Project 2-The xv6 Lottery Scheduler

In this project, you'll be putting a new scheduler into xv6. It is called a **lottery scheduler**, and the full version is described in this chapter of the online book; you'll be building a simpler one. The basic idea is simple: assign each running process a slice of the processor based in proportion to the number of tickets it has; the more tickets a process has, the more it runs. Each time slice, a randomized lottery determines the winner of the lottery; that winning process is the one that runs for that time slice.

The objectives for this project:

- To gain further knowledge of a real kernel, xv6.
- To familiarize yourself with a scheduler.
- To change that scheduler to a new algorithm.
- To make a graph to show your project behaves appropriately.

Details

You'll need a new system call **`int settickets(int number)`** to implement this scheduler, which can set the number of tickets of the calling process. By default, each process should get 10 tickets; calling this routine makes it such that a process can raise the number of tickets it receives, and thus receive a higher proportion of CPU cycles. This routine should return 0 if successful, and -1 otherwise (if, for example, the caller passes in a number less than one).

To make sure that **`int settickets(int number)`** works properly, each struct **`proc`** needs an additional field, **`tickets`**, that tracks how many tickets it has. New processes are assigned 10 lottery tickets when they are created. When the scheduler runs, it picks a random number between 0 and the total number of tickets of the runnable processes. It then uses the algorithm described in class to loop over runnable processes and pick the one with the winning ticket.

You'll need a pseudo-random number generator to create random numbers. In the starter code, **`rand.h`** and **`rand.c`** are provided for you to use as the pseudo-random number generator. To use it, you need to include **`rand.o\`** in **`OBJS`** in Makefile and copy files **`rand.h`** and **`rand.c`** in your xv6 directory. After doing these two steps you can include **`rand.h`** header file in your **`proc.c`** file which is responsible for scheduling all processes. By calling **`long random_at_most(long max)`**, the method would return a random number between 0 and max.

Most of the code for the scheduler is quite localized and can be found in **`proc.c`**; the associated header file, **`proc.h`** is also quite useful to examine. To change the

scheduler, not much needs to be done; study its control flow and then try some small changes.

Testing

CPUS = 1

(Note that the xv6 should be configured to run on a single CPU by updating **`CPUS: = 1`** in the Makefile, which would prevent multiple processes run parallely.)

A testing program **`lotterytest.c`** is provided for you to test if your lottery scheduler works properly or not. In this testing program, another new system call **`int getpinfo(struct pstat *)`** is needed. This routine returns some information about all running processes, including how many times each has been chosen to run and the process ID of each, and save them in the **`pstat`** structure in the parameters list. You can use this system call to build a variant of the command line program **`ps`**, which can then be called to see what is going on. In xv6 **`void procdump(void)`** is the default utility used to print some processes information. You can find it in **`proc.c`** and invoke it by pressing **`ctrl+p`**. The structure **`pstat`** is defined below; note, you cannot change this structure, and must use it exactly as is. This routine should return 0 if successful, and -1 otherwise (if, for example, a bad or NULL pointer is passed into the kernel).

To make sure that **`int getpinfo(struct pstat *)`** works properly, each struct **`proc`** needs an additional field, **`ticks`**, that tracks how many ticks it has. New processes have 0 ticks when they are created. Since the scheduler would choose one process to run after each time interrupt, each time a process is chosen it would run for a tick.

Also, you'll need to understand how to fill in the structure **`pstat`** in the kernel and pass the results to user space. The structure should look like what you see here, in a file you'll have to include called **`pstat.h`**, which is also provided in the starter code:

```
#ifndef _PSTAT_H_
#define _PSTAT_H_

#include "param.h"

struct pstat {
  int inuse[NPROC];  // whether this slot of the process table is in use (1 or 0)
  int tickets[NPROC]; // the number of tickets this process has
  int pid[NPROC];     // the PID of each process
  int ticks[NPROC];   // the number of ticks each process has accumulated
};
```

```
#endif // _PSTAT_H_
```

Good examples of how to pass arguments into the kernel are found in existing system calls. In particular, follow the path of **`read()`**, which will lead you to **`sys_read()`**, which will show you how to use **`argptr()`** (and related calls) to obtain a pointer that has been passed into the kernel. Note how careful the kernel is with pointers passed from user space -- they are a security threat(!), and thus must be checked very carefully before usage.

Test file

The testing program **`lotterytest.c`** is shown below and provide in the starter code as well. The **`main()`** method (ie., the parent process) creates three child processes with 100, 200, and 300 tickets respectively. Each child call **`spin()`** which would run an infinite computing loop. Meanwhile, the parent would periodically call **`int getpinfo(struct pstat *)`** to print all runnable processes information. Eventually, after five periods, the parent would kill all children and terminate itself.

Test file

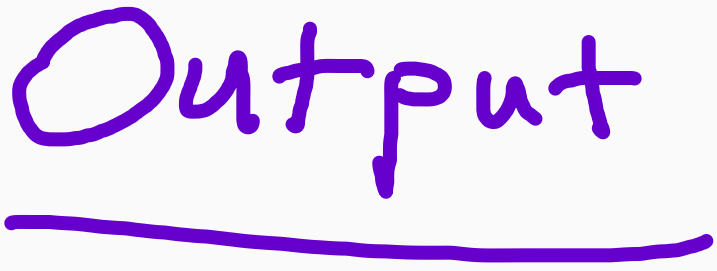
```c
#include "syscall.h"
#include "types.h"
#include "user.h"
#include "pstat.h"
#include "param.h"

void spin();
int main(int argc, char *argv[]){
	int tickets[] = {100, 200, 300};
	int pids[] = {0, 0, 0};
	struct pstat info;
	int i, rc;
	// create three children
	for(i = 0; i < 3; i++){
		rc = fork();
		if(rc == 0) {
			settickets(tickets[i]);
			spin();
			exit();
		} else {
			pids[i] = rc;
		}
	}
	// prevent parent from using too much CPU
	settickets(10);
	// periodically print process information
	for(i = 0; i < 5; i++){
```

```
                sleep(500);
                if(getpinfo(&info) < 0)
                        exit();
                printf(1, "PID\tTICKETS\tTICKS\n");
                for (int i = 0; i < NPROC; ++i) {
                        if(info.inuse[i] == 1){
        printf(1, "%d\t%d\t%d\n", info.pid[i], info.tickets[i], info.ticks[i]);
                        }
                }
        }
        // kill all children
        for (int i = 0; i < 3; ++i) {
                kill(pids[i]);
        }
        for (int i = 0; i < 3; ++i) {
                wait();
        }
        exit();
}
// infinite computing loop
void spin(){
        volatile int sink = 0;
        for(;;){
                sink = sink + 1;
                sink = sink + 1;
                sink = sink + 1;
        }
}
```

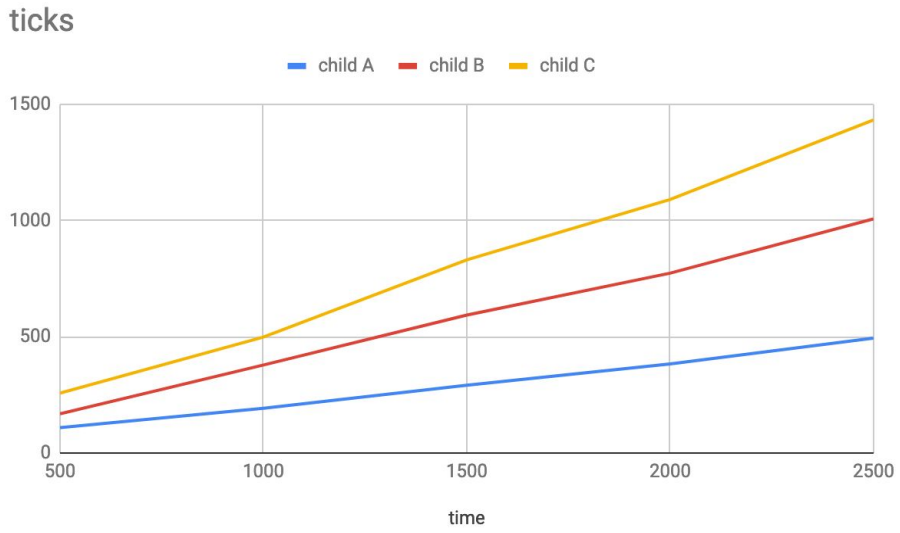
The testing program should produce something similar to the following output. The **lotterytest** executable (when booted into xv6) is based on the program above. Because the program depends on your implementation, however, it isn't compiled into xv6 by default. First you should copy **lotterytest.c** in xv6 folder, and modify the **Makefile** to add **lotterytest** as a new user program.

```
$ lotterytest
PID     TICKETS TICKS
1       10      25
2       10      17
3       10      19
4       100     109
5       200     168
6       300     257
PID     TICKETS TICKS
1       10      25
2       10      17
```



```
3       10      30
4       100     192
5       200     378
6       300     498
PID     TICKETS TICKS
1       10      25
2       10      17
3       10      47
4       100     291
5       200     593
6       300     830
PID     TICKETS TICKS
1       10      25
2       10      17
3       10      58
4       100     383
5       200     773
6       300     1090
PID     TICKETS TICKS
1       10      25
2       10      17
3       10      74
4       100     494
5       200     1006
6       300     1432
```

Beyond the usual code, you'll have to make a graph for this assignment based on the output of the testing program. The graph should show the number of time slices a set of three processes receives over time, where the processes have a 3:2:1 ratio of tickets (i.e., process A has 100 tickets, process B 200, and process C 300), which is implemented in **`lotterytest.c`**. The graph should clearly show that your lottery scheduler works as desired. The graph may look like below.



Order

1. Implement **`int settickets(int number)`**, but don't actually use ticket counts for anything.
2. Implement **`int getpinfo(struct pstat *)`**. Use the **`lotterytest.c`** or other programs (e.g. ones you write, such as **`ps`**) based on it to verify that it works.
3. Add tracking of the number of ticks a process runs. Use the **`lotterytest.c`** or other programs based on it to verify that it works. Note that with round-robin scheduling, if multiple processes are CPU-bound, then each process should run for approximately the same amount of time.
4. Implement the lottery scheduling algorithm. Use the **`lotterytest.c`** to test it.

Hints

1. The current **`scheduler()`** function found in **`proc.c`** has two for loops, one inside the other. The outer for loop just keeps on running forever. The inside for loop loops over all the processes and selects the first process that it finds in a **`RUNNABLE`** state. It then runs this process till its time quanta expire or the process yields voluntarily. It then selects the next process in **`RUNNABLE`** state and so on. This is the way it implements **Round Robin**.
2. We need to do the following things to implement lottery scheduling in XV6: Make a system call which allows you to set the tickets for a process. Code to generate a random number. In the scheduler function count the total number of tickets for all processes that are runnable. Generate a random number between 0 and the total tickets calculated above. When looping through the processes keep the counter of the total number of tickets passed. Just when

Total_tickets()

the counter becomes greater the random value we got, chose and run the process.

3. One possible way to track the total number of tickets is by carefully finding each place where the process state changes, and adding or subtracting tickets from a global as appropriate. However, this is actually rather tricky to get right. An alternative is to simply compute the number of tickets each time we enter the scheduler loop, before choosing the winning ticket.
4. Right now, the scheduler loop picks up where it left off after scheduling a process. So if it finds a runnable process at index i, the next time the scheduler runs it will keep going through the list at index i+1 . In order to make the lottery algorithm work, you will need to restructure the scheduler loop so that it starts back at the beginning of the list every time we return to the scheduler. For example, you can put a break at the end of for loop so that we don't execute the processes following the process we just run.

GOAL

5. Adding **`settickets`** - You can use **`argint`** to retrieve the integer argument to your system call. Adding **`getpinfo`** - You can use the **`argptr`** to retrieve the pointer argument in your system call handler; You should iterate through the process list **`ptable`**, if one process is **`UNUSED`** processes, then the **`inuse`** field in **`struct pstat`** would be 0; Before and after accessing the process table (**`ptable`**), you should acquire **`ptable.lock`** and afterwards you should release it. You can see an example of this in **`kill`** in **`proc.c`**. This will keep you from running into problems if a process is removed while you are iterating through the process table.
6. Since your system call **`getpinfo`** in **`sysproc.c`** or **`sysfile.c`** can **NOT** directly access the **`ptable`** in **`proc.c`**, you may call a method which is defined in **`proc.c`** and declared in **`defs.h`**. This method resides in **`proc.c`** so it can access **`ptable`**, it should pass the pointer of **`struct pstat`** and initialize it with the content of **`ptable`**. You may look at **`sys_read`** in **`sysfile.c`**, **`fileread`** in **`file.c`** and **`readi`** in **`fs.c`** to see how to pass pointer and look at **`sys_exit`** in **`sysproc.c`** and **`exit`** in **`proc.c`** to see how to access **`ptable`** from outside of **`proc.c`**.

getmypinfo()

GDB – debugger

1. When you are facing bugs in your kernel, in general you can rely on the debugging tools like **gdb** to locate the problems. Or you can just **print** some information to help you understand the execution of the code. Also, another good practice is to test your code cumulatively. Instead of finishing all the code and testing it, you should do the testing right after you implement some components, so that you would be able to quickly locate the bugs, sometime this is called **unit testing**.

2. The use of **gdb** to debug your code is highly encouraged. A nice article of *Remote Debugging xv6 under QEMU* can be used as a reference. Also, you can watch this Youtube video as a tutorial.
3. Here are some tips to identify **panics**:
   A. If xv6 prints a message containing something like **`panic: acquire`**, this means that something called **`panic("acquire");`**. The **`panic()`** function stops the OS, printing out an error message. Generally, these panics are caused by **`assertions`** in the xv6 code, checking the current state is consistent.
   B. Most xv6 panic messages include the name of the function that called **`panic`**, so you can often search for that function name and see when it called **`panic`**. In general, you can get **`grep`** the xv6 code to find out what exactly the cause was. For example, **`panic("acquire");`** appears in **`acquire()`** in **`spinlock.c`**. It is called if a thread tries to acquire a spinlock that the current thread already holds.
   C. There is a panic in the **`trap`** function that triggers when an unexpected trap occurs in kernel mode. You can infer more about those from the table of of trap numbers in **`traps.h`** and the message which is printed out before the **`panic`**. One of the possible traps is a **`page fault`**, which means you accessed a memory location that was invalid. This can be caused by using an invalid pointer, going out of bounds of an array, or using more space on the stack than is allocated.

End & Submission

Upload all of your source files (but not .o files, please, or binaries!) as a zip file **`xv6-public.zip`** to Blackboard. A simple way to do this is type **`make`** in your xv6-public folder to make sure it builds, and then type **`make clean`** to remove unneeded files. Generate a **`patch`** for you kernel and upload it. Also, please make a **`SCREENSHOT`** file, which shows your output of **`lotterytest`**. Finally, please submit the **`graph`** as a PDF file to demonstrate how your scheduler works.

Creating a patch using diff

A patch file is a text file which contains the differences between two versions of the same file (or same directory). Patch file is created by using diff command. To create a patch of your xv6 version, we need to have an original version of xv6 download and run the following commands in the terminal:

```
# clean the object files
cd xv6-project2
```

```
make clean

# go back to upper level folder
cd ..

# xv6-public is the original version
diff -Naur xv6-public xv6-project2 > patch.txt
```

diff is the a linux command line program, and is where patch files originated. It requires that you have two copies of the code, one with your changes, and one without. More details can be found in this tutorial.