ELL405 Assignment 1 Report

Nirjhar Das
2019EE30585

February 2022

1 Introduction to xv6 OS

1.1 Installing and Testing

Steps were followed as given, with a minor change that the "qemu-system-x86_64" was used to install qemu.

1.2 Implementing System Calls

To implement a system call, the following steps are to be taken:

1. Implement the function definition in the file *proc.c*
2. Implement its wrapper function in *sysproc.c*
3. Attach the definition of this function in the files *syscall.c* and *syscall.h*
4. To add user level access, we must also make necessary changes in the files *user.h* and *usys.S*
5. To link the function across multiple codes, we also need to include its signature in *defs.h*
6. Finally, we make changes in the Makefile so as to make the code compile when making

1.2.1 List the Running Process

To list the running process, we acquire the lock on the "ptable" data structure and iterate over the table to print all processes which are either "RUNNING" or "RUNNABLE" or "SLEEPING" because these are the processes that will be executed by the CPU. The information regarding process state is stored in "p-¿state" of a process p.

1.2.2 Printing available memory

The variable "PHYSTOP" denotes the limit to the physical memory. Thus, running over the "ptable", we sum up the "size" of each running process, which is the memory size currently being used by the process. We can get the difference between *PHYSTOP* and the sum to get the total available memory.

1.2.3 Context Switches

To count the number of context switches, we note that context switch happens only when:

- A process's time quanta is over, so it has to yield the CPU
- A process goes from RUNNING to SLEEPING
- A process voluntarily gives up the CPU

Thus, whenever the state of a process is changed from RUNNING to something else, we increment a variable "swtch_cnt" which is an attribute of the data structure "proc". Finally, when the process exits, we increment the counter one last time.

1.3 Scheduling Policy

xv6 employs a Round Robin scheduling policy where every process is run for one clock tick. Then it yields the CPU and the scheduler schedules the next RUNNABLE process. When a process is created, an UNUSED slot is found in the *ptable* and the process is assigned that slot. The process first stays as an EMBRYO and then becomes RUNNABLE. When a process returns from I/O operation, it changes its *state* to RUNNABLE and waits for its turn to be scheduled.

To allot a quanta to the process instead of single ticks, we maintain a counter in the "proc" data structure "tick_cnt" which counts the number of timer interrupts that occurred in the current quantum. Every time a timer interrupt happens (implemented in "trap.c"), we decrement the counter. When the counter is equal to 0, we must yield the CPU as the quantum is over.

1.3.1 First Come First Serve

To implement this, we maintain an attribute of the "proc" data structure called "cr_time" which is the creation time of the process. The scheduler iterates over the *ptable* (after locking it) and picks the process with the minimum *cr_time*. This process is then run till it either finishes, or sleeps or blocks. Hence, we do not decrement the *tick_cnt* when the timer interrupts arrive so that the process is never pre-empted till it voluntarily gives up the CPU.

1.3.2 Multi-Level Queue

To implement this, we maintain an attribute "priority" in the "proc" data structure. The priority is set by the system call as mentioned. First, the scheduler runs over the entire the *ptable* to check if there is a RUNNABLE process with priority 1. If yes, then that is scheduled for a quantum. Then the scheduler moves to the next RUNNABLE process with priority 1. This happens until there are no more priority 1 process. Then the scheduler moves to check if there are any priority 2 processes which are RUNNABLE. The scheduler keeps running such processes in Round Robin till there are no more. Note that, before scheduling any priority 2 process, we always check if there has been any new priority 1 process meanwhile. Similarly, we do for priority 3 processes.

This is implemented using the following pseudocode:

1.3.3 Dynamic MLQ

To implement this, we use reuse the code, with only addition that we change priority in the function *yield*, *sleep* and *exec* as per given scheme.

1.4 Testing the code

The function *MLQ_user_check.c* has been implemented. The result is not very precise as the processes finish execution by less than 1 quantum sometimes. Moreover, xv6 does not have option of printing floats so the averages are not very accurate. Moreover, on whole, priority 1 processes have lesser turnaround time than priority 3 processes.

Algorithm 1: Multi-Level Queue Scheduling

```
1  flag = false;
2  do
3      for process in ptable do
4          if process is RUNNABLE and process.priority = 1 then
5              flag = true;
6              Schedule process;
7          end
8      end
9  while (flag);
10 do
11     for process in ptable do
12         // Check if any higher priority process already available
            before scheduling this;
13         for process1 in ptable do
14             if process1 is RUNNABLE and process1.priority ¡ 2 then
15                 goto line 2
16             end
17         end
18         if process is RUNNABLE and process.priority = 2 then
19             flag = true;
20             Schedule process;
21         end
22     end
23 while (flag);
24 do
25     for process in ptable do
26         // Check if any higher priority process already available
            before scheduling this;
27         for process1 in ptable do
28             if process1 is RUNNABLE and process1.priority ¡ 3 then
29                 goto line 2
30             end
31         end
32         if process is RUNNABLE and process.priority = 3 then
33             flag = true;
34             Schedule process;
35         end
36     end
37 while (flag);
```

2 Linux Kernel Module

2.1 Kernel Module Writing

The kernel module is attached with the *insmod* command and removed with *rmmod* command. The *module_init* and *module_exit* are the entry and exit points. The C program compiles to give a <name>.ko file which is to be loaded as kernel module. The *printk()* function's log level has to be increased to get the output in the console. The implementation has the log level to the highest value.

2.2 Lisitng Processes

We use the data structure *task_struct* and iterate over all processes with the loop *for_each_process*. This is done at kernel entry point. Upon exit, simple exit message is printed.