Schedule Lab

Our Schedule Lab is a team lab. Students may work together in teams of two. Teams must work independently. Teams **may not** collaborate with other teams, other students, professors, or the Internet. Teams may use the Internet for point-specific questions. For example, if a team wants examples of how to call fork. Teams **may not** use the Internet to discover scheduler code that is copied or mimicked into their solution. Our OSTEP textbook explains scheduling and we have discussed scheduling in lectures. You are provided a starting code base. Use this material to design and implement your own code.

Introduction

In the Schedule Lab, you modify code to implement two scheduling algorithms - round robin and proportional share. You begin with code that has been extracted from the Xv6 code base. The code has been modified and built into an stand-alone executable program. The program mimics the execution of processes in various states - `EMBRYO`, `RUNNABLE`, `RUNNING`, `SLEEPING`, and `ZOMBIE`. A process does not execute when it is marked as `RUNNING`. Running a process is not required to implement a scheduling algorithm. The code provided consists of the following.

- `defs.h` - defines the function prototypes. In Xv6 these function prototypes are well-known functions such as `fork`, `wait`, `exit`, and `kill`. Since these functions exist in a normal C development environment, they have been changed to `Fork`, `Wait`, `Exit`, etc.
- `types.h` - defines general types such as unit (unsigned int), uchar (unsigned char) and ushort (unsigned short). The types uint, uchar, and ushort are common in C programming in a Linux environment.
- `proc.h` and `proc.c` - defines the proc module, which contains code to implement processes - the heart of Linux systems. A few examples.
  - `struct proc` - defines the components of a process.
  - `struct context` - defines the register context of a process.
  - `struct cpu` - defines the CPU of the computer
- `main.c` - contains a tiny shell with commands for manipulating the states of processes. A few sample commands.
  - `fork` - fork a process
  - `exit` - exit a process
  - `sleep` - put a process to sleep
  - `wait` - a parent process waits for a child to exit
  - `schedule` - change the currently executing process to the next
  - `ps` - show all of the current processes
- `Makefile` - a makefile to build the program, which is procprog.

Study

Study Chapters 7, 8, and 9 of OSTEP. Study Chapters 0 and 1 of xv6-book-rev11.pdf to familiarize yourself with Xv6. There are some details in this reading that are not applicable to the code provided; however, the details are good learning.

Preliminary Work

Build and run the program and experiment with the various commands. You run the program via the following.

```
$ ./procprog
```

Which results in a prompt where you can enter commands. The prompt contains your username. The following is a sample set of commands.

```
$ ./procprog
gusty@shell 1> fork
pid: 1 forked: 2
gusty@shell 2> fork
pid: 1 forked: 3
gusty@shell 3> fork
pid: 1 forked: 4
gusty@shell 4> ps
pid: 1, parent: 0 state: RUNNING
pid: 2, parent: 1 state: RUNNABLE
pid: 3, parent: 1 state: RUNNABLE
pid: 4, parent: 1 state: RUNNABLE
gusty@shell 5> fork 3
pid: 3 forked: 5
gusty@shell 6> ps
pid: 1, parent: 0 state: RUNNING
pid: 2, parent: 1 state: RUNNABLE
pid: 3, parent: 1 state: RUNNABLE
pid: 4, parent: 1 state: RUNNABLE
pid: 5, parent: 3 state: RUNNABLE
gusty@shell 7> schedule
Scheduler selected pid: 2
gusty@shell 8> ps
pid: 1, parent: 0 state: RUNNABLE
pid: 2, parent: 1 state: RUNNING
pid: 3, parent: 1 state: RUNNABLE
pid: 4, parent: 1 state: RUNNABLE
pid: 5, parent: 3 state: RUNNABLE
gusty@shell 9> wait 3
pid: 3 has children, but children still running.
```

```
gusty@shell 10> ps
pid: 1, parent: 0 state: RUNNABLE
pid: 2, parent: 1 state: RUNNING
pid: 3, parent: 1 state: SLEEPING
pid: 4, parent: 1 state: RUNNABLE
pid: 5, parent: 3 state: RUNNABLE
gusty@shell 11> schedule
Scheduler selected pid: 1
gusty@shell 12> ps
pid: 1, parent: 0 state: RUNNING
pid: 2, parent: 1 state: RUNNABLE
pid: 3, parent: 1 state: SLEEPING
pid: 4, parent: 1 state: RUNNABLE
pid: 5, parent: 3 state: RUNNABLE
gusty@shell 13> timer 4
Scheduler selected pid: 2
Scheduler selected pid: 1
Scheduler selected pid: 2
Scheduler selected pid: 1
```

When `procprog` starts running, it creates an initial process, which is part of the kernel. This process has pid of 1 and parent id of 0. This process is marked `RUNNING`. The fork command without an argument creates a process that is a child of the initial process. The initial three fork commands create three processes that are children of the initial process. The fork command with a PID creates a process that is a child of the PID. The fork 3 command creates a process that is a child of PID 3. The ps command shows the processes and their state. The schedule command terminates the current running process and starts running the next. The schedule command calls the scheduler function to accomplish this scheduling. The current schedule algorithm finds the first `RUNNABLE` process in the proc array. This algorithm is not very good. It alternates between two processes. You can observe this in the above sequence of commands.

The commands consist of the following

- `fork [pid]` - create a new process. The parent is either PID 1 or pid.
- `schedule` - replace the currently executing process with the next one to execute.
- `ps` - display the current collection of processes and their state, which can be `EMBRYO`, `RUNNABLE`, `RUNNING`, `SLEEPING`, and `ZOMBIE`.
- `sleep pid channel` - put process pid to sleep waiting on channel
- `wake channel` - wakeup all processes sleeping on channel
- `wait` - a parent process waits for its children to terminate
- `exit pid` - exit pid
- `timer N` - call schedule N times. You can think of this as a timer interrupt happening, which allows the OS to schedule a new process. This allows the scheduling algorithm to execute on each quantum.

Assignment

Create two scheduling algorithms. You must implement the Linux Completely Fair Scheduler. The second scheduling algorithm can be a round robin or a ticket-based scheduler (either lottery or stride). You will have to study the existing code base before you can design and implement your algorithms. You will most likely have to modify some of the underlying data structures.

Submissions

1. Demonstrate your solutions to me. Your demonstration must explain your design, execute your test cases, and explain how your test cases prove your code is correct.
2. Submit a design of your implementation. Your design must include (a) diagrams of the data structures and how they are related (b) pseudo code of your algorithms and how they affect the data structures, and (c) which files in the code base are updated. I suggest that you create this before trying to modify the code.
3. Submit a description of your test cases. I should be able to read your test case descriptions and understand them.
4. Submit the code for your two scheduling algorithms. You do not have to submit your entire code base. The code you submit must include files that you modified. Be sure to include comments in the code that describes your modifications. For example any .h files in which you changed the underlying data structures and to the `scheduler` function in `proc.c`.