Programming Project #2
CMPS 111, Winter 2014

Purpose

The main goal for this project is to modify the MINIX 3 scheduler to be more flexible. You must implement a lottery 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 may end up with an OS that doesn't work, so you'll learn how to manage multiple kernels, at least one of which works.

You should also read over the projects overview material (available on the Resources page) before you start this project. In them, you will find information about MINIX 3 as well as general guidelines and hints for projects in this class.

Details

In this project, you will modify the scheduler for MINIX. The Minix scheduler is split between kernel space and user space. The kernel space scheduler is a basic, default scheduler. The user space scheduler is intended for any user defined scheduling algorithms.

The Minix Wiki has an extensive discussion on user space scheduling. In particular is a report by Bjorn Smith. The report is also available on the Resources page on Piazza.

Lottery Scheduling

Each time the scheduler is called, it should randomly select a ticket (by number) and run the process holding that ticket. Clearly, the random number must be between `0` and `nTickets-1`, where `nTickets` is the sum of all the outstanding tickets. You may use the `random()` call (you may need to use the random number code in `/usr/src/lib/other/random.c`) to generate random numbers and the `srandom()` call to initialize the random number generator. A good initialization function to use would be the current date.

For dynamic priority assignment, you should modify lottery scheduling to decrease the number of tickets a process has by 1 each time it receives a full quantum, and increase its number of tickets by 1 each time it blocks without exhausting its quantum. A process should never have fewer than 1 ticket, and should never exceed its original (desired) number of tickets.

You must implement lottery scheduling as follows:

1. Basic lottery scheduling. Start by implementing a lottery scheduler where every process starts with 5 tickets and the number of tickets each process has does not change.
2. Lottery scheduling with dynamic priorities. Modify your scheduler to have dynamic priorities, as discussed above.
3. Lottery scheduling with an alternate policy **of your own design**. For example, see what happens to your interactive processes (e.g. shell) when your scheduler increase the number of tickets a process has each time it receives a full quantum. **Experiment!** In your design document,

informally compare your policy relative to that of dynamic priorities and basic lottery scheduling.

New processes are created and initialized in `kernel/system/do_fork.c`. This may be the place to initialize any data structures.

Deliverables

You must hand in a compressed `tar` file of your project directory, including your design document. You must do a "`make clean`" before creating the tar file. In addition, you should include a `README` file to explain anything unusual to the teaching assistant. Your `README` should also contain running instructions to the TA. Without running instructions, the TA is most likely to return your project to you. Your code and other associated files must be in a single directory; the TA will copy them to her MINIX installation and compile and run them there.

Do not submit object files, assembler files, or executables. Every file in the `tar` file that could be generated automatically by the compiler or assembler will result in a 5 point deduction from your programming assignment grade. If you have files that you created for testing, it is acceptable (encouraged even) to submit those with your project. Just be sure to make a list of source and test files in your `README` file.

Your design document should be called `DESIGN`, and should reside in the project directory with the rest of your code. Formats other than plain text are not acceptable. 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. A sample design document is available on the course Resources page.

Hints

- ***START EARLY!*** You should start with your 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...).
- You may want to edit your code outside of MINIX (using your favorite text editor) and copy it into MINIX to compile and run it. This has several advantages:
  - Crashes in MINIX don't harm your source code (by not writing changes to disk, perhaps).
  - Most OSes have better editors than what's available in MINIX.

  START EARLY!
- 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.
- Your scheduler should be statically selected at boot time. However, there's no reason you can't have the code for both lottery and multi-queue scheduling in the OS at one time. At the least, you should have a single file and use `#ifdef` to select which scheduling algorithm to include.
- For lottery scheduling, keep track of the total number of tickets in a global variable in `proc.c`.

This makes it easier to pick the ticket. You can then walk through the list of processes to find the one to use next.

- Use version control (RCS, SVN, git, etc) RCSto keep multiple revisions of your files. This allows you to keep multiple "coherent" versions of your source code and other files (such as Makefiles).
- Here are some additional tips to get you started.
- Did we mention that you should START EARLY!

We assume that you are already familiar with `Makefile`s and debugging techniques from earlier classes such as CMPS 101 or from the sections held the first week of class. If not, this will be a considerably more difficult project because you will have to learn to use these tools as well.

This project doesn't require a lot of coding (typically fewer than 250 lines of code), but does require that you understand how to use MINIX 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.

You should do your design ***first***, before writing your code. To do this, experiment with the existing shell template (if you like), inserting debugging print statements if it'll help. It may be more fun to just start coding without a design, but it'll also result in spending more time than you need to on the project.

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 may do this project, as well as the third and fourth projects with a project partner of your choice. However, you can't switch partners after this assignment, so please choose wisely. If you choose to work with a partner (and we encourage it), you both receive the same grade for the project. One of you should turn in a single file called `PARTNER` with the name and CATS account of your partner. The other partner should turn in files as above. Please make sure that *both* partners' names and accounts appear on all project files.

Assignment is adapted in part from Rob Duncan's OS class at Rochester Institute of Technology