Lottery Scheduler Report

Camille Williford - awillif4

October 19, 2021

1 Introduction

The point of this lab was to familiarize students with the xv6 simulator and the implementation of system calls and scheduler logic. I implemented two system calls and a lottery scheduling policy. I am going to take a chronological order to this report detailing changes that I made to the xv6 OS in the order that I made them. Here is a link to the change log in GitHub: Lottery Scheduler Commit

2 Data Changes

2.1 pstat.h

New File
I added this new structure *pstat*, as specified in the lab write up, to a header file.

2.2 proc.h

Modified: Added 2 Lines
I added integer properties *tickets* and *ticks* to the *proc* struct to persist ticket and tick information for the lottery scheduler. Processes with more tickets will run more often, and the ticks represent every time a process is run. Processes (also known as jobs) number of times run should be roughly proportional which will be evident with the later graph.

2.3 System Calls

2.4 syscall.c

Modified: Added 4 Lines
Defined two new system calls $sys\_settickets()$ and $sys\_getpinfo()$ and their respective system functions.

2.5 syscall.h

Modified: Added 2 Lines
Assigned numbers to the new system calls $sys\_settickets()$ and $sys\_getpinfo()$. These numbers are used when a user initiates a system call to safely determine which system call the user requested be performed.

2.6 user.h

Modified: Added 3 Lines
I defined a pstat struct, and prototyped the two new system calls as callable functions. These are the user functions that can be used to to call into the operating system.

$$int\ settickets(int);$$

$$int\ getpinfo(struct\ pstat*);$$

2.7 usys.S

Modified: Added 2 Lines
Assembly code that loads register with syscall number, makes syscall, and makes transition to the kernel mode using int instruction. I added the two new system calls to this list.

2.8 sysproc.c

Modified: Added 46 Lines
I implemented the two new system calls *sys_settickets*() and *sys_getpinfo*()! *sys_settickets*() is straightforward to implement. Simply get the user argument via the system utility *argint*(), and set this user integer as the number of tickets for the current process. If getting the user argument fails, or the user has passed in a negative number of tickets the system call will fail returning -1.

The *sys_getpinfo*() call is slightly more complex than the other system call. I had to make the ptable not specific to proc.c by defining the ptable as an extern variable. This enabled me to access the ptable in the system call function. The user argument is a pointer to a pstat struct that you have to populate in the system call. The first step is to get the user argument via the system utility *argptr*(), and assign it to a local pstat struct which means I had to include *pstat.h*. If an invalid pointer is passed in the call will fail and return -1 Then I loop over every process on the ptable aggregating process information into the pstat struct. For every process we store the pid, number of ticks, and number of tickets, and we determine if the given space on the ptable is "occupied" or not.

3 Miscellaneous Changes

3.1 proc.c

Modified: Added 30 Lines, Edited 3 Lines

- When a process is allocated in *allocproc*() set the new process's tickets to 1 and ticks to 0.
- When forking a new process in *fork*() set the child process's number of tickets to the parent process's number of tickets (i.e. the child process has the same number of tickets as the parent process).
- When exiting a process in *exit*() set the number of tickets to 0 so that the process will not be scheduled again.
- In the Makefile, I changed the OS to use only one CPU so that I can test the scheduler better.

4 Scheduler Changes

4.1 rand.h

New File
This file includes a method to generate a random number, *rand*(), and a method to seed the generator *srand*().

4.2 proc.c

Modified: Added 30 Lines, Edited 3 Lines
First, the scheduler makes sure that there is not a process currently running on the CPU, and seed the random number generator. Then we begin looping. In this indefinite loop, we begin by getting the total number tickets that belong to runnable processes by looping through the ptable. Then we generate a random "winning" lottery ticket limited by the total number of tickets that we will use to determine which process to run. Then we will loop through the processes again, but this time we will be incrementally counting how many tickets have been passed. When this count has passed the "winning" ticket, we switch to the current process. Then set that process on the CPU, and perform a context switch to the winning process. The ticks the process has taken are also incremented at this time.

5 Testing

5.1 grapher.c

New File
This test file creates 3 processes using fork, set number of tickets with $settickets()$, uses the $getpinfo()$ system call, and prints process info out over time. The table is to large to fit on this page, so you will find it on the next page.

Time	A	B	C
1	3	2	0
2	4	2	0
3	6	2	0
4	7	2	0
5	9	2	0
6	10	2	0
7	11	5	0
8	13	8	2
9	14	8	2
10	16	10	2
11	18	15	2
12	20	16	2
13	21	16	2
14	23	17	3
15	25	19	4
16	26	19	4
17	28	20	6
18	29	20	6
19	31	24	7
20	33	26	9
21	34	28	11
22	36	29	12
23	38	30	12
24	39	30	12
25	40	30	13
26	42	30	14
27	44	30	14
28	46	31	14
29	47	35	15
30	48	36	15
31	50	36	15
32	52	37	15
33	53	37	15
34	54	37	15
35	56	38	15
36	57	38	15
37	59	43	15
38	61	44	15
39	62	45	16
40	64	45	16
41	66	49	18
42	68	49	18
43	69	50	18
44	71	50	18
45	72	50	18
46	74	51	19
47	75	51	19
48	77	53	19
49	79	53	20
50	80	54	20

Table 1: Ticks accumulated over 50 seconds