

Anthony Mendez | 862307065 | amend303

Jordan Kushner | 862294132 | jkusc002

Due Date: 05/18/23

Lab 2 - Modifying XV6 Scheduler

Video Demo Link: <https://youtu.be/e7qYz3HHkPY>

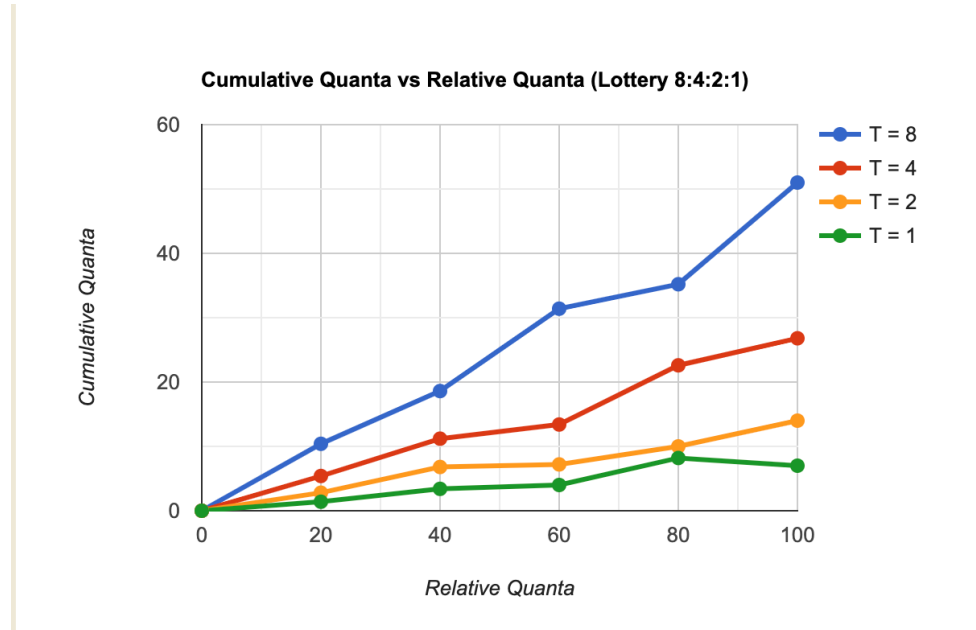
All Modified Files.....	1
Change Explanations and screenshots.....	2
Description of XV6 Source Code.....	11
Summary of Contributions.....	11

All Modified Files

- ~/kernel/ files
 - Defs.h
 - Proc.c
 - Proc.h
 - Syscall.c
 - Syscall.h
 - Sysproc.c
- ~/user/ files:
 - User.h
 - Usys.pl
 - lab2.c was created as a test files for the schedulers
- Makefile

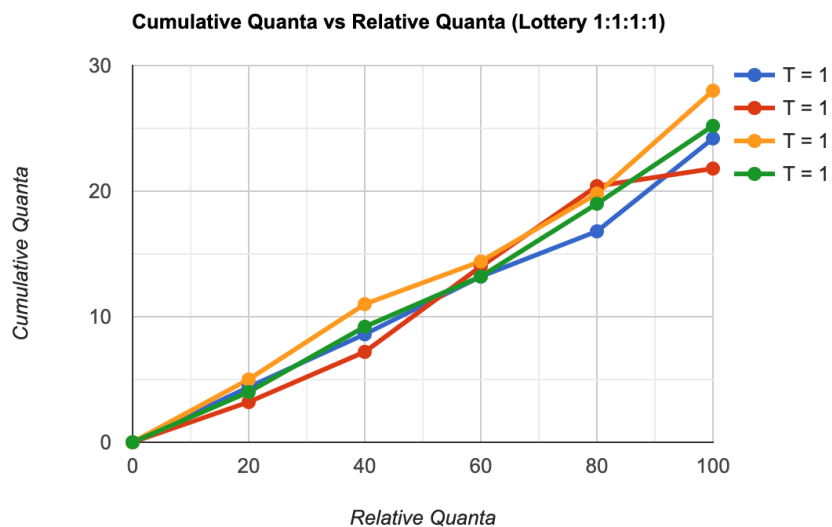
Change Explanations and screenshots

- Figures for Part 3
 - Lottery 8:4:2:1 Allocation



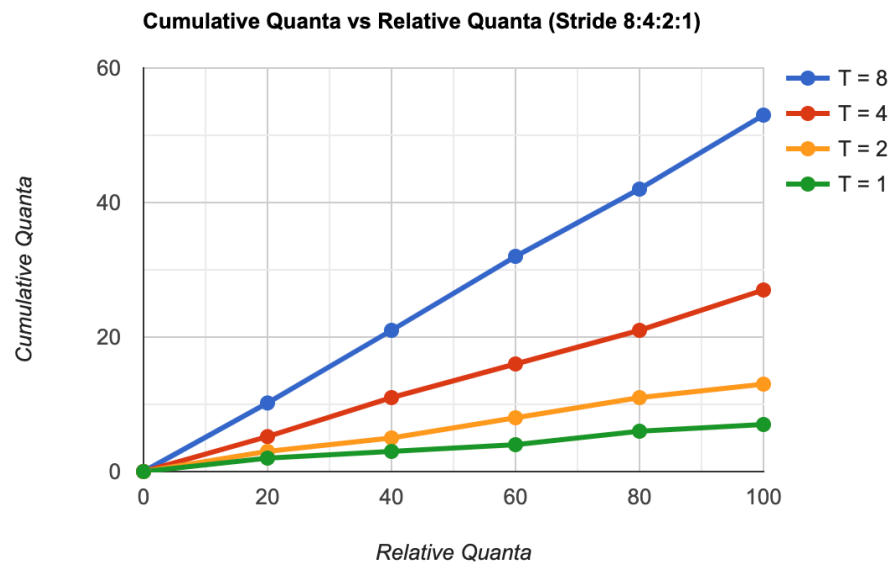
○

- Lottery 1:1:1:1 Allocation



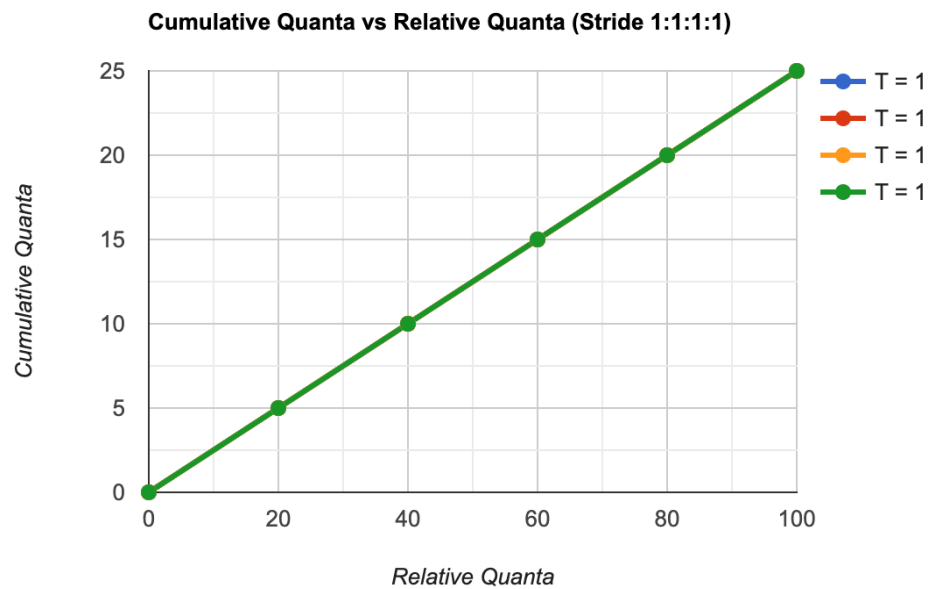
○

- Stride 8:4:2:1 Allocation



-

- Stride 1:1:1:1 Allocation



-

- The above four figures represent Part 3 of this lab demonstrating the number of ticks each process gets allocated based on their respective number of tickets. Simulations were run with both an 8:4:2:1 ticket allocation and a 1:1:1:1 allocation on step sizes of 20, 40, 60, 80, and 100 ticks. Each ticket allocation was tested 5 times at each step size and the average allocated tickets at each step size were plotted in the above charts. We can see that in the Lottery 8:4:2:1 allocation, the tick allocations rise somewhat linearly although they still fluctuate from the ideal allocation significantly, especially at smaller step sizes. Similarly, in the Lottery 1:1:1:1 allocation, the tick allocations fluctuate greatly at smaller step sizes but converge much closer to the ideal at larger step sizes. The Stride 8:4:2:1 allocation is nearly perfectly matching the ideal at each step size with a near linear increase for each process. This shows the deterministic strength of stride scheduling over lottery scheduling. This determinism is demonstrated again in the Stride 1:1:1:1 allocation where when allocated equal tickets, all 4 processes receive the same number of ticks at every step size.

- Defs.h

```
113 int sched_statistics(void); // sched_statistics
114 int sched_tickets(int); // sched_tickets
```

- Declaring syscall function headers in defs.h

- Proc.c

```
18 unsigned short rand()
19 {
20     bit = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5)) & 1;
21     return lfsr = (lfsr >> 1) | (bit << 15);
22 }
```

- Random number generator for lottery scheduling

```

167
168 // Initialize the tickets, stride, pass, and ticks
169 p->tickets = 10000;
170 p->stride = 10000 / p->tickets;
171 p->pass = p->stride;
172 p->ticks = 0;

```

- Initializing new proc variables for lottery/stride scheduler

```

498 struct proc *winner = 0;
499 int totalTickets = 0;
500 // Calculate the total number of tickets in the system
501 for (p = proc; p < &proc[NPROC]; ++p) {
502     acquire(&p->lock);
503     if (p->state == RUNNABLE) {
504         totalTickets += p->tickets;
505     }
506     release(&p->lock);
507 }
508
509 // Generate a random number between 0 and the total number of tickets
510 if (totalTickets > 0) {
511     int winningTicket = rand() % totalTickets;
512     int iteratedTickets = 0;
513
514     // Iterate through the processes to find the process
515     // with the winning ticket
516     for (p = proc; p < &proc[NPROC]; ++p) {
517         acquire(&p->lock);
518         if (p->state == RUNNABLE) {
519             iteratedTickets += p->tickets;
520             if (iteratedTickets > winningTicket) {
521                 winner = p;
522                 release(&p->lock);
523                 break;
524             }
525         }
526         release(&p->lock);
527     }
528 }
529
530 // If a process was chosen, run it
531 if (winner != 0 && winner->state == RUNNABLE) {
532     p = winner;
533     acquire(&p->lock);
534     p->ticks++;
535
536     p->state = RUNNING;
537     c->proc = p;
538     switch(&c->context, &p->context);
539     c->proc = 0;
540     release(&p->lock);
541 }

```

- Lottery Scheduler

```

568     struct proc *minimumPassProc = 0;
569     int minimumPass = 0x7FFFFFFF;
570
571     // Iterate through the processes to find the process
572     // with the smallest pass value
573     for (p = proc; p < &proc[NPROC]; ++p) {
574         acquire(&p->lock);
575         if (p->state == RUNNABLE && p->pass < minimumPass) {
576             minimumPassProc = p;
577             minimumPass = p->pass;
578         }
579         release(&p->lock);
580     }
581
582     // If a process was chosen, run it
583     if (minimumPassProc != 0 && minimumPassProc->state == RUNNABLE) {
584         p = minimumPassProc;
585         acquire(&p->lock);
586         p->ticks++;
587         p->pass += p->stride;
588         p->state = RUNNING;
589         c->proc = p;
590         swtch(&c->context, &p->context);
591         c->proc = 0;
592         release(&p->lock);
593     }
594
595     #else
596     /*

```

○

○ Stride Scheduler

```

int
sched_statistics(void)
{
    // Declare variables
    struct proc *p;

    // Print the process info
    for (p = proc; p < &proc[NPROC]; ++p) {
        acquire(&p->lock);

        if (p->state != UNUSED) {
            printf("%d|(%s): tickets: %d, ticks: %d\n", p->pid, p->name,
                p->tickets, p->ticks);
        }
        release(&p->lock);
    }

    return 0;
}

```

○

○ Sched_statistics() syscall implementation

```

int
sched_tickets(int tickets)
{
    // Declare variables
    struct proc *p = myproc();

    // Check if the input is valid
    if (tickets < 0 || tickets > 10000) {
        return -1;
    }

    // Set the number of tickets
    acquire(&p->lock);
    p->tickets = tickets;
    p->stride = 10000 / tickets;
    p->pass = p->stride;
    release(&p->lock);

    return 0;
}

```

-
- Sched_tickets() syscall implementation

- Proc.h

```

95     int tickets;           // Number of tickets for process
96     int stride;           // Stride of process
97     int pass;             // Pass of process
98     int ticks;            // Number of ticks process has run
99

```

-
- Defining new proc variables for lottery/stride scheduling

- Syscall.c

```

110     extern uint64 sys_sched_statistics(void); // sched_statistics: declaration
111     extern uint64 sys_sched_tickets(void); // sched_tickets: declaration
112
113     [SYS_sched_statistics] = sys_sched_statistics, // sched_statistics: syscall entry
140     [SYS_sched_tickets] = sys_sched_tickets, // sched_tickets: syscall entry
141
142 }

```

-
- Adding the new syscalls to the syscall entry table

- Syscall.h

```
26 #define SYS_sched_statistics 25 // sched_statistics
27 #define SYS_sched_tickets 26 // sched_tickets
28
```

-

- Defining new syscall numbers for the new syscalls

- Sysproc.c

```
121 // sched_statistics syscall definition
122 uint64
123 sys_sched_statistics(void)
124 {
125     return sched_statistics();
126 }
127
128 // sched_tickets syscall definition
129 uint64
130 sys_sched_tickets(void)
131 {
132     int n;
133     argint(0, &n);
134     return sched_tickets(n);
135 }
136
```

-

- Helper functions defined in syscall.c to execute the actual syscall

- User.h

```
29 int sched_statistics(void); // sched_statistics
30 int sched_tickets(int); // sched_tickets
31
```

-

- Declaring new syscalls in the user space

- Usys.pl

```
42 entry("sched_statistics"); # sched_statistics syscall for user
43 entry("sched_tickets"); # sched_tickets syscall for user
44
```

-

- Defining wrapper functions for new syscalls. These functions bridge the gap between user and kernel space

- Makefile

```
138      $U/_lab1_test\
139      $U/_lab2\
```

-
- Modified Makefile for lab2 test file

```
73      LAB2 = RR
74      CFLAGS += -D$(LAB2)
```

-
- Modified Makefile to handle Lottery and Stride scheduling (Round robin by default)

- Test file (lab2.c)

```
5 |
6 | #define MAX_PROC 10
7 | int main(int argc, char *argv[])
8 | {
9 |     int sleep_ticks, n_proc, ret, proc_pid[MAX_PROC];
10 |    if (argc < 4) {
11 |        printf("Usage: %s [SLEEP] [N_PROC] [TICKET1] [TICKET2]...\n", argv[0]);
12 |        exit(-1);
13 |    }
14 |    sleep_ticks = atoi(argv[1]);
15 |    n_proc = atoi(argv[2]);
16 |    if (n_proc > MAX_PROC) {
17 |        printf("Cannot test with more than %d processes\n", MAX_PROC);
18 |        exit(-1);
19 |    }
20 |    for (int i = 0; i < n_proc; i++) {
21 |        int n_tickets = atoi(argv[3+i]);
22 |        ret = fork();
23 |        if (ret == 0) // child process
24 |            sched_tickets(n_tickets);
25 |            while(1);
26 |        }
27 |        else { // parent
28 |            proc_pid[i] = ret;
29 |            continue;
30 |        }
31 |    }
32 |    sleep(sleep_ticks);
33 |    sched_statistics();
34 |
35 |    for (int i = 0; i < n_proc; i++) kill(proc_pid[i]);
36 |
37 |    exit(0);
38 | }
39 |
```

-

- Stride Test Output

```
init: starting sh
$ lab2 30 3 30 20 10
1|(init): tickets: 10000, ticks: 13
2|(sh): tickets: 10000, ticks: 13
3|(lab2): tickets: 10000, ticks: 34
4|(lab2): tickets: 30, ticks: 15
5|(lab2): tickets: 20, ticks: 10
6|(lab2): tickets: 10, ticks: 5
$
```

○

- Lottery Test Output

```
init: starting sh
$ lab2 30 3 30 20 10
1|(init): tickets: 10000, ticks: 25
2|(sh): tickets: 10000, ticks: 13
3|(lab2): tickets: 10000, ticks: 35
4|(lab2): tickets: 30, ticks: 15
5|(lab2): tickets: 20, ticks: 8
6|(lab2): tickets: 10, ticks: 7
$
```

○

Description of XV6 Source Code

By default, xv6 uses a round robin process scheduler in which processes are iterated over sequentially and scheduled for equal amounts of time. We implemented two new schedulers, lottery and stride schedulers, that provide a probabilistic and deterministic, respectively, way to prioritize and schedule some processes over others. Our test program iterates over each process passed in the command line and schedules them with the passed ticket allocation using the `sched_tickets()` syscall. The `sched_tickets()` syscall sets the passed value of tickets to the current process and computes stride and pass values as well. It then goes to the scheduler where, for lottery, a winning ticket value is selected and the first process to hold a ticket above that value gets scheduled and their ticks counted, while for stride, the process with the lowest pass value gets scheduled next and their ticks counted. This repeats for all processes. After iterating through all the processes and counting their ticks, the test file calls `sched_statistics()` to output the allocated tickets and actual number of allocated ticks for each process. To switch between each scheduler, one must run 'make clean; make LAB2=LOTTERY (or) make LAB2=STRIDE; make qemu'.

Summary of Contributions

- Anthony Mendez implemented the two syscalls, `sched_tickets()` and `sched_statistics()`. He also implemented the stride scheduler and assisted in debugging.
- Jordan Kushner implemented the lottery scheduler, generated the charts for part 3, assisted in debugging, and wrote the report.