P2A: Scheduling

Due Mar 10 by 11:59pm **Points** 120
Available Feb 24 at 12am - Mar 15 at 9:59pm 20 days

Objectives

- Understand existing code for performing context-switches in the xv6 kernel.
- Implement a basic lottery scheduler that boosts the priority of well-behaving processes.
- Modify how sleep is implemented in xv6 to match our scheduler.
- Implement system calls that set/extract process state.
- Use the provided user-level programs to empirically validate the fair and proportional share properties of a lottery scheduler.
- Gain familiarity in working on a large codebase in a group.

Updates (read these once you're familiar with the rest of the assignment):

- **Feb 26:** We wanted to clarify a few more rules of boosted behaviour. Assume process A has 5 tickets and 3 boosted ticks remaining:
  - Its children will always inherit 5 tickets, whether or not process A was boosted at the time of child creation.
  - Children processes will not inherit boostsleft from their parents.
  - When running the lottery, assume that this process has double tickets(10) since it has boosts left(3).
  - `getpinfo` should return tickets=5 and boostsleft=3. Do not return tickets=10 even when the process is boosted.

Overview

This project is different from the first one in that it can be done in pairs. Thus, expect to spend a lot of time communicating with each other and working together. **(See 6.2)**

In this project, you will modify xv6 to schedule processes using lottery scheduling. Further, your scheduler will boost the priority of sleeping processes so they get their fair share of CPU time. In doing so, you will master how a context switch happens in xv6.

You should be comfortable with an xv6 build environment after P1: syscall. You can either code directly on a CSL machine, or on your own machine. Remember to test that your code compiles on a CSL machine before submitting it. You will be using the current version of xv6. Copy the xv6 source code into your private directory using the following commands:

```
prompt> cp /p/course/cs537-swift/public/xv6.tar.gz .
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 the boosted lottery-scheduler should behave.
2. recommend a code structure for your scheduler.
3. specify the new system calls you must add
4. describe how to use the provided user-level applications to test your scheduler.
5. describe the existing xv6 scheduler implementation
6. give suggestions for implementing this project

1) Boosted Lottery 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.

Since RR treats all jobs equally, it is simple and fairly effective. However, there are instances where this "ideal" property of RR is detrimental: if compute-intensive background jobs (anti-virus checks) are given equal priority to latency-sensitive operations, operations like music streaming (latency-sensitive) suffer. **Lottery Scheduling (https://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-lottery.pdf)** is a proportional share policy which fixes this by asking users to grant tickets to processes. The scheduler holds a lottery every tick to determine the next process. Since important processes are given more tickets, they end up running more often on the CPU over a period of time.

You will implement a variant of lottery scheduler with the following properties:

1. At each tick, your scheduler holds a lottery between runnable processes and schedules the winner.
2. If a process is blocked and unable to participate in the lottery for ***x*** ticks, its tickets would be doubled for the next ***x*** lotteries.
3. For this assignment, you will need to improve the sleep/wakeup mechanism of xv6 so processes are unblocked only after their sleep interval has expired instead of every tick.

1.1. Basic Lottery Scheduler

In your scheduler, each process runs for an entire tick until interrupted by the xv6 timer. At each tick, the scheduler holds a lottery between RUNNABLE processes and schedules the winner. When a new process arives, it should have the same number of tickets as its parent. The first process of xv6

should start with 1 ticket. You will also create a system call `settickets(int pid, int tickets)`, to change the number of tickets held by the specified process.

As an example, if there is a process A with 1 ticket and process B with 4 tickets, this is what the timeline *might* look like:

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

In this case, A was scheduled for 4 ticks, and B for 16. Given sufficiently large number of ticks, we can expect the ratio of CPU time per process to be approximately the ratio of their tickets.

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". The next scheduling decision will be made the next time a timer tick occurs.

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 the number of ticks a process was sleeping. Once it's runnable, it will boost (double) the number of tickets for the same number of ticks. For instance, if a process slept for 5 ticks, its tickets would be artificially doubled for the next 5 lotteries it participates in.

Example 1: a process A has 1 ticket and B has 4 tickets. Process A sleeps for 2 ticks, and is therefore given double tickets for the next 2 lotteries after it wakes. This allows the scheduler to keep the CPU usage ratio equal to the ratio of tickets, even when process A is blocked. The timeline could look like the following in this case (description below the timeline):

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

Note the following:

1. At each tick, we consider the tickets a runnable process has. Ticks 3, 4, 15 and 16 only have 1 runnable process, thus only B participates in the lottery.
2. Once A is no longer blocked, it starts participating in lotteries. **If it slept for *x* ticks, it will participate in the next *x* lotteries with double the tickets.**
3. 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, process A becomes RUNNABLE at tick 17. You should boost its tickets only for the next 2 lotteries, regardless of whether A actually wins the lottery or not.

Example 2: Accumulation of favourable rounds. Assume A sleeps for 3 ticks, thus its tickets are boosted for the next 3 ticks. What happens if it sleeps for 3 ticks again while it still has `remaining` boosts left? In this case, its tickets should be boosted for the next (3+`remaining`) rounds after it wakes. This is important for maintaining the lottery scheduler's proportional-CPU-share property. Given that A couldn't even participate in the lottery when it was blocked, we need to ensure it has a higher chance of winning in the corresponding number of lotteries.

This is a potential timeline for the described situation. Realize that after tick 5, A was supposed to be boosted for 3 ticks. Since it got blocked before that, we ensure that it doesn't lose the number of favourable/boosted rounds once it wakes.

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

Example 3: Multiple Processes. In many workloads, there will be multiple processes that are blocked. The tickets of all of these processes will be independently doubled based on how long each slept. This isn't a special case; just ensure that you aren't assuming only one process is sleeping for a fixed amount of time. Different processes could be blocked for varying number of ticks.

Example 4: Inheritance during boost: Read the first 3 points of Feb 26th update.

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), it will lose at least 1 favourable round since it ended up participating in the lottery.

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 might also use this mechanism to count how many ticks a process was sleeping for so it can be boosted accordingly. 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) helpful.

2) Implementation details:

In order to hold a lottery, we first need a way to generate a random number. Here's the code to do so:

Copy the following at the top of `proc.c` after the headers:

```
#define RAND_MAX 0x7fffffff
uint rseed = 0;

// https://rosettacode.org/wiki/Linear_congruential_generator
uint rand() {
    return rseed = (rseed * 1103515245 + 12345) & RAND_MAX;
}
```

Now that we have a way to generate a random number, you have everything you need to implement a lottery scheduler. You need to do 2 things now: (a) hold a lottery among the RUNNABLE jobs, (b) modify the current RR policy to actually use your policy. Please place the lottery code in a separate function:

```
// returns a pointer to the lottery winner.
struct proc *hold_lottery(int total_tickets) {
    if (total_tickets <= 0) {
        cprintf("this function should only be called when at least 1 process is RUNNABLE");
        return 0;
    }

    uint random_number = rand();    // This number is between 0->4 billion
    uint winner_ticket_number = ... // Ensure that it is less than total number of tickets.

    // pick the winning process from ptable.
    // return winner.
}
```

You should call hold_lottery when you're modifying the existing scheduler code.

2) New system calls

You'll need to create 3 new system calls for this project.

1. `int settickets(int pid, int n_tickets)`. This syscall is used to set the number of tickets alloted to a process. It returns 0 if the pid value is valid and n_tickets is positive. Else, it returns -1.
2. `void srand(uint seed):` This sets the rseed variable you defined in proc.c. (Hint: in order to effectively mutate that variable in your sys_srand function, you will need to define `extern uint rseed`. Check **https://stackoverflow.com/a/1433226 (https://stackoverflow.com/a/1433226)**).
3. `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 successful, some basic information about each running 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 tickets[NPROC];  // how many tickets does this process have?
  int runticks[NPROC];  // total number of timer ticks this process has been scheduled
  int boostsleft[NPROC]; // how many more ticks will this process be boosted?
};
```

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-swift/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.

During a boost, you should not double the number of tickets you return in `getpinfo.` Instead, return the original number of tickets. Check the 4th point of Feb 26 update.

3) Creating your own user-level applications and using the provided applications.

To demonstrate that your scheduler is doing at least some of the right things, you might want to create user-level applications. We provide 2 such applications: `loop` which does some heavy compute, and a wrapper program, `schedtest` which runs multiple instances of `loop` with different tickets. The code for these programs is available in `/p/course/cs537-swift/public/scheduler`. Copy these into your src folder and have xv6 compile these as user programs by modifying the UPROGS variable in the Makefile.

`loop` is a dummy job that just does some work. 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! 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 number of tickets and sleep time of each child. The `schedtest` application takes exactly 5 arguments in the following order:

```
prompt> schedtest ticketsA sleepA ticketsB sleepB sleepParent
```

- `schedtest` spawns two children processes, each running the `loop` application. One child A is given `ticketsA` tickets and execs `loop sleepA`; the other child B is given `ticketsB` tickets and it runs `loop sleepB`.
- Specifically, the parent process calls `fork(), settickets()` and `exec()` for the two children `loop` processes, **A before B.**
- The parent `schedtest` process then sleeps for `sleepParent` ticks by calling `sleep(sleepParent)`
- After sleeping, the parent calls `getpinfo()`, and prints one line of two numbers separated by a space: `printf(1, "%d %d: %d\n", runticksA, runticksB, runticksA / runticksB)`, where `runticksA` is the `runticks` 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.

You will be able to run `schedtest` to test the basic compensation behaviour of your scheduler. For example:

```
prompt> schedtest 10 0 2 0 120
99 19 5
# Since it's randomized, it's difficult to completely predict the output of your program.
# However, the key thing to note is that the ratio of runticksA/runticksB is close to the ratio o
f tickets 10/2.
# You should play around with the number of tickets you grant each process and see the CPU-ratio
 match.
# As long as one of the child hasn't terminated yet, increasing the sleepParent value should lead
to the CPU-use ratio converging to 5.
# You can also modify the code for loop so that it runs longer if needed.
```

Thus, you might need to write your own userlevel programs to validate your scheduler instead of just doing `cprintf.`

As an example, in order to test the boosted algorithm, you can copy the `loop` program into another `loop-print-info` program. `loop-print-info` can print `boostsleft` using `getpinfo()` right after it finishes sleeping. Make sure you sleep for >= 10 ticks so that your process has a reasonable chance of getting scheduled during its boosted phase. Thus you can the expect value of `boostsleft` to be between 0 and 10. You could also print `boostsleft` just before the `loop-print-info` program terminates to ensure that boostsleft has counted down successfully.

We will not test your implementation of any userspace programs: their purpose is to primarily help you test your scheduler. You are welcome to write more user applications and play with different values to test other aspects of the scheduler.

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, `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()`.

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 that 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 will consume a boosted ticket (favourable round) and start participating in the lottery even when it can't do useful work. 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 rather than implementing too much functionality at one time. It might also be useful to use `cprintf` to print debugging information when you're making modifications inside the kernel.

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 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:

- When a process gets allocated, its scheduler-related PCB fields should be initialized correctly.
- Tracking tickets of active processes can become tricky given the myriad ways a process can change its state. Thus for simplicity, at each tick, you can count the number of tickets of RUNNABLE jobs.
- When determining a winner, you do not need to sort the jobs in any manner. You can simply iterate over the proclist.
- You should code your lottery algorithm in the provided code structure. You will need to call it within the scheduler's infinite loop such that it first picks a winner and then schedules it.
- Assume A has 3 tickets, B has 2 and C has 1 ticket: A wins if the winning ticket is 0,1 or 2. B wins if the winning ticket is 3 or 4. And C wins if the winning ticket is 5. This assumes that your proclist has A,B,C in order.
- Add in the new syscalls early. This allows you to quicky write user programs to test whether you're setting the process state correctly. You are an expert at adding ptr-based system calls after surviving P2.
- The `scheduler()` function in `proc.c` is the main place where modifications go. Spend some time to understand what the existing code is doing and what all you need to strip from it. Also ensure you're releasing and acquiring locks in a manner consistent with the existing implementation. `cprintf` should be a godsend to analyze whether lottery is running as expected.
- Other important routines that you may need to modify include `allocproc()` and `userinit()`. Of course, you may modify other routines as well.
- 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()`. This is a nice place for you to count how many ticks a process is blocked for, and thus how much it should be compensated.

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.

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 on**. Make sure in your Makefile, `CFLAGS` variable contains `-O2` and not `-Og`.

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-swift/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/p2". 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 Collaborating

This assignment requires you to understand the codeflow of xv6 context switches in depth. Given this is a group project, we strongly urge you to not work in isolation. Partners can write different pieces of code, but we recommend sitting together to figure out where to place the code.

One way of accomplishing this is by siting together and understanding the existing RR scheduler and the flow of an xv6 context switch. Then one person could focus on adding the syscalls which modify/extract state, while the other implements the lottery scheduler. You should then reconvene and test your merged code. After that, one person could attempt the sleep/wakeup part while the other focusses on boosting the processes for the requisite amount of ticks. At each point, ensure that both partners know exactly what the other person is working on and how the two implementations will fit together. (You could either work on a common code-base or use git based branches to periodically merge your code).

The other approach is *true pair programming:* When done correctly, pair programming is extremely productive and educational for both partners. Set up a common workplace (it could be in the CSL lab or even a zoom session works) where you share the same screen and hardware. Both partners read, understand and write code together at the same pace. It's common to come up with a mental solution together and then start implementing it. One partner can start writing the code, while the other keeps checking the logic of the code as it is being written. We also recommend frequently shifting roles to ensure understanding and productivity.

Key rule: do not work in isolation.

6.3 Testing (piazza post will be made when tests are available)

We provide a set of public tests under `/p/course/cs537-swift/tests/p2a/`. On any CSL machine, use `cat /p/course/cs537-swift/tests/p2a/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%).

6.4 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-swift/turnin/login/p2a/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-swift/turnin/login/p2a/slip1`, `/p/course/cs537-swift/turnin/login/p2a/slip2`, or `/p/course/cs537-swift/turnin/login/p2a/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 P1.

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/p2a* 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/p2a/partners.txt* file.** Suppose you are submitting to slip1, the final directory tree should look like:

```
/p/course/cs537-swift/turnin/login/p2a/partners.txt
/p/course/cs537-swift/turnin/login/p2a/slip_days -- should contain exactly a number 1
/p/course/cs537-swift/turnin/login/p2a/slip1/src/Makefile
/p/course/cs537-swift/turnin/login/p2a/slip1/src/proc.c
/p/course/cs537-swift/turnin/login/p2a/slip1/src/...
```