Scheduling

Prof. Darrell Long

Jack Baskin School of Engineering
University of California, Santa Cruz

Winter 2017
Assigned: Wednesday 25 January — *00:00*
Due: Friday 17 *February* — *17:00*

Goals

The primary goal for this project is to modify the FreeBSD scheduler to use lottery scheduling rather than the current scheduler.

This project will also teach you how to experiment with operating system kernels, and to do work in such a way that might crash a computer. You'll get experience with modifying a kernel, and (at some point) will end up with an OS that doesn't work, so you'll learn how to manage multiple kernels, at least one of which works.

Basics

The goal of this assignment is to get everyone up to speed on modifying FreeBSD and to gain some familiarity with scheduling. In this assignment, you are to implement lottery scheduling in FreeBSD. A lottery scheduler assigns each process's threads some number of tickets, then randomly draws a ticket among those allocated to "ready" processes (threads) to decide which process to run. That process is then allowed to run for a set time quantum, after which it is interrupted by a timer interrupt and the process is repeated. The number of tickets assigned to each process determines the likelihood that it'll run at each scheduling decision, and thus (over the long term) the relative amount of time that it gets to run. Processes that are more likely to get chosen each time will get chosen more often, and thus will get more CPU time.

In order to be able to increase/decrease the number of tickets assigned to the current process, you'll need to modify the `nice()` system call. In addition, you will need to add a new system call that can allow transferring tickets from a process to another process, called `gift(t,pid)`; where `t` is the number of tickets to transfer as a gift to the process with the process id `pid`. The transferred tickets would then be split evenly among its threads.

Details

In this project, you'll modify the scheduler for FreeBSD. This should mostly involve modifying code in `sys/kern/sched_ule.c`, though you may need to modify an *include file* or two. You'll need to modify the `nice()` system call to increase or decrease the number of tickets allocated to a process. You'll also need to introduce a `new gift(t,pid)` system call to allow a process to transfer a *t* number of its own tickets to any other process identified by *pid*. Invoking gift with *0* tickets and process id *0*; e.g. `gift(0,0)`, will return the number of tickets available for the invoking process that can be transferred to another process. Finally, Invoking gift with a number of tickets that cannot be satisfied will return the available number of tickets, acting like `gift(0,0)`, otherwise upon success gift will return *0*.

Note that the lottery scheduling, the modified `nice()`, and the introduced `gift(t,pid)` must only be used for *user processes*—those whose (effective) user IDs are non-zero (non-root). This is a good way to ensure that you don't end up with deadlocks or other problems.

We strongly recommend that you read Section 4.4 from the optional text (*Design and Implementation of the FreeBSD Operating Systems*). This section explains how the current scheduler works, and describes which routines are called when. This information will be invaluable in figuring out which routines need to be modified to implement lottery scheduling. Focus on the routines that are called both at context switch time and less frequently to place processes in the appropriate run queues.

Lottery Scheduling

The current FreeBSD scheduler uses 64 *run queues*, some of which only have system processes, and the rest of which are used for user processes. You're going to add exactly three queues for non-root user processes: one for *interactive* processes, one for *timeshare* processes, and one for *idle* processes. You can use the existing mechanism for deciding which processes go where, but once you've done that, you should place the processes in the appropriate (single) queue and use tickets to select the one to run. This means that only root processes will be placed into the (standard) 64 run queues; if there are any processes here, they get priority. If there are no processes here, the scheduler should first check the interactive queue and, if there are no processes there, check the timeshare queue, and then check the idle queue. This means that interactive processes are fully prioritized over timeshare processes, but that's OK—interactive processes are likely to end up being *descheduled* quickly anyway.

A lottery scheduler assigns each process some number of tickets, then randomly draws a ticket among those allocated to ready processes to decide which process to run. That process is then allowed to run for a set time quantum, after which it is interrupted by a timer interrupt and the process is repeated. The number of tickets assigned to each process determines both the likelihood that it will run at each scheduling decision as well as the relative amount of time that it will get to execute. Processes that are more likely to get chosen each time will get chosen more often, and thus will get more CPU time.

By default, each process gets 500 tickets when it's created. A process may not have more than 100,000 tickets, and may not go below 1 ticket; All necessary validations need to be applied to guard this restriction. The `nice()` call should be used to increase/decrease the tickets of a process, and the `gift(t,pid)` can be used to transfer tickets between processes.

Each time the scheduler is called (for a context switch), it uses the current mechanism to select a scheduler queue. If the selected queue is either the interactive or timeshare queue, it picks a number from [0, *T*−1], where *T* is the total number of tickets in the queue being run. *T* should not be calculated during the context switch, but rather should be tracked as processes are added or removed from the queue—calculating *T* each time would be too slow. The number use a 64-bit integer chosen from a pool of random numbers calculated during the infrequent scheduler calls (the random number generator shouldn't run at context switch time). The 64-bit integer can then be taken modulo *T*, yielding *r*. Go down the list of processes in the queue, adding the number of tickets each one has, until you reach a number larger than *r*, and that's the process to run.

Bonus Functionality (2%): You may extend the functionality of the gift system call with an extra boolean flag parameter `gift(t,pid,flag)`. The flag is a boolean indicator indicating if the transferred tickets can be transferred further by the receiving process or not; if flag is true (1) the tickets are transferable, otherwise they can only be used by the receiving process. For this you will need to have two variables (transferable, and local) whose sum represents the number of tickets available for each process. In that case the initial 500 tickets are considered transferable.

Building the Kernel

Rather than write up our own guide on how to build a FreeBSD kernel, we'll just point you at the guide from the FreeBSD web site. You don't need to worry about taking a hardware inventory, as long as you don't remove any drivers from the kernel you build (and there's no reason you should do this). Focus on Sections 9.4–9.6, which explain how to build the kernel and how to keep a copy of the "stock" kernel in case something goes wrong. Of course, we're happy to help you with building a kernel in laboratory section or office hours.

A couple of suggestions will help:

- Try building a kernel with no changes first. Create your own `config` file, build the kernel, and boot from it. If you can't do this, it's likely you won't be able to boot from a kernel after you've made changes.
- Make sure all of your changes are committed before you reboot into your kernel. It's unlikely that bugs will kill the file system, but it can happen. Commit anything you care about using `git`, and push your changes to the server before rebooting. "*The OS ate my code*" isn't a valid excuse for not getting the assignment done.

Deliverables

As usual, you'll submit your code using `git`. To make things easier for your team, there's a way to share a single remote repository among multiple people. First, you can control permissions on your repository so that others can read and write it. Do this by running the following command:

```
ssh git@git.soe.ucsc.edu perms classes/cmps111/winter17-01/my_repo + WRITERS other_acct
```

This will add `other_acct` as a user who can read and write `my_repo`. You can, of course, run this command multiple times to add multiple users. If you want to remove someone, use – instead of +. You can always run:

```
ssh git@git.soe.ucsc.edu perms -h
```

for a help message.

Next, create a new branch for Assignment 2:

```
git checkout -b assgn2
```

At this point, you now have a new branch that everyone in your group can share. Push your changes to the server.

IMPORTANT: Doing this won't destroy any of the hard work you've done. It merely puts it into a different branch.

Once you've set this up, the people you've allowed can push to your repository on the server. The default for `git` push is to push to the most recent server repository, though you can push to any repository for which you have permission by doing the following:

```
git push origin assgn2
```

You'll also need to create an `assgn2` directory under the root, which will contain your design document, which should be called DESIGN.txt (if plain ASCII text) or DESIGN.pdf (if in PDF) as well as any other documentation you might have, such as testing strategies or test code. Formats other than plain text or PDF are not acceptable; please convert other formats (MS Word, LaTeX, HTML, etc.) to PDF. Your design document should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs.

One final thing: you need to write up a few paragraphs describing what you contributed to the group effort, and how you'd rate the other members of your group. This should be submitted

via Canvas (individually, not collectively). The goal here is for us to understand how each person contributed to the group effort. We don't expect everyone to have done the same thing, but we expect that everyone will contribute to the project. Put any other information or instructions for the grading staff in the README.txt.

Testing

Write a program that uses `gift(pid, t)` to give its tickets away to another process. You will need to know the PID of the process you want to donate to. That program will go faster, or should, and you can also inquire using `gift(0,0)` to see that it got enough tickets.

We will use a test program that runs your kernel and calls `gift()`. We will use both `gift(0,0)` and `gift(pid, t)`. Your code must work for both. You need to do extensive testing to make sure that it does what (i) is required, (ii) and you expect. Be aware of values of pid and t that may not make sense: What happens if pid does not exist? What happens if you try to give away more tickets than you have? What happens if you were to try to give away a negative number of tickets?

Moreover, you should experiment with at least *two* policies for assigning tickets. For example, you could reward I/O intensive processes either *linearly* (adding tickets), or you could punish CPU hungry processes *exponentially* (dividing the number of tickets by a constant); you can utilize the modified `nice()` and/or the `gift()` to achieve that. There are many possibilities, and if you just do what we have suggested then we will be disappointed. Be creative, but explain and justify your choices in your DESIGN document.

Deliverables Summary:

- Design Document (.txt or .pdf) [git]
- README.txt [git]
- Source code only. [git]
- Commit Id. [Canvas]
- Your contribution on [Canvas]

Hints

- *START NOW*! Meet with your group *NOW* to discuss your plan and write up your design document. design, and check it over with the course staff.
- Experiment! You're running in an emulated system—you can't crash the whole computer (and if you can, let us know...).
- Test your scheduler. To do this, you might want to write several programs that consume CPU time and occasionally print out values, typically identifying both current process progress and process ID (example: *P1-0032* for process 1, iteration 32). Keep in mind that a smart compiler will optimize away an empty loop, so you might want to use something like this program for your long-running programs.

This project doesn't require a lot of coding (typically a few hundred lines of code), but does require that you understand FreeBSD and how to use basic system calls. You're encouraged to go to the class discussion section or talk with the course staff during office hours to get help if you need it.

IMPORTANT: As with all of the projects this quarter, the key to success is starting early. You can always take a break if you finish early, but it's impossible to complete a 20 hour project in the remaining 12 hours before it's due.

Project Groups

You are working in a team for this project, which means you need to elect a *Team Captain*. The captain turns in the code by checking it in using `git`. Every other member of the team turns (using git) in a README file listing the team members and the name of the designated captain.