Scheduling

Dr. Karim Sobh

Computer Science Department
Jack Baskin School of Engineering
University of California, Santa Cruz

Spring 2017
Assigned: Tuesday 25th April - *15:00*
Due: Tuesday 9th *May - 15:00*

Goals

The primary goal of 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 process 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" threads to decide which one to run. That thread is then allowed to run for a set time quantum, after which it is interrupted by a timer interrupt and this whole procedure is repeated. The number of tickets assigned to each thread 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. Threads 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 or completely modify the number of tickets assigned to the current thread, you'll need to modify the functionality of `nice()` in the kernel such that it can modify the tickets associated with the threads of the given process. In addition, you will need to add a new system call that can allow transferring tickets from one process's threads to another process's threads, called `gift(pid, t)`; 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 among its threads, based on some policy that you should come up with.

Details

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

You will need to come up with a design decision on how to distribute the gifted tickets among the threads of the receiving process. You might do that equally or based on the threads prior execution history; give incentive to some threads over the others. Please, state your design decision very clearly in your design document and provide all justifications possible.

Note that the lottery scheduling, the modified `nice(),` and the introduced `gift(pid, t)` must only be used for *user processes/threads*—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 the relevant chapter on the ULE process scheduler 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 threads in the appropriate run queues.

Lottery Scheduling

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

A lottery scheduler assigns each thread some number of tickets, then randomly draws a ticket among those allocated to ready threads to decide which thread to run. That thread is then allowed to run for a set time quantum, after which it is interrupted by a timer interrupt and the whole procedure is repeated. The number of tickets assigned to each thread 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. Threads that are more likely to get chosen each time will get chosen more often, and thus will get more CPU time.

By default, each thread gets 500 tickets when it's created. A thread 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's threads, and the `gift(pid, t)` 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 threads 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 threads in the queue, adding the number of tickets each one has, until you reach a number larger than r, and that's the threads to run. For generating random numbers, you can use the kernel's built-in functionality or you can implement your own. If you do implement your own, it should obviously be a legitimate random number generator obviously and please do explain your random number generator in your design document. You can implement the same random number generation algorithms taken off the internet or other public domains, but make sure you cite your sources. Do not copy code directly though.

Bonus Functionality (2% Extra Credit): You may extend the functionality of the gift system call with an extra boolean flag parameter `gift(pid, t, 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 to give away. In that case, the initial 500 tickets assigned to each thread 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, if you don't remove any drivers from the kernel you build (and there's no reason you should do that). Do make sure you know how to build the kernel and how to keep a copy of the "stock" kernel in case something goes wrong before you start your design and code. Of course, we're happy to help you with building a kernel in laboratory section or TAs office hours.

If it is deemed necessary, more clarifications and references will be posted on the class forum.

A couple of suggestions will help:

- Try building a kernel with no changes first. Create your own `KERNCONFIG` 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 your changes are committed and pushed before you reboot into your updated 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.
- As this is a team project, commit and push often if you want your team members to have your latest code to work on.

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. The designated team captain can do this by running the following command:

```
ssh git@git.soe.ucsc.edu perms classes/cmps111/spring17-01/my_repo + WRITERS
other_acct
```

The `other_acct` is the team member's CruzID. 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. Make sure you're branching off the `master` branch:

```
$ git checkout -b asgn2
```

At this point, you now have a new branch that everyone in your group can share. Push your branch and any commits in it to the server so that the rest of your team can see it

IMPORTANT: Doing this won't destroy any of the hard work you've done for the previous assignments. 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 and branch for which you have permission by doing the following:

```
$ git push origin asgn2
```

You'll also need to create an `asgn2` 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 (like `Readme.txt`) 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 C programmer could duplicate your work. This includes descriptions of the data structures you created or used, all non-trivial algorithms and formulas, and a description of each function you created or used, including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs.

One final thing: each of you need to write up a paragraph or two describing what you contributed to the group effort, and how you'd rate the other members of your group. This should be submitted via eCommons. 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 threads either *linearly* (adding tickets), or you could punish CPU hungry threads *exponentially* (dividing the number of tickets by a constant); you can utilize the modified `nice()` and/or the `gift()` to achieve that. Based on your gift allocation policy, you might be assigning gifted tickets in different ways and according to different policies, and it is recommended to make this as configurable as possible. There are many possibilities, and if you just do what we have suggested then we will be disappointed. Remember, a lot of the design decisions must be made by you and your team. Be creative, but explain and justify your choices in your Design document.

Deliverables Summary:

- Design Document (.txt or .pdf), in `<repo>/asgn2/` directory. [git]
- README.txt, in `<repo>/asgn2/` directory. [git]
- Kernel source code, modifications made to existing kernel files and maybe new files added to the kernel source directory, inside `<repo>/usr/src/.` [git]
- Git Commit ID. [eCommons]
- Your individual contribution(.txt). [eCommons] (This is done individually by everyone)

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.
- If you haven't formed a team yet, do so ASAP and elect a team captain. The captain must submit team details on eCommons. eCommons will accept late submissions of team details, but the sooner you finalize your team the sooner you can start. If you're working individually, you are your own captain and make sure your team details say so.
- 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.
- You're writing code inside the kernel now, there is no C standard library that you can rely on. All you have are the system calls, which are obviously just functions somewhere inside the kernel source. Feel free to use any of them. If you want anything else, you'll have to implement that from scratch in C.

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

IMPORTANT: As with all 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 member, including the captain of the team, turns in a contribution file on eCommons.