CS 350 Operating Systems, Spring 2024
Programming project 2 (PROJ2)

Out: 10/07/2024, MON
Due date: 10/20/2024, SUN 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 the outcome of race conditions after fork() in xv6, and a stride scheduler for xv6. In the second part, you will need to answer the questions about xv6 process scheduling.

This is a group project, and is expected to be completed with your group members. You can choose to divide and conquer the assignment, or work as a group to solve problems together. Regardless, each group member is responsible for submitting their own code on Brightspace.

This project contributes 6.25% toward the final grading.

1 Baseline source code

For this and all the later projects, you will be working on the baseline code that needs to be downloaded from Brightspace. This code should be familiar to you from project 1. It is the baseline XV6 version from this course with some slight differences to facilitate this assignment. Please extract the zip file provided on Brightspace and that code (not your old XV6 code from prior assignments) to complete this assignment.

(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. The actual code that changes the race condition (i.e., not including the system-call-related code) can be less than 3 LOC.

(Continue to next page ...)

2.2 Implementing stride scheduling in xv6 (50 points)

The default scheduler of xv6 adopts a round-robin (RR) policy. In this part, you are going to implement the stride scheduler, which is a type of proportional share scheduler, for xv6.

2.3 The stride scheduling policy

- Each process will maintain two numbers used for scheduling: stride and pass.
- The whole system has 100 tickets in total (i.e., `STRIDE_TOTAL_TICKETS = 100`).
- Whenever a new process is added to or an existing process removed from the system, the 100 tickets are evenly assigned to all active processes (i.e., RUNNABLE and RUNNING processes). Formally, when a new process is added to or an existing process is removed from the system, the tickets of each active process p ($ticket_p$) is calculated as

$$ticket_p = \lfloor \frac{\texttt{STRIDE\_TOTAL\_TICKETS}}{N} \rfloor \tag{1}$$

  where N is the number of active processes. At the same time, the *pass* values of all the active processes should be reset (e.g., to 0).

  For example, if there is only one active process (p_A), p_A should have all the 100 tickets. When a new process (p_B) is added to the system, p_A and p_B should have 50 tickets each.

  Another example is that, if there are four active processes (p_A, p_B, p_C, p_D) and p_A exits, then the 100 tickets should be evenly redistributed to the remaining three active processes. In this case, p_B, p_C, and p_D should have 33 tickets each.
- The stride value of each process p ($stride_p$) is fixed, and calculated as

$$stride_p = \lfloor \frac{\texttt{STRIDE\_TOTAL\_TICKETS} \times 10}{ticket_p} \rfloor \tag{2}$$

  where $ticket_p$ is the number of tickets that p has.
- At the time the scheduler needs to make a scheduling decision, the active process with the lowest *pass* value gets scheduled. When a process is scheduled, its pass value should be increased by the stride value of said process:

$$pass_p = pass_p + stride_p \tag{3}$$

  If there are multiple processes with the same smallest *pass* value, the one with the smallest pid gets scheduled.

  This allows processes to encode priority. Processes with more tickets (and therefore a lower stride value) will run for longer than processes with less tickets, but not forever! Because pass values gradually increase, the stride scheduler includes implicit aging.
- How can priority change if all processes are given equal numbers of tickets? In this stride scheduler, a process cannot increase its own tickets. But it can transfer its own tickets to another process (details later). This is a variation of niceness, as discussed in our course material.

2.3.1 The test driver program and test cases

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 contains a test driver program which uses and above scheduling tracing functionality. The program is implemented in `schdtest.c`. The test driver program also provides the five test cases which will be in the grading. The five test cases are briefly introduced in the following (read the code the understand what exactly each of the test cases does).

- Test case 1: the scheduler is set to the default one (i.e., RR). Three child processes are created by the parent. The expected output is that four processes are scheduled in a round robin manner. (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.)

- Test case 2: the scheduler is set to the stride scheduler. Two child processes are created by the parent. Since all the processes evenly share the 100 tickets (33 each in this case), the expected output is that thre three processes are scheduled in a round robin manner.

- Test case 3: the scheduler is set to the stride scheduler. This test case tests if your implementation of `transfer_tickets()` returns values correctly (see the spec of this function in the next section).

- Test case 4: the scheduler is set to the stride scheduler. The parent process forks a child process, and transfers half of its tickets to the child. The expected output is that, since now the parent has 25 tickets and the child has 75 tickets, the chance that the child gets scheduled should be three times of that of the parent.

- Test case 5: the scheduler is set to the stride scheduler. The parent process forks a child process, and transfers all but one of its tickets to the child. The expected output is that after the tickets are transferred, the child gets to run all the way to finish before the parent is get scheduled .

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 stride scheduling 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 stride scheduling policy is used.

(2) Implement the functionality that allows a user process to get the number of its tickets.

- Write a system call that allows the calling process to get the number of tickets owned by a particular process.
- Write the corresponding system call user space wrapper function, and declare it in "user.h". The wrapper function's prototype should be:

```
int tickets_owned(int pid);
```

This user-level wrapper function returns the number of tickets owned by the process whose PID is passed as the argument to this function.

(3) Implement the functionality that allows a user process to transfer its own tickets to another process.

- Write a system call that allows the calling process to transfer its own tickets to another process.
- Write the corresponding system call user space wrapper function, and declare it in "user.h". The wrapper function's prototype should be:

```
int transfer_tickets(int pid, int tickets);
```

This user-level wrapper function takes two arguments: the first argument (`pid`) is the PID of the recipient process, and the second argument is the number of tickets that the calling processes wishes to transfer. For the calling process p, this number cannot be smaller than 0, and cannot be larger than $(ticket\_p - 1)$, where $ticket_p$ is the number of the tickets owned by the process p before the ticket transfer.

Regarding the return value:

- On success, the return value of this function should be the number of tickets that the calling process has after the transfer.
- If the number of tickets requested to transfer is smaller than 0, return -1.
- If the number of tickets requested to transfer is larger than $ticket\_p - 1$, return -2.
- If the recipient process does not exist, return -3.

(4) Implement the stride scheduling policy, remove the stub functions defined at the beginning of `schdtest.c` (by simply removing the "`STUB_FUNCS`" macro definition), and test your implementation.

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 xv6 and Scheduling.

Q1: (10 points) Does the 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 the Brightspace.

(Continue to next page ...)

4 Submit your work

Please zip your entire xv6 directory and submit it to the assignment link on Brightspace along with the answers to the questions in this assignment.

Please also include a PROJ2.txt file explaining how you and your group members worked on this project. Please comment on the work that each of you completed.

You can include the following info in this file:

- The status of your implementation (especially, if not fully complete).
- A description of how your code works, if that is not completely clear by reading the code (note that this should not be necessary, ideally your code should be self-documenting).
- Possibly a log of test cases which work and which don't work.
- Any other material you believe is relevant to the grading of your project.

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 submission time of your files submitted to the Brightspace will be used to determine if your submission is on time or to calculate the number of late days. Please consult the syllabus for our course late assignment policy.

(2) If you are to compile and run the xv6 system on the department's remote cluster, remember to use baseline xv6 source code provided on Brightspace. 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.

(3) 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.

(4) 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.

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