Schedule Project

You may discuss this Schedule Project description with others to understand the problem. You may use the Internet for point-specific questions. For example, you may search the Internet for how to call `fork`. You **may not** use the Internet to discover scheduler code that is copied or mimicked into your 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.

Use the file `ScheduleProjectAnswers.docx` to answer questions in this project.

Introduction

In the Schedule Project, you modify code to implement two scheduling algorithms - see Section Assignment below for the specific algorithms. You begin with code that has been extracted from the Xv6 code base. The code has been modified and built into a stand-alone executable program. The program mimics the execution of processes in various states - `EMBRYO`, `RUNNABLE`, `RUNNING`, `SLEEPING`, and `ZOMBIE`. A process does not really execute when it is marked as `RUNNING` - we just pretend it is running. Actually 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 `uint` (`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.
    - `fork` - function the forks processes.
    - `scheduler` - function that implements the scheduling algorithm.
    - `kill` - function that kills processes.
- `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 calls `pinit` and `userinit` (in `proc.c`) to create an initial process, which is part of the kernel - like `kernel_proc` in FirstLab. This initial process has `pid` of 1 and parent id (`ppid`) of 0. This process is marked `RUNNING`. The commands are processed by `main.c`. The `fork` command without an argument creates a process that is a child of the initial process. The `pid` is incremented each time a process is forked. 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 the process with `pid` 3. The `ps` command shows the processes and their state. The `schedule` command performs a context switch. It terminates the current running process (i.e., the process with state `RUNNING`), performs the scheduling algorithm to select the next process to run, and starts running the selected process. The `schedule` command calls the `scheduler` function (in `proc.c`) to accomplish this scheduling. The current schedule algorithm finds the first `RUNNABLE` process in the `struct proc proc` array, which is a member of the `struct ptable`. - a table of all procs - those `RUNNABLE`, `SLEEPING`, `RUNNING`, etc. This algorithm is not very good. It alternates between two processes and starves any processes further in the array that are `RUNNABLE`. You can observe this in the above sequence of commands, where it alternated between processes with `pid` 1 and `pid` 2.

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.

Running, Quantum, Time Slice, and Context Switch

This assignment allows you to study and understand real OS code that manages processes. When an OS performs a context switch, it stops the currently running process and starts the next. A context switch is a low-level function that performs two actions.

1. Saves the CPU registers of the currently running process in its `struct proc`, which has a member `struct context` to hold the registers.
2. Places the register values in the `struct proc`'s `struct context` from the scheduled process into the CPU registers.

These actions involve assembly code to access the registers. Figure 1 shows ARM assembly code.

Our Schedule Project does not actually run processes, thus it does not contain the low-level context switch function; however, each time you call `scheduler`, you have finished one quantum (or time slice). When you are implementing your Linux Completely Fair Scheduler and other chosen scheduler, you update values such as vruntime and total tickets.

```
.global swtch
swtch:
       push {lr}  /* save the return address
*/
       push {lr}
       /* save old callee-save registers */
       push {r12}
       push {r11}
       push {r10}
       push {r9}
       push {r8}
       push {r7}
       push {r6}
       push {r5}
       push {r4}

       /* switch stacks */
       str sp, [r0]
       mov sp, r1

       /* load new callee-save registers */
       pop {r4}
       pop {r5}
       pop {r6}
       pop {r7}
       pop {r8}
       pop {r9}
       pop {r10}
       pop {r11}
       pop {r12}

       /* return to previously saved pc */
       pop {lr}
       pop {pc}
```

Figure 1. ARM Assembly for Context Switch

Assignment

Create two scheduling algorithms. You must implement the Linux Completely Fair Scheduler. Designing good data structures is key to creating a good design. To do this, you must create a diagram of the existing data structures, which you will modify to show your data structures design and include in your submission for 1a. For example, (1) you may decide to retain the array design of all procs in `ptable` or (2) you may decide to create separate linked lists for `RUNNABLE` and `SLEEPING`. You must create pseudo code for your scheduling algorithms. The pseudo code must reference information in your data structures design. For example, if you implement a linked list of `struct proc`s, your pseudo code must show traversing the linked list. Your pseudo code algorithm design is included in your submission for 1b. The second scheduling algorithm can be a lottery, stride or MLFQ scheduler. You will have to study the existing code base before you can design and implement your algorithms. You will have to modify some of the underlying data structures. You do not have to merge the two scheduling algorithms into one program. You can create two programs, each in their own directory. If you like, you can place both schedulers in the same program, with some mechanism to select between them.

Submissions

Be sure your submission uses the `ScheduleProjectAnswers.docx` to answer the questions so I can easily decipher your answers in your submission. Realize that you have two solutions.

1. Place the design of your implementation in this item. You should study the problem and create a design before trying to modify the code.
    a. Place diagrams of the data structures and how they are related here.
    b. Place pseudo code of your algorithms and how they affect the data structures here.
    c. Place a list of source files in the code base that you updated.
2. Place your test cases here. Your collection of test cases should demonstrate you have tested the entire program. You will have a collection of test cases for each of you schedulers. For each test case, first describe the objective of your test case. Then insert a copy/paste of you applying your test cases to your program. The copy/paste must demonstrate that actual output achieved the described objective.
3. On Canvas submit a zip file with the code (.c, .h, and makefile) for your two scheduling algorithms. You must submit all code, even code that you did not modify. Your modified source files must include comments in the code that describes your modifications. If you implement two programs - one for each scheduler, your submitted zip file must unzip into two directories - one for each scheduler. You may include your .docx file in either directory.