CS 4414-001 Spring 2019

Main page Homeworks Policies Gradebook Study Materials Submission Quizzes Schedule

This website is not for a current offering of CS 4414.

Table of contents:

Changelog:

- 18 August 2019: add section re: this assignment being based on Arpaci-Dusseau's

Your Task

1. Add support for tracking the number of "tickets" in a process:
   - adding a system call called `settickets` which sets the number of "tickets" for a process with the following prototype: `int settickets(int number)`. By default, processes should have 10 tickets. You can assume that the maximum number of tickets per process is 100000. The number of tickets should be inherited by children created via `fork`.
2. Add a system call called `getprocessesinfo` with the following prototype

```
int getprocessesinfo(struct processes_info *p);
```

   where `struct processses_info` is defined as:

```
struct processes_info {
    int num_processes;
    int pids[NPROC];
    int ticks[NPROC];       // ticks = number of times process has been scheduled
    int tickets[NPROC];     // tickets = number of tickets set by settickets()
};
```

The system call should fill in the `struct` pointed to by its arguments by:

- setting `num_processes` to the total number of non-UNUSED processes in the process table
- for each i from 0 to `num_processes`, setting:
  - `pids[i]` to the pid of `i`th non-UNUSED process in the process table;
  - `ticks[i]` to the number of times this proces has been scheduled since it was created (don't adjust for whether or not the process used its whole timeslice);
  - `tickets[i]` to the number of tickets assigned to each process.

3. Add support for a lottery scheduler to xv6, by:
   - changing the scheduler in `proc.c` to use the number of tickets to randomly choose a process to run based on the number of tickets it is assigned;
4. Submit your code by running `make submit` and submitting the result `.tar.gz` file.

Some Supplied Test Programs

We have supplied some testing programs to aid you in figuring out whether your implementation is correct. Note that these programs are not complete tests. For example, they do not test that you set the correct default ticket count, that ticket counts are inherited by child processes correctly, and they may miss combinations of ticket counts some implementation techniques have problems with.

`processlist.c`

We have supplied processlist.c which is a test program which runs `getprocessinfo` and outputs the information in the struct.

You can add this test program the same way you added a test program for the xv6intro assignment.

`timewithtickets.c`

We have supplied timewithtickets.c which is a test program which:

- takes as command line arguments an amount of time to run for followed by a list of numbers of tickets to assign to each subprocess. Each subprocess runs an infinite loop and is killed after running of the designated amount of time. By default, the process does nothing during the infinite loop, but you can change `#undef USE_YIELD` to `#define USE_YIELD` to have the process instead call the system call `yield()` repeatedly in the loop.
- outputs a table of the programs in order, with the number of tickets assigned to each and the number of ticks it ran, as reported by getprocessesinfo
- reports an error if `getprocessesinfo`
  - the `num_processes` field returned is negative or exceeds `NPROC`
  - indicated that a child process had the wrong number of tickets;
  - was missing a child process

`lotterytest.c`

> *If you get an error about `__divdi3` not being defined when trying to use this, try running `sudo apt install gcc-multilib` and recompiling.*

We have supplied an automated test program lotterytest.c which essentially runs several cases of `timewithtickets.c` and checks the number of ticks reported. It also has a couple of test cases to make sure that programs that are not runnable don't confuse your scheduler.

It has some rather complicated code that attempts to perform a statistical test. We've set the parameters of the test so it should almost never indicate your code is wrong because of "bad luck". However, this test might be sensitive enough to detect if you don't use unbiased randomness (particularly if you use a pseudorandom number generator with a small range). I'm hoping this code is correct, but it's very tricky and not easy to test if it's too sensitive or not sensitive enough. See also the note on bias below.

- If you get errors about tests not running for a non-trivial number of ticks, your schedule might be scheduling processes that aren't part of the test or not scheduling any process when it should schedule one or your scheduler might not be tracking ticks correclty or your system may just be too slow (especially if it spends a lot of time running `cprintf`s you added). You can try to make the test wait longer by increasing the `#define` for `SLEEP_TIME` to see if it's just the system being too slow.

- When your scheduler appears to have the wrong ratio of ticks, lotterytest runs some additional tests to diagnose this (checking if the ticks are in the right ballpark even if they are definitely statistically wrong). This means that the total tests run count will vary.
- If you see a tick count of -99999, this usually means that `lotterytest` didn't find the process at all in the `getprocessesinfo` output. (There is usual a test failure before this tick count indicating this.)

Hints

Reading on xv6's scheduler

1. Read Chapter 5 of the xv6 book for documentation on xv6's existing scheduler.
2. You can review Chapter 3 of the book, and/or the instructions on the intro homework on how to add system call and utility functions like `argptr`.

Reading on Lottery Scheduling

1. For an alternate explanation to the lecture slides, see Chapter 9 of Arpaci-Dusseau.

Suggested order of operations

1. Implement `settickets`, but don't actually use ticket counts for anything.
2. Implement `getprocessesinfo`. Use the processlist.c or other programs (e.g. ones you write) based on it to verify that it works.
3. Add tracking of the number of ticks a process runs. Use the timewithtickets.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 timewithtickets.c to test it. Then use lotterytest.c to further test it.

Tracking the number of ticks a process has been running

1. `proc.h` contains xv6's process control block, to which you can add a member variable to track the number of ticks a process has used.

2. You may need to modify `fork` (in `proc.c`) or related functions to make sure the tick count variable is initialized correctly.

Adding `settickets`

1. You can use `argint` to retrieve the integer argument to your system call. (Making `sys_settickets` take an argument will not work.)
2. Like for tracking the number of ticks a process has been runing, you will need to edit the process control block in `proc.h`.
3. Follow the example of similar system calls in `sysproc.c`.

Adding `getprocessesinfo`

1. You can use the `argptr` to retrieve the pointer argument in your system call handler.
2. You should iterate through the process list `ptable`, skipping over UNUSED processes.
3. Look at the code for `kill` in `proc.c` for an example of how to search through the list of processes by `pid`.
4. 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.

Adding the lottery scheduling algorithm

1. You will need to add a psuedorandom number generator to the kernel. We've supplied a suitable version of the Park-Miller psuedo-random number generator (use the `next_random` function to get a random number between 1 and MAX_RANDOM) from Wikipedia's page on Lehmer random number generators. You may use psuedorandom number generator code from elsewhere, so long as you clearly cite where obtained the code from.
2. It is okay if your pseudorandom number generator uses a fixed seed. But if you don't want to do this xv6 has a function `cmostime` that reads the current time of day.
3. The logic in `schedule` implements a round-robin scheduling algorithm, by iterating through all processes, scheduling each one as it iterates though them. You most likely will modify it to instead, iterate through all processes (perhaps more than once) ***each time a new process needs to be scheduled*** to choose which one to run.
4. xv6 does not support floating point, so you will need to do your random selection without `float`s or `double`s

4. xv6 does not support floating point, so you will need to do your random selection without `float`s or `double`s.

5. `getprocessesinfo` provides you the information you need to test that your lottery scheduler behaves correctly. You will probably not use it in the implementation of the scheduler itself.

6. When there are no runnable processes, your schduler should release the process table lock to give interrupts (like for keypresses) a chance to make programs runnable, then reacquire that lock and iterate through the process table again.

Identifying panics

1. 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.

   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.

2. 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*.

Note on random bias

1. A common source of bias is off-by-one errors in using ticket counts. Looking at what process is chosen with small ticket counts (e.g. 1, 2, 3) is helpful for this.

2. If you take a random number `x` in the range, say, 0 to 1023, and use it to choose a random number from 0 to 99 using `x % 100`, then you will choose numbers between 0 and 23 too often. One way to avoid this is to check if `x` is greater than 999, and, if so, select another random number `x` between 0 and 1023 (and keep doing this until you get one between 0 and 999 inclusive).

3. If you are experiencing large deviation from the expected ratios, the cause is probably something else, like double-counting

a process’s tickets when deciding how to map a random number to a process to run.

Credit

This assignment is based on Arpaci-Dusseau’s version of an xv6 lottery scheduler project.

CS 4414-001 Spring 2019

Charles Reiss
cr4bd@virginia.edu