Lab Assignment: Processes and Scheduling in xv6

29 August, 2019

Goal

The goal of this lab is to understand process management and scheduling in xv6.

Before you begin

- For this lab, you will need to understand and modify following files: *proc.c, proc.h, syscall.c, syscall.h, sysproc.c, user.h, and usys.S*. Below are some details on these files.
  - *user.h* contains the system call definitions in xv6. You will need to add code here for your new system calls.
  - *usys.S* contains a list of system calls exported by the kernel.
  - *syscall.h* contains a mapping from system call name to system call number. You must add to these mappings for your new system calls.
  - *syscall.c* contains helper functions to parse system call arguments, and pointers to the actual system call implementations.
  - *sysproc.c* contains the implementations of process related system calls. You will add your system call code here.
  - *proc.h* contains the struct proc structure. You may need to make changes to this structure to track any extra information about a process.

- *proc.c* contains the function scheduler which performs scheduling and context switching between processes.

- Understand how scheduling and context switching works in xv6. xv6 uses a simple round-robin scheduling policy, as you can see in the scheduler function in proc.c.

Assignments

1. You will implement system calls to get information about currently active processes, much like the ps and top commands in Linux do. Implement the system call getNumProc(), to return the total number of active processes in the system (either in embryo, running, runnable, sleeping, or zombie states). Also implement the system call getMaxPid() that returns the maximum PID amongst the PIDs of all currently active (i.e., occupying a slot in the process table) processes in the system.

2. Implement the system call getProcInfo(pid, processInfo). This system call takes as arguments an integer PID and a pointer to a structure processInfo. This structure is used for passing information between user and kernel mode. This structure is already implemented in the xv6 patch provided to you, within the file processInfo.h. You may want to include this structure in user.h, so that it is available to userspace programs. You may also want to include this header file in proc.c to fill in the fields suitably.
   You must write code to fill in the fields of this structure and print it out. The information about the process that must be returned includes the PID, the number of times the process was context switched in by the scheduler, and the process size in bytes. Note that while some of this information is already available as part of the struct proc of a process, you will have to add new fields to keep track of some other extra information. If a process with the specified PID is not present, this system call must return -1 as the error code.

3. In the next part, you will change the xv6 scheduler to take process priorities into account. To that end, add new system calls to xv6 to set/get process priorities. When a process calls setprio(n),the priority

of the process should be set to the specified value. The priority can be any positive integer value, with higher values denoting more priority. Also, add a system call getprio() to read back the priority set, in order to verify that it has worked. For now, you do not have to do anything with these priorities, except storing and retrieving them in the process structure.

For all system calls that do not have an explicit return value mentioned above (e.g., getProcInfo), you must return 0 on success and a negative value on failure.

Note: It is important to keep in mind that the process table structure ptable is protected by a lock. You must acquire the lock before accessing this structure for reading or writing, and must release the lock after you are done. Please ensure good locking discipline to avoid subtle bugs in your code.