CS 550-04 Operating Systems, Spring 2017
Programming Project 1

Out: 2/12/2017, Sun.
Due date with bonus: 2/23/2017, Thur. 23:59:59 (10% bonus points of the points scored)
Final due date: 3/4/2017, Sat. 23:59:59

In this project, you are going to practice on system calls and CPU scheduling by hacking the xv6 OS.

1 Baseline xv6 source code

Download the xv6 baseline source code of the Project 1 from here: `https://drive.google.com/open?id=0B5j-vGKxJzqxc0NmWkl4Sm01S0k`. Note that this baseline code is different from the one of the previous warm-up project. You will need to work on the correct baseline code for this project.

If you need to work on a department's machine, remember to patch the Makefile (see the spec of the project 0 for details).

2 PART I: system call that shuts down the machine (10 points)

As you may have noticed, the original xv6 system doesn't have a "`shutdown`" command to properly turn off the emulated machine. In the baseline code, a new file named `shutdown.c` implements the user space command `shutdown`. The missing parts for the `shutdown` command (and thus your jobs for this part) are the implementation of the system call that can shut down the machine and the corresponding system call user-level wrapper function, which is called by the `shutdown` command program (i.e., `shutdown.c`). The system call user-level wrapper function should have following prototype:

```
void shutdown(void);
```

and should be declared in "user.h". In `shutdown.c`, a stub implementation of the wrapper function is given to ensure the correct compilation of the baseline code. Remember to deactivate the stub function after having you own implementation.

After correctly implementing the missing parts, running the command `shutdown` should shut down the QEMU VM.

2.1 Tips

- To shut down the virtual machine in your system call, you only need one line of code:

  ```
  outw(0xB004, 0x0|0x2000);
  ```

- Reading and understanding how the existing system calls and the corresponding user-level wrapper functions are implemented will be helpful.
- Understanding the mechanism is important. You may be requested to explain how things work in xv6 in midterm/final exams.

3 PART II: 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()`.

3.1 The test driver program and the expected outputs

The baseline code has included a test driver program `proj1_forktest` that allows you to check the race condition after a `fork()`. The program is implemented in `proj1_forktest.c`. In the program, the parent process repeatedly calls `fork()`. After `fork()`, the parent process prints string a "parent" when it gets the CPU, 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

```
$ proj1_forktest
Usage: proj1_forktest 0|1
0: Parent is scheduled to run most often
1: Child is scheduled to run most often
```

When calling the program using "proj1_forktest 0", the parent-policy (the default) is used, and you will see output like:

```
$ proj1_forktest 0

Setting parent as the fork winner ...

CS550 proj1 fork test ==>

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 "proj1_forktest 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:

```
$ proj1_forktest 1

Setting child as the fork winner ...

CS550 proj1 fork test ==>
```

```
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!
```

3.2 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 `proj1_forktest` program has a stub implementation for the wrapper function above. Remember to comment it out after developing your own solution.

3.3 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 just contain several LOC.

4 PART III: proportional-share process scheduler - stride scheduling (70 points)

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

4.1 The stride scheduling policy

- 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 calculated as

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

  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 (refer to our course material for the details of the stride scheduling algorithm).

  If there are multiple processes with the same smallest *pass* value, the one with the smallest pid gets scheduled.
- A process cannot increase its own tickets. But it can transfer its own tickets to another process (details later).

4.2 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 given 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 `proj1_schdtest` to illustrate how to use the above scheduling tracing functionality. The program is implemented in `proj1_schdtest.c`. In the program, the parent process first enables the scheduling tracing, then it forks a child process, which does a busy computation for a while. When you run this program, you will see outputs like:

```
[6][6][7][6][7][6][7][6][7][6][7][6][7][6][7][6][7][6][7][6][7][6]
```

where 6 is the PID of the parent process and 7 is the PID of the child process. You will need to replace the example test code in `proj1_schdtest.c` with your own test code to properly test your scheduler implementation.

4.3 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 it has.
- 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(void);
  ```

  This user-level wrapper function takes no argument, and returns the calling process's ticket number.

(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 $(ticket\_p - 1)$.

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, and design your test code to test your implementation based on the test cases described next. You do not need to submit your test code.

Note: Your implementation should keep the patch that fixes the always-100% CPU utilization problem (hint: to understand this patch, pay attention to how the local variable "`ran`" in `scheduler(void)` (in proc.c) is used). If your code causes the problem to re-occur, 10 points off.

4.4 The test cases

When grading, we will be testing the following test scenario:

- Set the scheduling policy to be 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 be stride scheduling, fork processes, and check the process scheduling output (no ticket transfer). The correct output should be round-robin-like as all active processes have the same ticket value.
- Set the scheduling policy to be stride scheduling, fork processes, make the parent process transfer tickets to its child processes, and check the process scheduling output. The correct output should be that the probability of each process getting scheduled is inversely-proportional to its stride value. Note that the transfer can be done at any time when the parent process is running.
- Set the scheduling policy to be stride scheduling, fork processes, let parent/child processes run for a while, and then let the parent processes fork a new process, and check the process scheduling output. The correct output should be that the tickets are redistributed to all active processes when the new process is added to the system.
- Check the return value of the `transfer_tickets` function.

5 Submit your work

- **Prepare the `diff` patch**: before generating the diff patch, remove all the compiled object files by running:

```
make clean
```

The diff patch file should be named "`proj1.your-bu-id.patch`". Run the following to generate the file:

```
diff -uNr DIR_XV6_BASE DIR_XV6_WORKING >proj0.your-bu-id.patch
```

where `DIR_XV6_BASE` is the path to the baseline xv6 code, and `DIR_XV6_WORKING` is the path to your xv6 working directory. Refer back to the specification of Project 0 for more details. If your BU email is "`abc@binghamton.edu`", the name of the patch should be "`proj1.abc.patch`".

- **Log your work**: besides the files needed to build your project, you must also include a README file which minimally contains your name and B-number. Additionally, it can contain the following:
  - 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.
- **Compress the files**: use the "`zip`" command in a Linux machine to compress the diff patch and your README file into a ZIP file. Name the ZIP file based on your BU email ID. For example, if your BU email is "abc@binghamton.edu", then the zip file should be named "proj0_abc.zip".
- **Submission**: submit the ZIP file to MyCourses before the deadline. Note that submissions by the bonus due date will receive 10% bonus point of the final points scored. You are allowed to submit again after the bonus due date, in which case no bonus will be granted.

Suggestion: since the we will grade on a CS machine, test your code thoroughly (i.e., patch the baseline source using the to be submitted diff patch, compile, run, and test) on a CS machine before submitting.

6 Grading

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

(1) 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.

(2) Compress your submission using the zip command in a Linux machine. If your zip file cannot be uncompressed, 5 points off.

(3) If the submitted ZIP file or the patch file inside is not named as specified above (so that it causes problems for TA's automated grading scripts), 10 points off.

(4) If the submitted diff patch contains object files, 10 points off.

(5) 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 course website. Compiling and running xv6 source code downloaded elsewhere can cause 100% CPU utilization on QEMU. If you are reported by the system administrator to be running QEMU with 100% CPU utilization on QEMU, 10 off the pending project.

(6) 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.

(7) 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.

(8) 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.