OS LAB

Navneet Agarwal - 140100090
Ritwick Chaudhry – 14D070063

1. Additions to the scheduler:

a. The **setprio(int n)** system call is used to set the priority of the process. (** Note that a process' priority can be set only when that process is running when that process is running. This is in accordance with the problem statement).
b. The **getprio()** system call when called within the process returns that process' priority.
c. **STRICT PRIORITY SCHEDULING** which just transfers control to the highest priority runnable process (***Can lead to starvation*** where the highest priority process is always running even on yield)

2. Data Structures:

a. An integer 'priority' in the struct proc to store the priority of each process in its PCB

3. Changes to Code:

a. proc.c
   i. The default priority = 0 and is set in the **allocproc**(in the file proc.c) function which is called when any new process is forked.
   ii. The **Scheduler** is changed(explained below)

b. sysproc.c
   i. The functions for both the new system calls setprio and getprio is made(functions explained below in detail)

c. usys.S
   i. **SYSCALL(setprio)** and **SYSCALL(getprio)** are added in the file to define the syscalls

d. user.h
   i. Declaration of the two system calls is done in the file

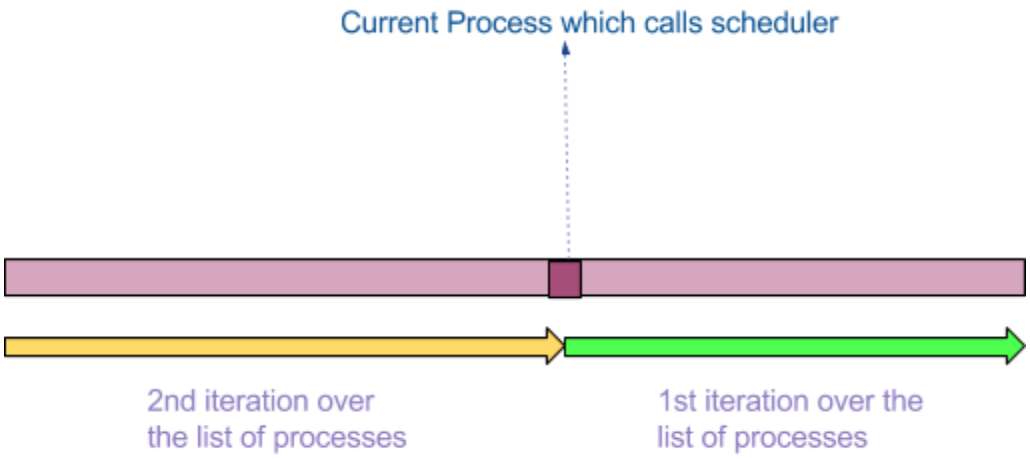
e. syscall.h
   i. Define the system call numbers for the two new system calls

f. syscall.c
   i. Extern functions for the two new system calls are declared
   ii. The two system calls are added in the static array containing all the system calls.

4. Scheduler Design

The scheduler works in the following way:

1. The list of processes is iterated from the current process + 1 to the last process in the array and the maximum priority process and its priority is maintained
2. Then the list of process is iterated from ptable.proc to the current process + 1 and the maximum priority process is maintained (maximum over all processes, not separately over the two regions)



3. Note that if there are multiple processes with the highest priority, then the one which is next closest to the process which called scheduler
4. If there is no process found then the scheduler releases the ptable.lock, re-enables the interrupts (for a short while) and then reacquires the ptable.lock. Then the scheduler starts to iterate again from the start of the list.
5. This process is then repeated till a runnable process is found
6. The default priority = 0 and is set in the **allocproc**(in the file proc.c) function which is called when any new process is forked.
7. It will find the next runnable process with the maximum priority if a process we want to schedule blocks before its quantum finishes
8. Code is safe on multiple cores because of the acquiring and releasing of ptable.lock in the scheduler

5. setprio and getprio

setprio(int n) - It sets the priority of a process to the specified value (n).
getprio() - It returns the priority value of the process in which we call the function

6. Testcases

Testcase1 details:

This testcase shows the case of starvation. What we have done is run three forked processes with different priority and have run infinite loops in them. The parent is of the highest priority among all processes. So it may happen that the second CPU core takes up the minimum priority process initially but after a timer interrupt both the cores would be taking the processes with the highest and the second highest priority(as the parent would be in a blocked state because of wait()) leading to starvation of minimum priority process. Hence both the cores will always run the 2 higher priority children of the three children forked. Note that the parent won't run even though it has the highest priority because it made a blocking wait call and hence isn't runnable.

Testcase2 details:

In this testcase we have run the loops for finite but reasonably large number, for the three forked processes. The parent is of the highest priority among all processes. As a result what might happen is the parent running on first core, the first forked process running on the second core, but once a timer interrupt comes or maybe because of the initial sleep the core running the first child takes up the process with maximum priority. As a result when the parent executes the blocking wait call the cores would be working on the top two priority processes. And only when one of them ends the minimum priority process continues execution on the core that finished the execution of one of the children. Again, note that the parent won't run even though it has the highest priority because it made a blocking wait call and hence isn't runnable.

Testcase3 details:

In this testcase we have created four forked processes and written to different files in all of the processes. What is observable from the output is that the high priority processes tend to have a lower completion time (in terms of clock cycles) compared to the lower priority processes.

What actually happens is that write is a blocking call and it causes a lot of context switching. And what should happen is on an average the high priority processes should execute in a lesser time compared to low priority ones, which is the case here. Note that whenever there is a context switch and even one of the two higher priority processes is **runnable**, the OS switches to that thus increasing the time of execution of the lower priority process and decreasing the time of execution of the higher priority processes. Again, note that the parent won't run even though it has the highest priority because it made a blocking wait call and hence isn't runnable.

All these testcases prove that:

1. The scheduler follows a strict scheduling policy
2. The scheduler schedules only those processes which are runnable
3. It works perfectly in cases where the process block
4. It can lead to a starvation but that is implicit to the Strict Priority scheduler design

7. Time Complexity of the Scheduler

The scheduler works with a time complexity of $\theta(N)$ where N is the number of process in the list of processes

8. Instructions for running:

i. Extract 14D070063_1401100090.tar.gz into the xv6 folder.
ii. Copy the contents of the untarred folder generated (namely PatchFiles/ directory and patchAlFiles.sh) into the xv6 source code folder
iii. Run “bash patchAllFiles.sh” which will modify the source code to our code
iv. Run “make”
v. Run “make qemu”
vi. To execute TestCase1, execute “TestCase1” and similarly for the other test cases