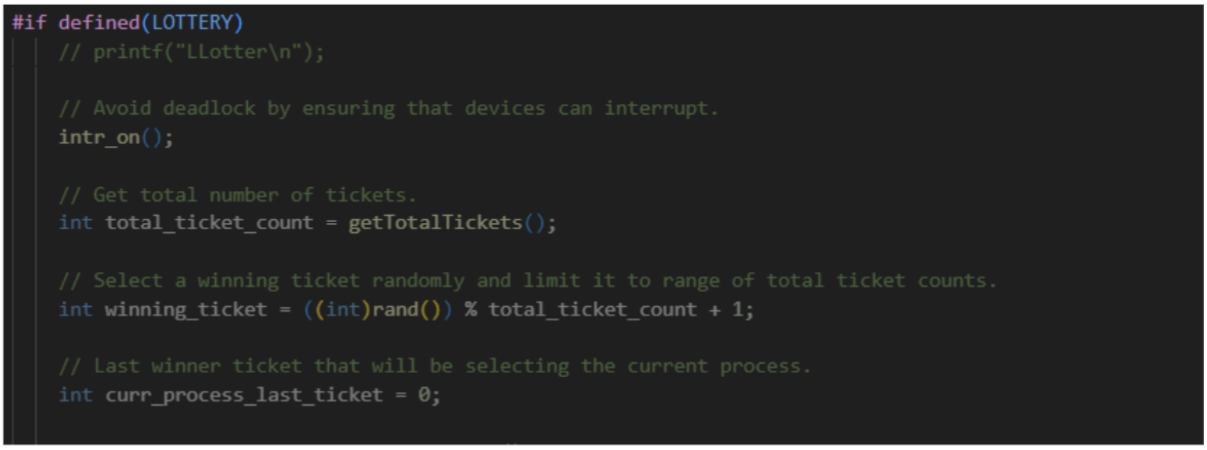
5 Scheduler Implementation

5.1 Lottery Scheduler

The implementation of the Lottery Scheduler involved several key steps:

- We implemented the `getTotalTickets()` function to calculate the total number of lottery tickets available in the system.
- A winning ticket is randomly selected to determine which process will be executed next.
- We iterated through the processes to match the winning ticket with the corresponding process.

The `getTotalTickets()` function is crucial as it determines the total number of lottery tickets in the system. The `winning_ticket` is generated randomly within the range from 1 to the total ticket count, thereby determining which process will be executed.



```
#if defined(LOTTERY)
    // printf("LLotter\n");

    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    // Get total number of tickets.
    int total_ticket_count = getTotalTickets();

    // Select a winning ticket randomly and limit it to range of total ticket counts.
    int winning_ticket = ((int)rand()) % total_ticket_count + 1;

    // Last winner ticket that will be selecting the current process.
    int curr_process_last_ticket = 0;
```

The process involves iterating through the `proc` structure to identify the process holding the winning ticket. The variable `curr_process_last_ticket` is used to keep track of the cumulative ticket count as the iteration proceeds. If the value of the winning ticket is greater than the cumulative ticket count, it implies that the winning ticket has not been found yet. In such cases, we release the lock on the current process and continue to the next one.

```c
int curr_process_last_ticket = 0;

// int total_tickets = getTotalTickets();
for (p = proc; p < &proc[NPROC]; p++)
{
  acquire(&p->lock);
  if (p->state != RUNNABLE)
  {
    release(&p->lock);
    continue;
  }

  curr_process_last_ticket+=p->tickets;

  if (winning_ticket > curr_process_last_ticket) {
    // We still couldn't find the winner, we should check next proc.
    release(&p->lock);
    continue;
  }

```

Upon finding a winning ticket value that is less than or equal to the cumulative ticket count, we designate the current process as the winning process. This process is then set to the Running state, its context switch is updated, and the lock on the winning process is released.

```c

    p->state = RUNNING;
    p->ticks += 1;
    c->proc = p;
    swtch(&c->context, &p->context);

    c->proc = 0;
  release(&p->lock);
  break;
}

```

5.2 Stride Scheduler

In the Stride Scheduler, we incorporated several key features:

- Logic was introduced to track the minimum pass value among the processes.
- The process with the lowest pass value is prioritized to run next.
- We adjust the pass value of the running process to ensure fair scheduling.

The `current_proc` variable was initialized to monitor the process with the lowest pass value. Simultaneously, `minPass` was set to `INT_MAX` to represent the largest possible integer value. This configuration is essential for iterating through the process table (`Proc`) to identify the process with the minimum pass value.

```c
#elif defined(STRIDE)
    // printf("Stride\n");
    struct proc *current_proc = 0;
    // both values of minpass should work
    // int minPass = -1;
    int minPass = INT_MAX;

    // Avoid deadlock by ensuring that devices can interrupt.
    intr_on();

    for (p = proc; p < &proc[NPROC]; p++)
    {
      acquire(&p->lock);

```

When a process is identified with a pass value less than or equal to the current minimum and is in a runnable state, we update `minPass` and assign `current_proc` to this process. This step involves acquiring a lock on the process deemed to be the winner and adjusting its pass value (`pass = pass + stride`). Following this, the process's state is set to 'Running'. The scheduler then performs a context switch and subsequently releases the lock on the winning process.

```c

      if (p->state == RUNNABLE && (p->pass <= minPass || minPass < 0))
      {

        minPass = p->pass;
        current_proc = p;
      }
      release(&p->lock);
    }

    if (current_proc != 0 && current_proc->state == RUNNABLE)
    {
      acquire(&current_proc->lock);
      // Set the current process in the CPU structure and Adjust the pass value
      c->proc = current_proc;
      current_proc->pass += current_proc->stride;

      current_proc->state = RUNNING;
      current_proc->ticks += 1;
      swtch(&c->context, &current_proc->context);
      c->proc = 0;
      release(&current_proc->lock);
    }
```

5.3 Round Robin Scheduler

The Round Robin Scheduler was enhanced with several important modifications to improve its efficiency and reliability:

- We implemented the use of `intr_on()` to prevent potential deadlocks. This function ensures that interrupts are enabled, reducing the likelihood of deadlock scenarios during process scheduling.
- The scheduler iterates through the process table (`proc`) to locate the next process that is in a runnable state. This step is crucial for maintaining the fairness and efficiency characteristic of Round Robin scheduling.
- Upon identifying a runnable process, its state is set to 'Running'. This state change is a key part of the context-switching mechanism, which is central to process scheduling in operating systems.
- A context switch is performed, and the lock on the process is released once the process execution is completed. This step is vital for allowing other processes to be scheduled and executed, ensuring that each process gets its fair share of CPU time.

Additionally, entries for `sched_statistics` and `sched_tickets` were made. These provide a kernel-level interface for applications, allowing them to trigger and utilize these functionalities. The inclusion of these entries is a part of enhancing the XV6 scheduler to be more responsive and versatile in handling various scheduling tasks.

Note: The figures corresponding to these implementations can be inserted here if available.

```
#else
    //  round-robin scheduler
    //  Avoid deadlock by ensuring that devices can interrupt.
    intr_on();
    for (p = proc; p < &proc[NPROC]; p++)
    {
      acquire(&p->lock);

      if (p->state == RUNNABLE)
      {
        // Switch to chosen process.  It is the process's job
        // to release its lock and then reacquire it
        // before jumping back to us.
        p->state = RUNNING;
        p->ticks += 1;
        c->proc = p;
        swtch(&c->context, &p->context);

        // Process is done running for now.
        // It should have changed its p->state before coming back.
        c->proc = 0;
      }
      release(&p->lock);
    }
#endif
```

6 Demonstration

This section demonstrates the practical application and functionality of the lottery and stride schedulers within the XV6 operating system, reflecting the modifications we implemented.

6.1 Lottery Scheduler Demonstration

The lottery scheduler's functionality is showcased in a test run, evidenced by the screenshot below. The command `$ lab2 100 3 30 20 10` triggers the scheduling of three processes with 30, 20, and 10 tickets respectively. The outcome indicates that processes with a higher number of tickets tend to accumulate more CPU time, demonstrating the probabilistic and weighted nature of the lottery scheduler. This is consistent with the expected behavior, where processes with more tickets have an increased likelihood of selection in each scheduling round.

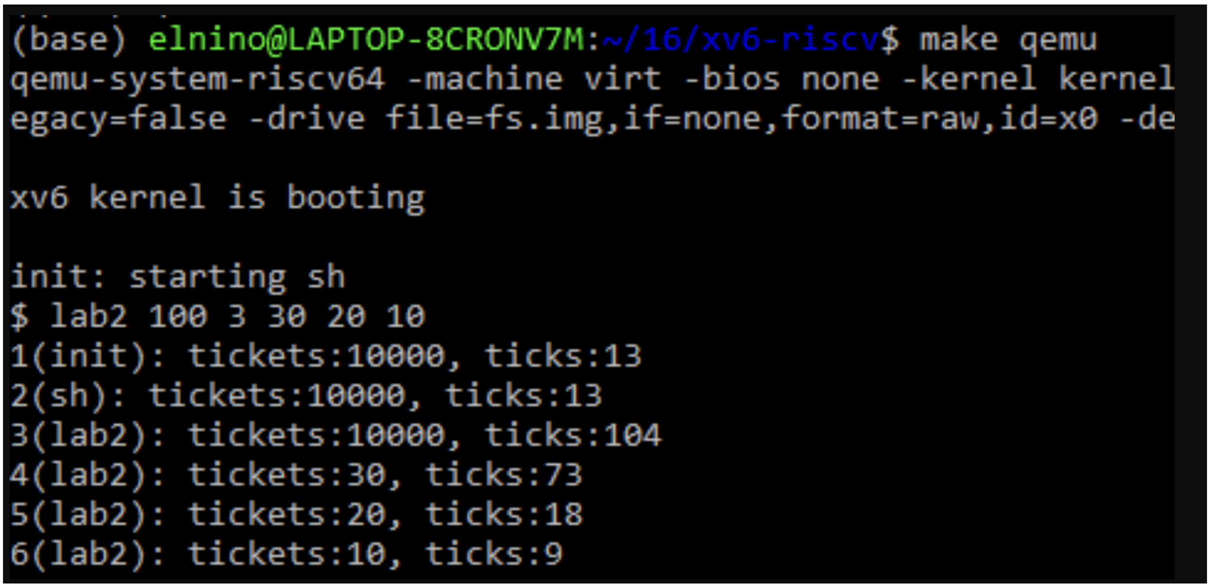


Figure 1: Lottery scheduler output showing CPU time allocation among processes.

Further analysis through graphical representation provides an overview of CPU time allocation in a lottery-scheduled system. Processes with varying ticket counts are scheduled in a manner that, over time, aligns with the ideal expectation of proportional distribution, affirming the scheduler's effectiveness.

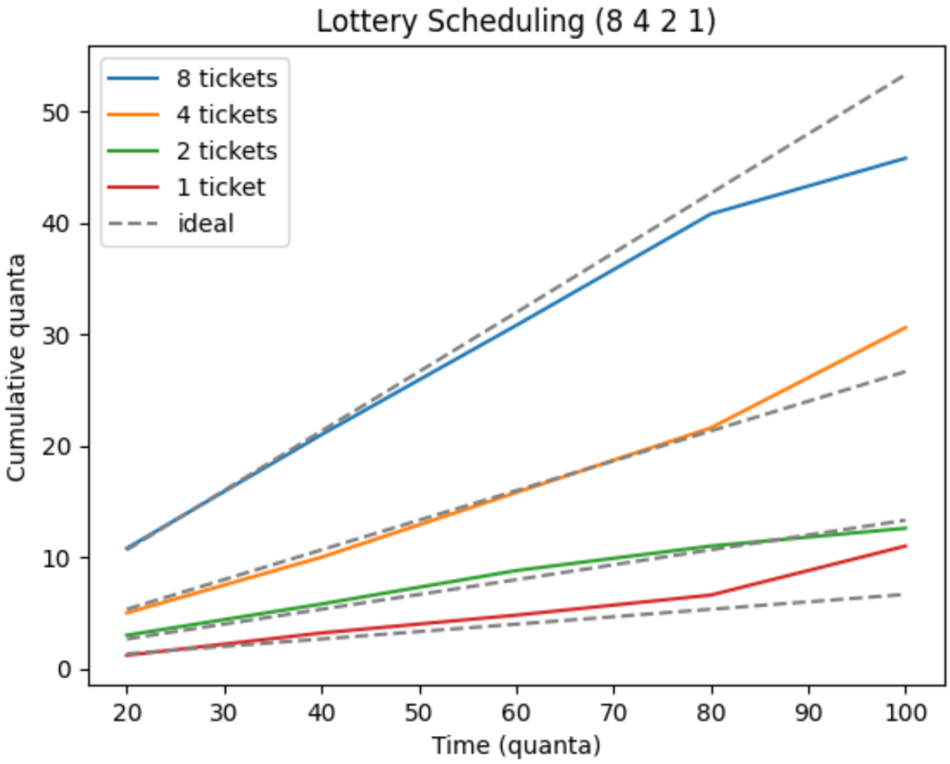


Figure 2: CPU time distribution in a lottery-scheduled environment.

When every process holds an identical number of tickets, the randomness of the lottery scheduler is apparent as the CPU time dispersal among processes fluctuates. This variability is a hallmark of the lottery scheduling mechanism and its inherent randomness, distinguishing it from deterministic scheduling approaches.

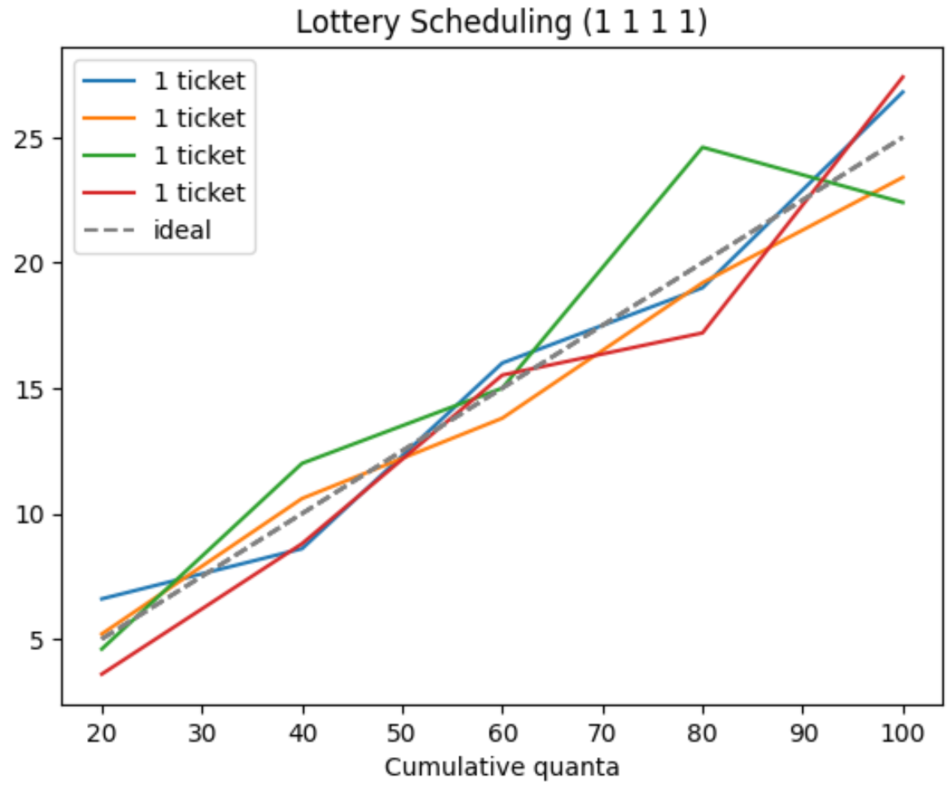


Figure 3: Lottery scheduling behavior with uniform ticket distribution.

6.2 Stride Scheduler Demonstration

The stride scheduler's demonstration, as captured in the following screenshot, reveals its capacity to distribute CPU time based on ticket and stride values. The stride mechanism ensures a more predictable and equitable CPU time allocation, as shown in the test with the command `$ lab2 100 3 30 20 10`, where CPU ticks are distributed in correlation to the stride values assigned to each process.

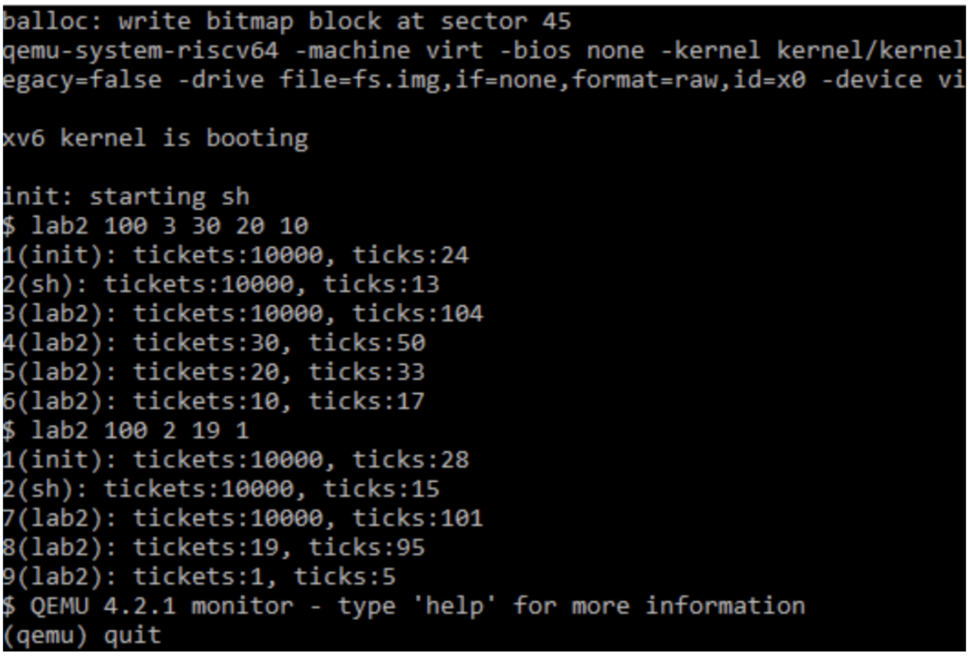


Figure 4: Operational demonstration of the stride scheduler allocating CPU ticks.

The accompanying graph depicts the cumulative CPU time each process receives under the stride scheduling policy. The near-linear progression of CPU allocation demonstrates the scheduler's commitment to proportional distribution, where processes with a greater number of tickets receive a corresponding increase in CPU time.

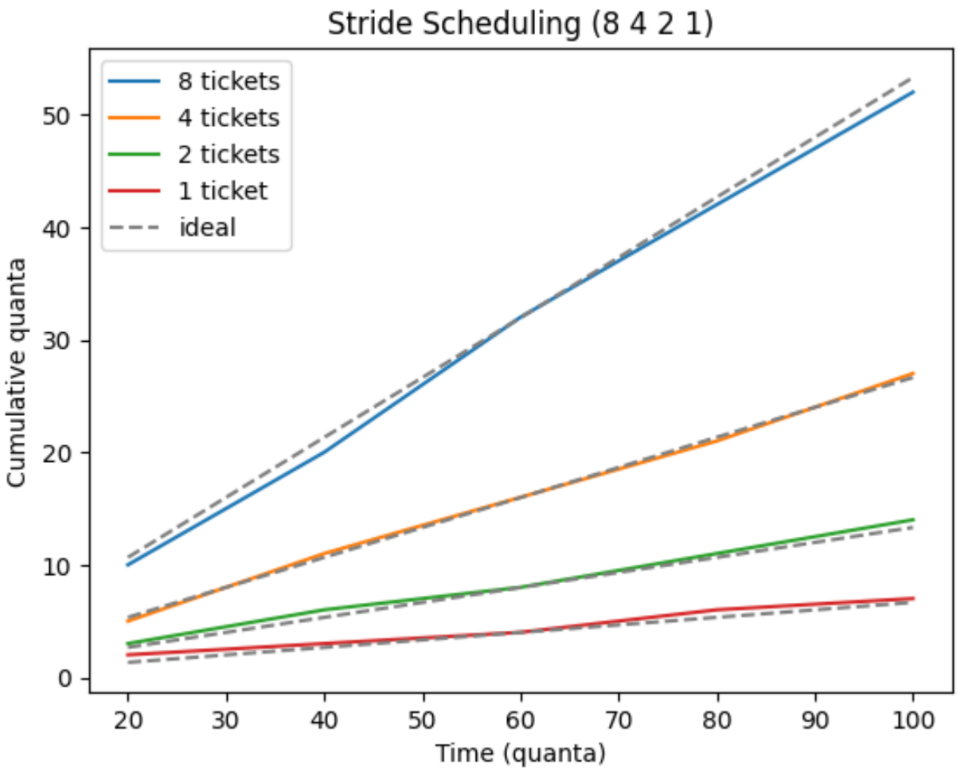


Figure 5: Visual representation of CPU time allocation in stride scheduling.

In conditions where processes have equal ticket allocations, the stride scheduler's deterministic distribution is clearly demonstrated. The plot showcases an equitable CPU time share across processes, closely adhering to the ideal distribution and underscoring the stride scheduler's fairness in equal-ticket scenarios.

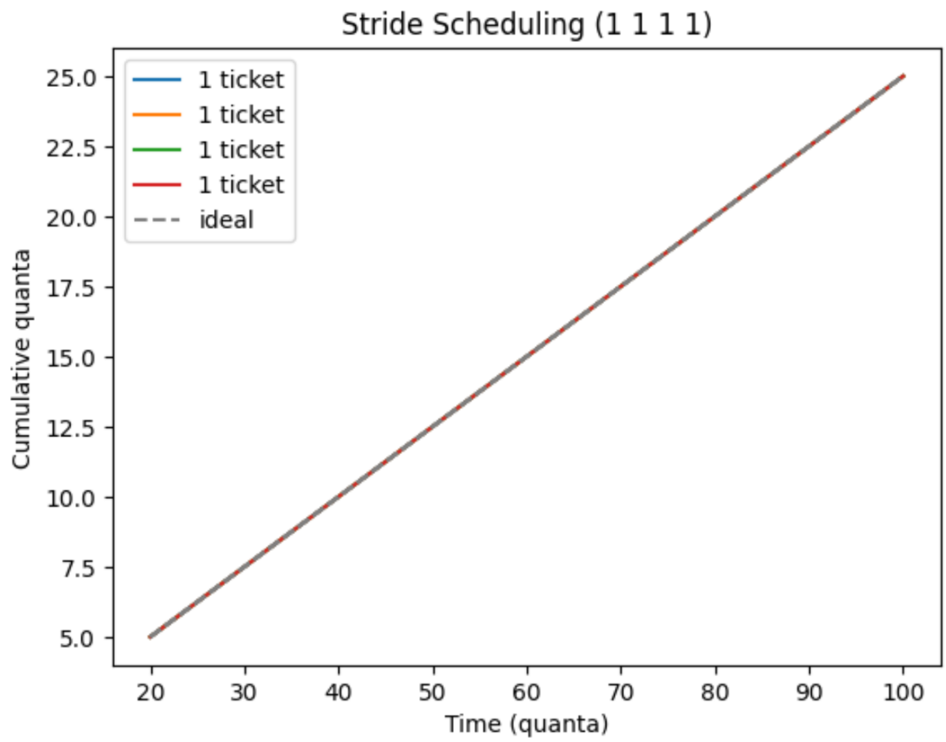


Figure 6: Equal ticket distribution demonstrating the deterministic nature of stride scheduling.

Each scheduler's test run was replicated five times to ensure the robustness of these observations. The collected data across these runs provides a reliable basis for assessing the schedulers' performance and fairness.

7 Contributions

- **Xiao Fan**: Focused on implementing part 1 system calls and also worked on the overleaf format of the report.
- **Kaya Gokalp**: Led the development of the lottery scheduler logic and contributed to the graphs in part 3.
- **Ashwin Nellimuttath**: Worked on the stride scheduler implementation and modifications in kernel/proc.h.
- **Hemanth Paladugu**: Managed the overall project coordination and contributed to the part 2 debugging and testing. Also worked on the report with Xiao.

8 Conclusion

The task of modifying the XV6 scheduler to include lottery and stride scheduling algorithms was a challenging yet rewarding experience. It provided us with practical insights into the complexities of operating system design and the intricacies of process scheduling. This project has significantly enhanced our understanding of operating systems and their core functionalities.