P3: Scheduling

Due Jul 5 by 11:59pm **Points** 120 **Available** Jun 18 at 12am - Jul 8 at 11:59pm 21 days

This assignment was locked Jul 8 at 11:59pm.

Objectives

- To understand existing code for performing context-switches in the xv6 kernel
- To implement a basic round-robin (RR) scheduler that compensates processes that relinquish the CPU
- To implement system calls that extract process states
- To implement a user-level program that tests the basic compensation behavior of the scheduler

Overview

In this project, you'll be modifying the Round Robin (RR) CPU scheduler in xv6 so that processes can have different time-slice lengths and are "compensated" in different ways for the amount of time they are blocked (and thus cannot be scheduled).

You will be using the current version of xv6. You could find a copy of xv6 source code in `/p/course/cs537-yuvraj/public/xv6.tar.gz`. Copy it into your private directory and untar it.

```
prompt> cp /p/course/cs537-yuvraj/public/xv6.tar.gz /path/to/your/private/dir
prompt> tar -xvzf xv6.tar.gz
```

Particularly useful for this project: **Chapter 5 (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** in xv6 book.

In this document we:

1. specify how your new xv6 scheduler (CRR) must behave
2. specify the new system calls you must add
3. specify the simple user-level application you must write
4. describe the existing xv6 scheduling implementation

5. give suggestions for implementing this project
6. specify other requirements for completing this project

1) Compensated Round-Robin (CRR) Scheduler Requirements

The current xv6 scheduler implements a very simple Round Robin (RR) policy. For example, if there are three processes A, B and C, then the xv6 round-robin scheduler will run the jobs in the order A B C A B C … , where each letter represents a process. The xv6 scheduler runs each process for at most one timer tick (10 ms); after each timer tick, the scheduler moves the previous job to the end of the ready queue and dispatches the next job in the list. The xv6 scheduler does not do anything special when a process sleeps or blocks (and is unable to be scheduled); the blocked job is simply skipped until it is ready and it is again its turn in the RR order.

You will implement a new Compensated RR scheduler with three new features:

1. Different time-slice lengths (i.e., a different number of timer ticks) for different processes
2. Compensating processes for the amount of time they were blocked by scheduling those processes for a longer time-slice when they awake
3. Improving the sleep/wakeup mechanism so that processes are unblocked only after their sleep interval has expired, instead of on every 10 ms timer tick

1.1. Basic Time-Slices

In your CRR scheduler, each process will have the same priority, but each process can have a **time-slice** of a different length, specified by the number of **timer-ticks**. After a process consumes its time-slice it should be moved to the back of the queue.

For example, if a process has a time-slice of 8 timer ticks, then you should schedule this process for 8 ticks before moving to the next process. Another example: if there are 3 active processes, where A has a time-slice of 4 ticks, B has a time-slice of 3 ticks, and C has a time-slice of 2 ticks, the processes should be scheduled in this pattern: AAAABBBCCAAAABBBCC...

When a new process arrives, it should inherit the time-slice of its parent process and be added to the **tail** of the queue. **The first user process should start with a time-slice of 1 timer tick.** You will also create a system call, `setslice(int pid, int slice)`, to change the length of the specified process's time-slice.

Scheduling is relatively easy when processes are just using the CPU; scheduling gets more interesting when jobs are arriving and exiting, or performing I/O.

There are three events in xv6 that may cause a new process to be scheduled:

- whenever the xv6 10 ms timer tick occurs
- when a process exits
- when a process sleeps.

For simplicity, your scheduler should also not trigger a new scheduling event when a new process arrives or wakes; you should simply mark the process as "RUNNABLE" and move it to the tail of the queue. The next scheduling decision will be made the next time a timer tick occurs.

When a timer tick occurs, whichever process was using the CPU should be considered to have used an entire timer tick's worth of CPU, even if it did not start at the previous tick. (Remember that a timer tick is different than the time-slice.)

In the base case for your scheduler (which will be modified in the text below), when a process sleeps and then wakes, its time-slice should be reset. For example, consider a process A that has a time-slice of 3 ticks, uses up 2 of those ticks and then blocks; when process A wakes and is scheduled the next time, A should be scheduled again for the full 3 ticks.

1.2 Compensating Processes for Blocking

Many schedulers contain some type of incentive for processes with no useful work to sleep (or block) and voluntarily relinquish the CPU so another process can run; this improves overall system throughput since the process isn't wastefully holding on to the CPU, doing nothing. To provide this incentive, your scheduler will track how long a process sleeps and extend its next time-slice by a function of this amount.

Example 1: **Base case.** We'll start with the simplest case: a process A sleeps, wakes, is given compensation ticks, and A uses those compensation ticks the next time it is scheduled.

For example, if process A has a time-slice of 2 ticks and process B of 3 ticks and A sleeps for **3** timer ticks (at time 7), then when A wakes (at time 10), A is compensated for that sleep time by increasing its next time-slice by those 3 timer ticks; thus, at time 10, A is scheduled for a total of 5 ticks (2 ticks in its standard time-slice + 3 compensation ticks). In future intervals, A's time-slice reverts back to its base 2 ticks. When a process uses compensation ticks, those ticks are marked in red; **bold** indicates the first tick of a time-slice.

timer tick	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
scheduled	**A**	A	**B**	B	B	**A**	**B**	B	B	**A**	A	A	A	A	**B**	B	B	**A**	A	**B**

sleeping							A	A	A										

There are a few special cases to consider.

Example 2: Proper Sleep accounting. Make sure that when a process sleeps, it is given compensation ticks for the amount of time it was actually sleeping, not the amount of time it wasn't scheduled.

For example, what happens if A sleeps for a very short time, but B has a very long time-slice? We don't want A to be able to game the scheduler and get a large compensation time when A wasn't going to be scheduled in that interval anyways; we want to compensate A only for the time it was actually sleeping.

If we modify our previous example so A sleeps for only 1 timer tick, then when B's time-slice expires and A is rescheduled at time 10, it is given only 1 compensation tick.

timer tick	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
scheduled	**A**	A	**B**	B	B	**A**	**B**	B	B	**A**	A	A	**B**	B	B	**A**	A	**B**	B	B
sleeping							A													

Example 3: No accumulation of compensation ticks. What happens if A doesn't use its compensation ticks before it sleeps again? We don't want A to be able to accumulate a huge number of compensation ticks for future use.

If we modify the first example (where A blocks for 3 ticks and thus acquires 3 compensation ticks), but A blocks again at time 13 after using only 1 of those compensation tickets, A will lose the 2 compensation ticks it did not use; however, A will obtain new compensation ticks based on how long it blocks the next time (1 compensation tick acquired at time 13 and used at time 18 in this example).

interval	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
scheduled	**A**	A	**B**	B	B	**A**	**B**	B	B	**A**	A	A	**B**	B	B	**A**	A	A	**B**
sleeping							A	A	A				A						

Example 4: No Ready Processes. Finally, what should your scheduler do when there are no active processes to schedule? Blocking processes still acquire compensation ticks when they block, there just won't be another active process to schedule. This shouldn't require much special handling.

Consider example 1 without B:

interval	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
scheduled	**A**	A	**A**				**A**	A	A	A	A	**A**	A	**A**	A	**A**	A	**A**	A	**A**
sleeping				A	A	A														

When A sleeps from intervals 4 - 6 it will acquire 3 compensation ticks that are then used from 9-11. The boldface **A'** s designate that this is considered a new scheduling **time-slice** for A.

Example 5: Multiple Processes. In many workloads, there will be multiple processes that are blocked. All of these processes will be acquiring compensation ticks for the time they are blocked. This isn't a special case; just ensure that you aren't assuming only one process is sleeping or blocked at a time.

These mechanisms are far from perfect to prevent gaming, but they are good enough for this project. You could imagine many different policies that would fix more problems but might be more complicated.

1.3 Improving the Sleep/Wake Mechanism

Finally, you need to change the existing implementation of `sleep()` syscall. The sleep syscall allows processes to sleep for a specified number of ticks. Unfortunately, the current xv6 implementation of the `sleep()` syscall forces the sleeping process to wake on every timer tick to check if it has slept for the requested number of timer ticks or not. This has two drawbacks. First, it is inefficient to schedule every sleeping process and force it to perform this check. Second, and more importantly for this project, if the process was scheduled on a timer tick (even just to perform this check), your scheduler might not properly identify that process as sleeping and grant it compensation ticks.

Additional details about the xv6 code are given below. You are required to fix this problem of extra wakeups by changing `wakeup1()` in `proc.c`. You are likely to add additional condition checking to avoid falsely waking up the sleeping process until it is the right time. You may want to add more fields to `struct proc` to help `wakeup1()` decide whether if it is time to wake a process. You may find the section "sleep and wakeup" in the **xv6 book (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** (starting from page 65) are very helpful.

2) New system calls

You'll need to create several new system calls for this project.

1. `int setslice(int pid, int slice)`. This sets the time-slice of the specified `pid` to `slice`. You should check that both `pid` and `slice` are valid (`slice` must be > 0); if they are not, return -1. On successful change, return 0. The time-slice of a process could be increased, decreased, or not changed; if `pid` is the currently running process, then its time-slice should be immediately changed and applied to this scheduling interval. If the process has run for the number ticks it should run (or more) according to the new slice value (e.g. it has run 6 ticks, but the new time slice value is 4 ticks), you should schedule the next process when the timer interrupt fires.
2. `int getslice(int pid)`. This returns the time slice of the process with `pid`, which must be a positive interger. If the `pid` is not valid, it returns -1.
3. `int fork2(int slice)`. This routine is exactly like the existing `fork()` system call, except the newly created process, which should begin with the specified time-slice length. Thus, `fork()` could now be implemented as `fork2(getslice(getpid()))` since by default the child process inherits the time-slice length of the parent process. If `slice` is not positive, then `fork2()` should return -1.
4. `int getpinfo(struct pstat *)`. Because your scheduler is all in the kernel, you need to extract useful information for each process by creating this system call so as to better test whether your implementation works as expected.
   To be more specific, this system call returns 0 on success and -1 on failure (e.g., the argument is not a valid pointer). If success, some basic information about each process will be returned:

```
struct pstat {
  int inuse[NPROC]; // whether this slot of the process table is in use (1 or 0)
  int pid[NPROC]; // PID of each process
  int timeslice[NPROC]; // number of base ticks this process can run in a timeslice
  int compticks[NPROC]; // number of compensation ticks this process has used
  int schedticks[NPROC];  // total number of timer ticks this process has been scheduled
  int sleepticks[NPROC]; // number of ticks during which this process was blocked
  int switches[NPROC];  // total num times this process has been scheduled
};
```

Most of the fields should be self-explanatory, but we describe some in more detail. `compticks`, `schedticks`, `sleepticks`, and `switches` are accumulative, so they will increase during the lifecycle of the process and only get reset when the process dies.

The `schedticks` field is the total number of timer ticks the process has been scheduled; that is, ticks should usually be incremented exactly once for exactly one process when a timer tick occurs.

The `switches` field should be incremented whenever a process is scheduled (it should **not** be incremented for each timer-tick within that time-slice); the switches field should be incremented even if there is only one process ready process (e.g., if process A finishes its time-slice, but because there are no other ready processes, A is scheduled again, then switches should still be incremented). Switches should

also be incremented when a process is scheduled after waking (even if its previous time-slice was not used up entirely); it should not be incremented separately for compensation ticks. (Remember switches are marked in bold in the examples above.)

For example, if process A in example 3 calls `getpinfo` in interval 15, it should get:

```
A: timeslice = 2; compticks = 1; schedticks = 6; sleepticks = 4; switches = 3.
B: timeslice = 3; compticks = 0; schedticks = 9; sleepticks = 0; switches = 3.
```

You can decide if you want to update your `pstat` statistics whenever a change occurs, or if you have an equivalent copy of these statistics in `ptable` and want to copy out information from the `ptable` when `getpinfo()` is called.

The file should be copied from `/p/course/cs537-yuvraj/public/scheduler/pstat.h`.

Do not change the names of the fields in `pstat.h`. You may want to add a header guard to `pstat.h` and `param.h` to avoid redefinition.

3) New User-Level Applications

To demonstrate that your scheduler is doing at least some of the right things, you will create two new user-level applications named loop and schedtest.

First, to have a dummy job with known behavior, you will need to implement a user-level program, `loop`. It takes exactly 1 argument `sleepT`, and does the following:

- First, sleep for `sleepT` ticks;
- Then, call a loop like this which loops on a huge workload (don't try to code and run any real programs like this! It is just for testing purpose of this project):

```
int i = 0, j = 0;
while (i < 800000000) {
  j += i * j + 1;
  i++;
}
```

Second, `schedtest` runs two copies of **loop** as child processes, controlling the time-slice and sleep time of each child. The schedtest application takes exactly 5 arguments in the following order:

```
prompt> schedtest sliceA sleepA sliceB sleepB sleepParent
```

- `schedtest` spawns two children processes, each running the `loop` application. One child A is given initial timeslice length of `sliceA` and runs `loop sleepA`; the other B is given initial timeslice length of `sliceB` and runs `loop sleepB`.
- Specifically, the parent process calls `fork2()` and `exec()` for the two children `loop` processes, **A before B**, with the specified initial timeslice;
- The parent `schedtest` process then sleeps for `sleepParent` ticks by calling `sleep(sleepParent)` (you may assume that `sleepParent` is much larger than sliceA+2*sleepA+sliceB+2*sleepB);
- After sleeping, the parent calls `getpinfo()`, and prints one line of two numbers separated by a space: `printf(1, "%d %d\n", compticksA, compticksB)`, where `compticksA` is the `compticks` of process A in the `pstat` structure and similarly for B.
- The parent then waits for the two `loop` processes by calling `wait()` twice, and exits.

(Note that a time tick of 10ms is a lot of time for a fast CPU, hence it is likely that the calls to fork and exec will finish and the parent will call `wait` before any child process is scheduled; therefore, you don't need to worry about the parent process interfering with the scheduling queue.)

Once you have these two user applications, you will be able to run `schedtest` to test the basic compensation behavior of your scheduler. For example:

```
prompt> schedtest 2 3 5 5 100
3 5   # Expected output -- you should be able to figure out why if you have understood our RR compensation policy
```

Assume that all the arguments are reasonable and small positive numbers like the example above. We will not test your implementation of `schedtest` extensively; the purpose of it is primarily for you to understand how to test your round-robin scheduler. You are welcome to write more user applications to test other aspects of the scheduler: round-robin behavior, different time-slice lengths, process setting timeslice value of itself, etc.

4) xv6 Scheduling Details

Before implementing anything, you should have an understanding of how the current scheduler works in xv6. Particularly useful for this project: **Chapter 5 (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** in the xv6 book and the section "sleep and wakeup" in the **xv6 book (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)** (starting from page 65).

Most of the code for the scheduler is localized and can be found in `proc.c`. The header file `proc.h` describes the fields of an important structure, each of `struct proc` in the global `ptable`.

You should first look at the routine `scheduler()` which is essentially looping forever. After it grabs an exclusive lock over ptable, it then ensures that each of the RUNNABLE processes in the ptable is scheduled for a timer tick. Thus, the scheduler is implementing a simple Round Robin (RR) policy.

A timer tick is about 10ms, and this timer tick is equivalent to a single iteration of the **for loop** over the ptable in the `scheduler()` code. Why 10ms? This is based on the timer interrupt frequency setup in xv6. You should examine the relevant code for incrementing the value of **ticks**in `trap.c`.

When does the current scheduler make a scheduling decision? Basically, whenever a thread calls `sched()`. You'll see that `sched()` switches between the current context back to the context for `scheduler()`; the actual context switching code for `swtch()` is written in assembly and you can ignore those details.

When does a thread call `sched()`? You'll find three places: when a process exits, when a process sleeps, and during `yield()`. When a process exits and when a process sleeps are intuitive, but when is `yield()` called? You'll want to look at `trap.c` to see how the timer interrupt is handled and control is given to scheduling code; you may or may not want to change the code in `trap()`.

Other important routines that you may need to modify include `allocproc()` and `userinit()`. Of course, you may modify other routines as well.

For sleep and wake-up, the current xv6 implementation of the `sleep()` syscall has the following control flow:

- First, it puts the process into a SLEEPING state and marks its *wait channel* (field `chan` in `struct proc`) as the address of the global variable `ticks`. This operation means, this process "waits on" a change of `ticks`. "Channel" here is a waiting mechanism which could be any address. When process A updates a data structure and expects that some other processes could be waiting on a change of this data structure, A can scan and check other SLEEPING processes' `chan;` if process B's `chan` holds the address of the data structure process A just updated, then A would wake B up.
- Every time a tick occurs and the global variable `ticks` is incremented (in `trap.c` when handling a timer interrupt), the interrupt handler scans through the ptable and wakes up every blocked process waiting on `ticks`. This causes every sleeping process to get falsely scheduled.
- When each sleeping process is scheduled but finds it hasn't slept long enough, it puts itself back to sleep again (see the while-loop of `sys_sleep()`).

As mentioned previously, this implementation causes the process that invoked syscall `sleep()` to falsely wake inside the kernel on each tick, and as a result, the process won't get the compensation it deserves. To address this problem, you should change `wakeup1()` in `proc.c`

to have some additional condition checking to avoid falsely waking up the sleeping process (e.g. checking whether `chan == &ticks`, and whether it is the right time to wake up, etc). Feel free to add more fields to `struct proc` to help `wakeup1()` decides whether the processing it is going to wake up really needs to be wakened up at this moment .

5) Tips

We recommend writing very small amounts of code and testing each change you make (e.g., by running `forktest`) rather than implementing too much functionality at one time.

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 (and modify). You will also want to look at `trap.c` to see how the timer interrupt is handled and control is given to scheduling code; you may or may not want to change the code in `trap()`. To change the scheduler, not too much needs to be done; study its control flow and then try some small changes. To handle sleeps correctly, you way also want to look at and add a few lines in `sys_sleep()` in `sysproc.c`.

Here are some hints on where you may want to add code/which functions you may want to change and the order:

- In `proc.c`, add a queue-like data structure to hold the current scheduling queue. A linked list might be the easiest to implement, though other data structures work as well. It is much easier to deal with fixed-sized arrays in xv6 than allocating kernel memory. It is fine to continue to assume that xv6 can handle some maximum number of active processes. Remember to initialize this data structure correctly and add the init user process to the queue so it can be scheduled and run.
- When a process gets allocated, its scheduler-related PCB fields should be initialized correctly, and it should be put to the tail of the queue. When a process exits, it needs to be removed from the queue.
- Add in the new syscalls. We recommend adding them early so you can write user programs to test whether your queue is working as expected or not. You are an expert at adding system calls after surviving P2. This time, you'll need to understand how to fill in the structure `pstat` in the kernel and pass the results to user space and how to pass the arguments from user space to the kernel. 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` to obtain a pointer that has been passed into the kernel.
- The `scheduler()` function in `proc.c` is the main place where modifications go. Instead of iterating over the ptable, your scheduler should peek at the current head of the scheduler queue. If its timeslice (+ compensation ticks if applicable) has not been used up for this scheduling cycle, keep scheduling it. Otherwise, move to the next process in the queue and put the previous head to the tail. Do the

accounting of necessary counters properly. After making these changes, give yourself a pause and write small user programs to test whether your scheduler robin-robins over the queue correctly and gives each process the correct number of ticks per cycle.

- Finally, to handle sleeps & wakeups correctly, you may want to check out the `sleep()` and `wakeup1()` functions in `proc.c`. To incorporate with processes within the duration of a sleep syscall, some changes may need to be done to `proc.c: wakeup1()` and `sysproc.c: sys_sleep()`.

If you learn better by listening to someone talk while they walk through code, you are welcome to watch **this (old) video (https://www.youtube.com/watch?v=eYfeOT1QYmg)** of another CS 537 instructor talk about how the default xv6 scheduler works. Note that the project for that semester is different than what you are implementing, but you might find the code walkthrough useful.

6) Other Requirements

This project may be performed with **one project partner**. We recommend that each of you understand each portion of your code. This project does not entail writing too many lines of code, but it does require that you get everything exactly right. Thus, if you both work together on understanding the existing code and debugging your new code, you will probably work most efficiently.

We will do and test this entire project with **compiler optimizations turned off**. Make sure in your Makefile `CFLAGS` variable contains `-Og` and not `-O2`.

To make your life easier and our testing easier, you should run xv6 on only a **single CPU**. Make sure in your Makefile `CPUS := 1`.

We suggest that you start from the initial source code of xv6 instead of your own code from p2 as bugs may propagate and affect this project.

Copy the provided `pstat.h` from `/p/course/cs537-yuvraj/public/scheduler/pstat.h`.

Remember to run the xv6 environment, use `make qemu-nox`. Type **ctrl-a**, release, followed by **x** to exit the emulation. There are a few other commands like this available; to see them, type **ctrl-a**, release, followed by **h.**

When debugging, remember that gdb can be quite useful!

6.1 Code Development

When working with a project partner, you may want to be able to read and write the files in a shared directory on the instructional machines. The CSL machines run a distributed file system called AFS that allows you to give different file permissions to your project

partner than to everyone else.

First, use the "fs" command to make sure no one can snoop about your directories. Let's say you have a directory where you are working on project 2, called "~/myname/p3". To make sure no one else can look around in there, do the following:

'cd' into the directory

```
prompt> cd ~/myname/p2
```

check the current permissions for the "." directory ("." is the current dir)

```
prompt> fs la .
```

make sure system:anyuser (that is, anybody at all) doesn't have any permissions

```
prompt> fs sa . system:anyuser ""
```

check to make sure it all worked

```
prompt> fs la .
Access list for . is
Normal rights:
system:administrators rlidwka
myname rlidwka
```

As you can see from the output of the last "fs la ." command, only the system administrators and myname can do anything within that directory. You can give your partner rights within the directory by using "`fs sa`" again:

```
prompt> fs sa . partnername rlidwka
prompt> fs la .
Access list for . is
Normal rights:
system:administrators rlidwka
myname rlidwka
partnername rlidwka
```

If at any point you see the directory has permissions like this:

```
prompt> fs la .
Access list for . is
Normal rights:
system:administrators rlidwka
myname rlidwka
system:anyuser rl
```

that means that **any user** in the system can read ("r") and list files ("l") in that directory, which is probably **not what you want.**

6.2 Testing

We recommend making sure that your scheduler can handle a large range of jobs. If you can run `usertests` and `forktests`, your scheduler is probably fairly robust (though unfortunately, it might not be implementing the correct policy). We strongly recommend you to first write a few small user programs to test various aspects of this project, such as getting and setting the timeslice or a process, round-robining over the queue with the correct number of ticks for each process per cycle, and having processes sleep and wakeup. Ensure correct behavior from these tests before moving on to our tester.

As always, we provide a set of public tests under `/p/course/cs537-yuvraj/tests/p3/`. On any CSL machine, use `cat /p/course/cs537-yuvraj/tests/p3/README.md` to read more details about how to list the tests and how to run the tests in batch. Note that there will be a small number of hidden test cases (20%).

6.3 Turnin

You can use up to 3 slip (late) days across all the projects throughout this semester; for example, you can use 1 slip day on three separate assignments or 3 slip days on a single assignment (or other combinations adding up to a total of 3 days). If you are using slip days on this project, you must create a file called **slip_days** with the full pathname `/p/course/cs537-yuvraj/turnin/login/p3/slipdays`, where ***login*** is your CS login name. The file should contain a single line containing the integer number of slip days you are using; for example, you can create this file with "echo 1 > slip_days". You must copy your code into the corresponding slip directory: `/p/course/cs537-yuvraj/turnin/login/p3/slip1`, `/p/course/cs537-yuvraj/turnin/login/p3/slip2`, *or* `/p/course/cs537-yuvraj/turnin/login/p3/slip3`. We will grade the latest submission.

To hand in code, create a `src/` directory under `ontime` or `slipX` and put all the files under your `xv6/` under that `src/` folder, just like in P2.

Each project partner should turn in their joint code to each of their turnin directories. Each person should place a file named **partners.txt** in their *turnin/p3* directory, so that we can tell who worked together on this project. The format of **partners.txt** should be exactly as follows:

```
cslogin1 wiscNetid1 Lastname1 Firstname1
cslogin2 wiscNetid2 Lastname2 Firstname2
```

It does not matter who is 1 and who is 2. If you worked alone, your **partners.txt** file should have only one line. There should be no spaces within your first or last name; just use spaces to separate fields.

To repeat, **both project partners should turn in their code and both should have a *turnin/p3/partners.txt* file.** Suppose you are submitting to slip1, the final directory tree should look like:

```
/p/course/cs537-yuvraj/turnin/login/p3/partners.txt
/p/course/cs537-yuvraj/turnin/login/p3/slip_days -- should contain exactly a number 1
/p/course/cs537-yuvraj/turnin/login/p3/slip1/src/Makefile
/p/course/cs537-yuvraj/turnin/login/p3/slip1/src/proc.c
/p/course/cs537-yuvraj/turnin/login/p3/slip1/src/...
```

P3 Bonus

A) Project 3 Bonus Problem:

The objective for this bonus problem is to implement a lottery scheduling system. Working in a **separate copy** of your Project 3 code, you'll modify some of the work you did on the round robin scheduler to add a lottery-based aspect to process scheduling.

The bonus problem is worth **20 points**

A.1 Continuing from Project 3 Code:

Before you begin to implement the lottery features for the bonus problem, you must create a new folder for your work on the bonus problem. Simply copy your final source code for the "core" project 3 work, and work from this copy. Because this bonus problem will change the core behavior of the scheduler, you need one copy to pass the project tests, and this new copy to pass the bonus tests. When you submit the project, you will place two subfolders insider your `/p/course/cs537-yuvraj/turnin/login/p3/turnin` folder: `src` and `src-b`. If we do not see a `src-b` folder for your submission, we will assume you chose not to attempt the bonus problem.

More details on turning the code in are given in A.5; for now just make sure you are not overwriting the code you intend to turn in for the "core" project 3.

A.2 Lottery Ticket System

Recall, in part 1.1 of the project, you added a feature to allow each process a given time slice when it receives a turn on the processor. However, this can be an issue in a lottery system. Suppose Process A has a time slice of 100 ticks, while processes B and C each have 5. If the lottery works in A's favor on a few consecutive rounds, it will starve B and C. Thus, the first step will be to return to the original behavior, where each process only runs for a single tick before the next scheduling event happens.

Second, in part 1.2, you added a feature to compensate processes for any ticks of their time slice that went unused due to the process sleeping. Because we no longer run processes for more than one tick before the next lottery round, you should remove all compensation features from your code (or at least disable those features). In general, we will not deal with sleeping processes for this P3 Bonus.

However, this does not mean ***all*** your work on the time slices is wasted. You may modify your code for time slices (`getslice`/`setslice` system calls, `pstat timeslice` value, etc.) to represent the number of lottery tickets a process possesses. For the situation above, then, A will have 100 tickets, while B and C will have 5 tickets each. Over a sufficient time period, A will average 100 ticks on the processor for every (combined) 10 ticks of B and C, without starving those lower-priority processes.

A.3 Select Random Ticket in Schedule

With processes storing tickets instead of ticks, and the original 1-tick time per scheduling event restored, you can now implement the lottery. Your `scheduler` function should first determine the total number of tickets for this round of the lottery (that is, the combined number of tickets held by runnable processes). You can either store this value, updating when a new process arrives/leaves, or calculate the value on the fly each time. Then, generate a random ticket number, and schedule the process holding this ticket. You may wish to review the documentation on random number generation in C: **Pseudo-random number generation (https://en.cppreference.com/w/c/numeric/random)**

But which process holds which numbers? For simplicity, assume that the processes are given tickets sequentially, in the order they show within the ptable. Thus, continuing our example from above, assuming the order within the ptable is A, B, C:

- Process A holds tickets 1-100
- Process B holds tickets 101-105
- Process C holds tickets 106-110

Then given a random ticket number in the range 1-110 (inclusive), it is easy to determine which process holds the winning ticket.

A.4 Testing Your Scheduler

You can test your code for P3 Bonus with a lightly-modified version of your `loop` and `schedtest` user programs from P3. The `loop` program can remain the same, except that it should no longer accept a `sleepT` parameter (since we are not handling sleep compensation). When a process runs the `loop` program, it should simply jump directly into the work loop. `schedtest`'s `sliceA` and `sliceB` parameters should be reinterpreted as `ticketsA` and `ticketsB`, the number of tickets for each process. In addition, because we no longer have compensation ticks, there is no use for the parent process in `schedtest` to print `comptricksA` and `comptricksB`. Instead, the parent should print the values of `schedticksA` and `schedticksB`, respectively.

An additional note: because a lottery is based on randomness, it should never do the same thing twice, in principle. However, the random number generator in the C standard library is not truly random, but a pseudo-random generator whose sequences of "random" numbers are determined by a "seed" value. For testing your code, you should hard-code the seed value to 0, to ensure that you get the same results each time the program is run with the same parameters.

With these modifications, you can effectively test the behavior of your scheduler. By using increasing values of `sleepParent` (holding other `schedtest` parameters constant), you should be able to see the `schedticks` values converge to the correct proportion (e.g. if `ticketsA` = 2 * `ticketsB`, you should see the `schedticksA` value approach double the `schedticksB` value).

A.5 Code Submission

As mentioned in A.1, you will submit your work for the P3 Bonus in a separate folder alongside the source for your main P3 submission. Your work for P3 Bonus should be a full, separate copy of the xv6 operating system, with the modifications specified above.

Recall, your P3 code will be submitted as a folder called `src` within the `/p/course/cs537-yuvraj/turnin/login/p3/turnin` folder (where `login` is your CSL user name, and `turnin` is replaced by `slip1` / `2` / `3` if you are using slip days). Your P3 Bonus code must be submitted in a folder called `src-b`, also within the `turnin` folder. All other aspects of assignment submission remain the same. Thus, if you are submitting to `slip1`, your directory tree should look like:

```
/p/course/cs537-yuvraj/turnin/login/p3/partners.txt
/p/course/cs537-yuvraj/turnin/login/p3/slip_days -- should contain exactly a number 1
/p/course/cs537-yuvraj/turnin/login/p3/slip1/src/Makefile
/p/course/cs537-yuvraj/turnin/login/p3/slip1/src/proc.c
/p/course/cs537-yuvraj/turnin/login/p3/slip1/src/...
/p/course/cs537-yuvraj/turnin/login/p3/slip1/src-b/Makefile
/p/course/cs537-yuvraj/turnin/login/p3/slip1/src-b/proc.c
/p/course/cs537-yuvraj/turnin/login/p3/slip1/src-b/...
```

If we do not see a `src-b` folder for your submission, we will assume you did not choose to attempt the bonus problem.