CS 550 Operating Systems, Spring 2023
Project 2 (PROJ2)

Out: 2/26/2023, SUN
Due date: 3/25/2023, SAT 23:59:59

There are two parts in this project: coding and Q&A. In the first part, you will implement a functionality which changes outcomes of race conditions after fork in xv6, as well as a priority scheduler for xv6. In the second part, you will need to answer the questions about xv6 process scheduling.

1 Baseline source code

You will be working on the baseline code that needs to be cloned/downloaded from your own private GitHub repository. Please make sure you read this whole section, as well as the grading guidelines (Section 5), before going to the following link at the end of this section.

- Go to the link at the end of this section to accept the assignment.
- **Work on and commit your code to the default branch of your repository. Do not create a new branch. Failure to do so will lead to problems with the grading script and 5 points off of your project grade.**

Assignment link: `https://classroom.github.com/a/Sq42TED9`

(Continue to next page ...)

2 Process scheduling in xv6 - coding (70 points)

2.1 Race condition after **`fork()`** (20 points)

As we discussed in class, after a `fork()`, either the parent process or the child process can be scheduled to run first. Some OSes schedule the parent to run first most often, while others allow the child to run first mostly. As you will see, the xv6 OS by default schedules the parents to run first after `fork()`s mostly. In this part, you will change this race condition to allow the child process to run first mostly after a `fork()`.

2.1.1 The test driver program and the expected outputs

The baseline code has included a test driver program `fork_rc_test` that allows you to check the race condition after a `fork()`. The program is implemented in `fork_rc_test.c`. In the program, the parent process repeatedly calls `fork()`. After `fork()`, the parent process prints string a "parent" when it gets the, and the child process prints a string "child" and exits.

The program takes one argument to indicate whether parent-first or child-first policy is adopted. Here is the usage of the program

```
$ fork_rc_test
Usage: fork_rc_test 0|1
        0: Parent is scheduled to run most often
        1: Child is scheduled to run most often
```

When calling the program using "`fork_rc_test` 0", the parent-policy (the default) is used, and you will see output like:

```
$ fork_rc_test 0

Setting parent as the fork winner ...

Trial 0:  parent!  child!
Trial 1:  parent!  child!
Trial 2:  parent!  child!
Trial 3:  paren child! t!
Trial 4:  parent!  child!
Trial 5:  child! parent!
...
Trial 45:  parent!  child!
Trial 46:  parent!  child!
Trial 47:  parent!  child!
Trial 48:  child!  parent!
Trial 49:  parent!  child!
```

When calling the program using "`fork_rc_test` 1", the child-first (the one you're gonna implement) is used. If things are done correctly, here is what the expected output of the test driver program look like:

```
$ fork_rc_test 1

Setting child as the fork winner ...

Trial 0:  child!  parent!
Trial 1:  child!  parent!
Trial 2:  child!  parent!
Trial 3:  child!  parent!
Trial 4:  parent! child!
Trial 5:  child!  parent!
...
Trial 45:  child!  parent!
Trial 46:  parent!  child!
Trial 47:  child!  parent!
Trial 48:  child!  parent!
Trial 49:  child! parent!
```

2.1.2 What to do

(1) Figure out what to do to change the race condition to child-first after a fork.

(2) Write a system call that can control whether parent-first or child-first policy is used.

(3) Implement a user space wrapper function for the above system call, and declare it in "user.h". This wrapper function's prototype should be

```
void fork_winner(int winner);
```

This function takes one argument: if the argument is 0 (i.e., `fork_winner(0)`), the parent-policy (xv6 default) is used; if this argument is 1 (i.e., `fork_winner(1)`), the child-first policy (the one you implemented) is used.

Note: for the proper compilation of the base code, the `fork_rc_test` program has a stub implementation for the wrapper function above. Remember to comment it out after developing your own solution.

Tips: understanding the code for `fork` and CPU scheduling is the key part. The actual code that changes the race condition (excluding the system-call-related code) can be less than 3 LOC.

(Continue to next page ...)

2.2 Priority-based CPU scheduling (50 points)

The default scheduler of xv6 adopts a round-robin (RR) policy. In this part, you are going to implement a simple priority-based policy.

2.2.1 The priority-based policy

The policy has 3 priorities (1, 2, and 3), with 3 being the highest, 1 being the lowest, and 2 being the default. The rules are simple:

- The process with the highest priority always gets the CPU.
- If there are multiple processes with the same priority, RR is used.
- Every process created has a default priority of 2.

You neither need to break the queue into multiple, nor consider things like priority boost or time accounting.

2.2.2 The test driver program

To help you implement and debug, a scheduling tracing functionality has been added to the base code. When this tracing functionality is enabled, the kernel prints the PID of the currently running process every time before the CPU is transferred back to the scheduler in the timer interrupt handler. With this scheduling tracing functionality, you can see the sequence of processes that the scheduler schedules.

A system call (`sys_enable_sched_trace()`) and its corresponding user space wrapper function (`void enable_sched_trace(int)`) have been added in the base code to enable/disable the scheduling tracing functionality. A call of "`enable_sched_trace(1)`" will enable the tracing, and a call of "`enable_sched_trace(0)`" will disable it.

The baseline code has a simple test driver program `schdtest` to illustrate how to use the above scheduling tracing functionality. The program is implemented in `schdtest.c`. In the program, the parent process first enables the scheduling tracing, and forks a child process. Then both the parent and the child preform some busy computation, and child's computation roughly take three times of parent's. When you run this program, you will see outputs like:

```
6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7
6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7 - 6 - 7
6 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7
7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7 - 7
7 -
```

where 6 is the PID of the parent process and 7 is the PID of the child process. From the output we can see that out of the 81 scheduling decisions, the parent was scheduled 21 times, and the child was scheduled 60 times, which matches the behavior of the code (i.e., child runs 3 times long as the parent does).

You will need to replace the example test code in `schdtest.c` with your own test code to properly test your scheduler implementation.

2.2.3 The test cases

When grading, we will be testing the following four test cases:

- Set the scheduling policy to the default, fork processes, and check the process scheduling output. (The only thing you need to do to pass the test case is correctly implementing the system call and the corresponding user space wrapper function that set the type of scheduler.)

- Set the scheduling policy to priority-based, fork processes and set priority for the processes such that each priority has one process, and check the process scheduling output. The correct output should be that the processes are scheduled sequentially according to their priorities.

- Set the scheduling policy to priority-based, fork processes and set priority for the processes such that each priority has multiple processes, and check the process scheduling output. The correct output should be that the processes with different priorities are scheduled sequentially according to their priorities, and processes with the same priority are scheduled in a round-robin manner.

- Set the scheduling policy to priority-based, fork processes and set priority for the processes such that each priority has multiple processes. After the processes with the top two priorities finish executing, a new process with the highest priority (i.e., priority 3) is created. The correct output should be that, in addition to the correct output similar to the previous test case, the new process should be scheduled to run before the remaining processes, which have the lowest priority.

2.2.4 What to do

(1) Implement the functionality that allows user program to set the type of scheduling policy.

- Write a system call that can control whether the default policy (RR) is used or the priority-based policy is used.
- Write the corresponding system call user space wrapper function, and declare it in "user.h". The wrapper function's prototype should be:

```
void set_sched(int);
```

This user-level wrapper function takes one integer argument: If the argument is 0, the default policy is adopted. If the argument is 1, the priority-based policy is used.

(2) Implement the functionality that allows user program to set the priority of a process.

- Write a system call to set the priority of a given process.
- Write the corresponding system call user space wrapper function, and declare it in "user.h". The wrapper function's prototype should be:

```
void set_priority(int pid, int priority);
```

This user-level wrapper function takes two argument. The first argument is the pid of the process whose priority is being set. The second argument is the new priority.

(3) Implement the priority-based scheduling policy, and design your test code to test your implementation. You do not need to submit your test code.

Note: Your implementation should keep the patch that fixes the always-100% CPU utilization problem. If your code causes the problem to re-occur, 10 points off (see the 4th point in the "Grading" section for details).

(Continue to next page ...)

3 Process scheduling in xv6 - Q&A (30 points)

Answer the following questions about process scheduling implementation.

Q1: (10 points) Does xv6 kernel use cooperative approach or non-cooperative approach to gain control while a user process is running? Explain how xv6's approach works using xv6's code.

Q2: (10 points) After `fork()` is called, why does the parent process run before the child process in most of the cases? In what scenario will the child process run before the parent process after `fork()`?

Q3: (10 points) When the scheduler de-schedules an old process and schedules a new process, it saves the context (i.e., the CPU registers) of the old process and load the context of the new process. Show the code which performs these context saving/loading operations. Show how this piece of code is reached when saving the old process's and loading the new process's context.

Key in your answers to the above questions with any the editor you prefer, export them in a PDF file named "xv6-sched-mechanisms.pdf", and submit the file to the assignment link in Brightspace.

(Continue to next page ...)

4 Submit your work

Once your code in your GitHub private repository is ready for grading, submit a text file named "DONE" (and the the previous "xv6-syscall-mechanisms.pdf") to the assignment link in Brightspace.

If you have referred to any form of online materials or resources when completing this project (code and Q&A), please state all the references in this "DONE" file. Failure to do so, once dectected, will lead to zero point for the entire project, and possibly further penalty depending on the severity of the violation.

Suggestion: Test your code thoroughly on a CS machine before submitting.

(Continue to next page ...)

5 Grading

The following are the general grading guidelines for this and all future projects.

(1) **The code in your repository will not be graded until a "DONE" file is submitted to Brightspace**.

(2) The submission time of the "DONE" file submitted to Brightspace will be used to determine if your submission is on time or to calculate the number of late days. Late penalty is 10% of the points scored for each of the first two days late, and 20% for each of the days thereafter.

(3) If you are to compile and run the xv6 system on the department's remote cluster, remember to use baseline xv6 source code provided by our GitHub classroom. Compiling and running xv6 source code downloaded elsewhere can cause 100% CPU utilization on QEMU.

Removing the patch code from the baseline code will also cause the same problem. So make sure you understand the code before deleting them.

If you are reported by the system administrator to be running QEMU with 100% CPU utilization on QEMU, 10 points off.

(4) If the submitted patch cannot successfully patched to the baseline source code, or the patched code does not compile:

```
1   TA will try to fix the problem (for no more than 3 minutes);
2   if (problem solved)
3     1%-10% points off (based on how complex the fix is, TA's discretion);
4   else
5     TA may contact the student by email or schedule a demo to fix the problem;
6     if (problem solved)
7       11%-20% points off (based on how complex the fix is, TA's discretion);
8     else
9       All points off;
```

So in the case that TA contacts you to fix a problem, please respond to TA's email promptly or show up at the demo appointment on time; otherwise the line 9 above will be effective.

(5) If the code is not working as required in the project spec, the TA should take points based on the assigned full points of the task and the actual problem.

(6) Lastly but not the least, stick to the collaboration policy stated in the syllabus: you may discuss with you fellow students, but code should absolutely be kept private. Any kind of cheating will result in zero point on the project, and further reporting.