Homework 3

Academic Honesty

Aside from the narrow exception for collaboration on homework, all work submitted in this course must be your own. Cheating and plagiarism will not be tolerated. If you have any questions about a specific case, please ask me. We will be checking for this!

NYU Poly's Policy on Academic Misconduct: http://engineering.nyu.edu/academics/code-of-conduct/academic-misconduct

Homework Notes :

General Notes:

- Read the assignment carefully, including what files to include.
- Don't assume limitations unless they are explicitly stated.
- Treat provided examples as just that, not exhaustive list of cases that should work.
- **TEST** your solutions, make sure they work. It's obvious when you didn't test the code.

Prerequisites:

Before starting on this assignment, make sure you have gone through the guide to setting your system up for development (available in the Resources section in NYU classes). You will need the following for this assignment:

1. git clone a **fresh** xv6 repository

```
$ git clone https://github.com/gussand/xv6-public.git
Cloning into 'xv6-public'...
remote: Counting objects: 4475, done.
remote: Compressing objects: 100% (2679/2679), done.
remote: Total 4475 (delta 1792), reused 4475 (delta 1792), pack-reused 0
Receiving objects: 100% (4475/4475), 11.66 MiB | 954.00 KiB/s, done.
Resolving deltas: 100% (1792/1792), done.
Checking connectivity... done.
```

You can follow the same instructions as **Homework 1** for setting up your system.

Homework 3: Processes and Scheduling

Part 1: Performance Measurements

In part 2 you will implement various scheduling policies. However, before that, we will implement an infrastructure that will allow us to examine how these policies affect performance under different evaluation metrics.

The first step is to extend the proc struct (see proc.h). Extend the proc struct by adding the following fields to it: ctime, ttime, stime, retime and rutime. These will respectively represent the creation time, termination time and the time the process was at one of the following states: SLEEPING, READY and RUNNING. Note that these fields contain enough information to calculate the turnaround time and waiting time for each process.

After the creation of a new process the kernel will update the process' creating time. The fields for each process state) should be updated for all processes whenever a clock tick occurs (you can assume that the process state is SLEEPING only when the process is waiting for I/O). Finally, make sure you are careful when marking the termination time of the process (note: a process may stay in the ZOMBIE state for an arbitrary length of time. Naturally this should not affect the process' turnaround time, wait time, etc)

Since all this information is retained by the kernel we need to find a way to present it to the user. To do this, create a new system call wait_stat which extends the wait system call and is defined as:

```
int wait_stat(int *wtime, int *rtime, int *iotime, int* status )
```

Where the three arguments are pointers to integers to which the wait_stat function will assign:

- wtime: The aggregated number of clock ticks during which the process waited (was able to run but did not get CPU)
- rtime: The aggregated number of clock ticks during which the process was running
- iotime: The aggregated number of clock ticks during which the process was waiting for I/O (was not able to run).
- status return the pid of the terminated child process or -1 upon failure.

Part 2: Scheduling Policies

Scheduling is a basic and important facility of any operating system. The scheduler must satisfy several conflicting objectives: fast process response time, good throughput for background jobs, avoidance of process starvation, reconciliation of the needs of low-priority and high-priority processes, and so on. The set of rules used to determine when and how to select a new process to run is called a scheduling policy.

You first need to understand the current xv6 scheduling policy. Locate it in the code and try to answer the following questions to yourself: which process does the policy select for running, what happens when a process returns from I/O, what happens when a new process is created and when/how often does the scheduling take place. HINT: We have discussed this in class and you can also look at the slides.

First, change the current scheduling code so that process preemption will be done every quanta size (measured in clock ticks) instead of every clock tick. To do this, add the following line to `param.h`

```
#define QUANTA <int value>
```

and initialize the value of QUANTA to 5. In the next part of the assignment you will add four different scheduling policies in addition to the existing policy. Add these policies by using the C preprocessing abilities.

Hints:

1. You should read about #IFDEF macros. These can be set during compilation by gcc (see http://gcc.gnu.org/onlinedocs/cpp/Ifdef.html) Modify the `Makefile` to support `SCHEDFLAG` – a macro for quick compilation of the appropriate scheduling scheme. Thus the following line will invoke the xv6 build with Round Robin scheduling:

   ```
   make qemu SCHEDFLAG=FRR
   ```

   The default value for `SCHEDFLAG` should be `DEFAULT` (in the Makefile).

2. You can (and should!) read more about the make utility here: http://www.opussoftware.com/tutorial/TutMakefile.htm

Policy 1: Default policy (SCHEDFLAG=DEFAULT)

Represents the scheduling policy currently implemented at xv6 (with the only difference being the newly defined QUANTA).

Policy 2: FIFO Round Robin (SCHEDFLAG=FRR)

This policy will extend the previous default policy. In this policy the decision of which process will run next will be on a First In - First Out basis. When a process finishes running for QUANTA time, created or finishes waiting for I/O it is considered to be the last process to arrive and waits for its next turn to run.

Policy 3: First Come First Served (SCHEDFLAG=FCFS)

This is a non-preemptive policy. A process that gets CPU runs until it no longer needs the CPU (yield/ finish/blocked). This policy is somewhat like FIFO Round Robin with infinite quanta.

Part 3: Sanity Test

In this section you will add an application that tests the impact of each scheduling policy. Similarly to several built-in user space programs in xv6 (e.g., ls, grep, echo, etc.), you can add your own user space programs to xv6.

Add a user space program called **`sanity`**. This program will start and immediately fork 20 child processes. Each child process perform a time-consuming computation. This computation must take at least 30 time units (clock ticks) and you must not perform blocking system calls (e.g., write, printf etc.) during this computation. After the child process finishes its computation, it will exit with exit status equals to its pid. The parent process will wait until all its children exit. For every finished child, the parent process must validate its exit status (by comparing it to child's pid) and print the waiting time, running time and turnaround time of each child. In addition averages for these measures must be printed.

Hints:

1. To add a user space program, first write its code (eg. Sanity.c). Next update the Makefile so that the new program is added to the file system image. The simplest way to achieve this is by modifying the lines right after "UPROGS" in the Makefile.
2. You have to call the exit system call to terminate a process execution.
3. Note on sanity test: before running a test you should estimate what the output should look like. After running the test you should compare the estimated output with the actual output. In case the test does not behave as you assumed, think carefully why it is so. This does not necessarily mean that your implementation is wrong. In any case, you should be able to explain the program's behavior to the grader.

Submit

You will use git to create a patch that contains your changes. First, tell git who you are:

```
$ git config --global user.email "you@example.com"
$ git config --global user.name "Your Name"
```

Then, commit your changes with the appropriate message for example:

```
$ git commit --all --message="Implement scheduling algorithms"
    [hw3 94b0cf7] Implement date system call
     7 files changed, 60 insertions(+), 7 deletions(-)
```

(Note: if you added any new files, you will also have to use

```
`git add <filename>` before you run `git commit`.)
```

Finally, create a patch file. The command takes an argument that specifies what code we're comparing against to make the patch; in this case I have created a git *tag* that refers to the unmodified Homework 3 called `hw3.unmodified`, so you should run:

```
$ git format-patch hw3.unmodified
    0001- Implement-date-system-call.patch
```

The command creates a file, `0001-Implement-date-system-call.patch`, containing the changes you've made. Submit this file on NYU Classes.

****Submission Note****

Don't try to edit the patch file after creating it. Doing so will most likely corrupt the patch file and make it impossible to apply. Instead, change the original file, commit your changes, and run `git format-patch` again:

```
$ git commit --all --message="Description of your change here"
    [hw3 7cc4977] Description of your change here
     1 file changed, 1 insertion(+), 1 deletion(-)
$ git format-patch hw4.unmodified
    0001-Implement-date-system-call.patch
    0002-Description-of-your-change-here.patch
```

Then submit *both* patch files.