ELL 783/ ELL 405: Operating Systems

Assignment-1

Total: 100 marks

Released: 21 January 2022

1. The assignment has to be done in a group of 2. Only one person needs to submit it.
2. Deadline: 21 February 2022
3. -15% penalty per day, after the deadline
4. We will use Ubuntu Linux 18.04+ to evaluate the assignment. Please ensure that it works correctly on an Ubuntu machine. If the assignment does not work on it, it is the student's fault. Please be careful with MacOS. In the past, we have faced issues with this.
5. Marks Distribution

COMPONENT	**MARKS**
Installing and testing xv6	5
Listing running processes	5
Printing the available memory	5
Context switching	10
First come first serve scheduling policy	10
Multi-level queue scheduling policy	10
Dynamic multi-level queue scheduling policy	15
Kernel module writing	10
List running processes in kernel	20
Report	10
TOTAL	100

1. Introduction to the xv6 operating system

1.1. Installing and Testing xv6[1]

- Download the assignment extract it.
- The folder "*assignment1*" contains the xv6-rev11.tar.gz file. xv6 can be installed by extracting this file.

- Follow these commands to install Qemu and xv6:

```
sudo apt-get install qemu
sudo apt-get install libc6-dev:i386
tar xzvf xv6-rev11.tar.gz
cd xv6-public
make
make qemu
```

- For more information on xv6, refer to the following link : https://pdos.csail.mit.edu/6.828/2018/xv6.html

1.2. System calls

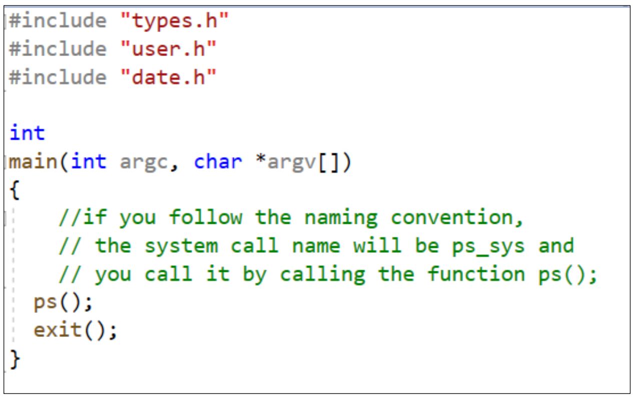
In this part, you will be implementing system calls in xv6. These are simple system calls and form the basis for the subsequent parts of the assignment.

1.2.1. Listing the running processes

In this part, you have to add a new system call, *ps_sys*, to print a list of all the currently running processes in the following format:

```
pid:<Process-Id> name:<Process Name>
eg:
pid:1 name:init
pid:2 name:sh
```

Create a user-level program to test it. A sample code to print the list of processes is shown in Figure 1.



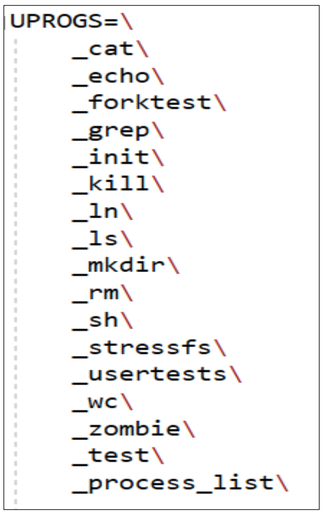
```c
#include "types.h"
#include "user.h"
#include "date.h"

int
main(int argc, char *argv[])
{
    //if you follow the naming convention,
    // the system call name will be ps_sys and
    // you call it by calling the function ps();
  ps();
  exit();
}
```

Figure 1: Sample code calling ps_sys

Instructions to add User Programs to xv6:

- Create a file called "process_list.c" in the root directory.

- Add a line "_process_list\" in Makefile (already present in the root directory). Figure 2 shows the part where the changes are to be done.

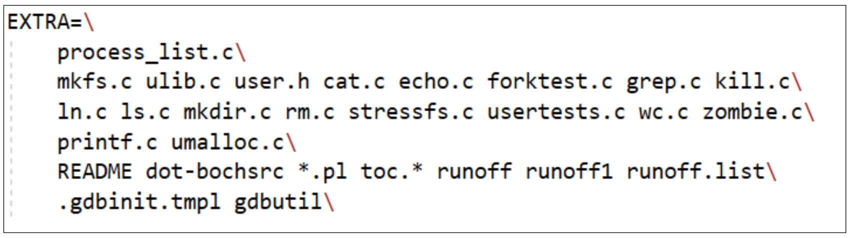


```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_test\
	_process_list\
```

Figure *2*: Entry of process_list in Makefile

The only change in this part is the last line.

- Also, make changes to the Makefile as follows (the only change is in the second line):



```
EXTRA=\
	process_list.c\
	mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\
	ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
	printf.c umalloc.c\
	README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
	.gdbinit.tmpl gdbutil\
```

- Now, enter the following commands to build the OS in the xv6 root directory.
  ```
  make clean
  make
  ```
- If you want to test the user program, launch xv6 using the command
  ```
  make qemu-nox
  ```
- After xv6 has booted in the new terminal, you can type
  ```
  ls
  ```

This will show the process list in the list of available programs. If it does not, there is some error either in the code or in setting up the Makefile.

1.2.2. Printing the available memory

In this part, you have to add a new system call, *memtop.* It takes no arguments and returns the amount of memory available in the system. When you invoke it from your test program, you should print the number in bytes on the console. Create a user-level program to test it.

Format:

int memtop()

Output:

available memory: 29712384

1.2.3. Context switching

You have to add a new system call, *csinfo*. It returns the number of context switches in the process from the time it started.

Format:

int csinfo()

Implementing this will require keeping an additional counter in the proc structure for the file. Create a user-level program to test it.

To test it out, try code along the lines of:

```
int cs1, cs2, cs3, cs4;
cs1 = csinfo();
cs2 = csinfo();
sleep(1);
cs3 = csinfo();
sleep(1);
cs4 = csinfo();
printf(1, context switch counts = %d, %d, %d, %d\n", cs1, cs2, cs3, cs4");
```

1.3. Scheduling Policy:

The set of rules used to determine when and how to select a new process to run is called a scheduling policy. You first need to understand the current scheduling policy. Locate it in the code and try to answer the following questions: which process does the policy select for running, what happens when a process returns from an I/O operation, what happens when a new process is created and when/how often does the scheduling take place.

First, change the current scheduling code so that process preemption will be done in every time quantum (measured in clock ticks) instead of every clock tick. Add this line to ***param.h*** and initialize the value of QUANTA to 5.

#define QUANTA <Number>

In this part, you will add three different scheduling policies to the existing scheduling policy of xv6. Add these policies by using the C preprocessor.

Modify the ***Makefile*** to support *SCHEDFLAG* – a macro for quickly compiling the appropriate scheduling scheme. Thus the following line will invoke the xv6 build with the default scheduling policy:

- *make qemu SCHEDFLAG=DEFAULT*

The default value for SCHEDFLAG should be DEFAULT (in the Makefile).

1.3.1. First Come First Serve: (*SCHEDFLAG= FCFS*)

Represents a non-preemptive policy and selects the process with the lowest creation time.

1.3.2. Multi-level Queue Scheduling: (*SCHEDFLAG= MLQ*)

Represents a preemptive policy that includes three priority queues. Priority 2 is assigned to the initial process, and the priority is copied upon a fork. In this scheduling policy, the scheduler will only select a lower priority queue if no process is ready to run at a higher priority queue. The algorithm first runs all the processes with the highest priority, and then, when they finish, it will consider all the processes with a lower priority. Moving between priority queues is only available via a system call. The process selection is set by the current system default (Round-Robin) within a given queue. The priority range in this algorithm is 1-3 (default is 2), where we give priority equals to 1 for the processes which we want to be completed first.

The following system call will change the priority queue of the process with a specific pid process:

int chpr (int pid, int priority)

priority: A number between 1 and 3 represents a new process priority.

1.3.3. Dynamic Multi-level Queue Scheduling: *(SCHEDFLAG= DMLQ*)

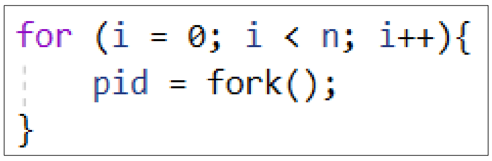
Dynamic Multi-level Queue Scheduling represents a preemptive policy similar to multi-level queue scheduling. The difference is that the process cannot manually change its priority. There are the dynamic priority rules:

- Calling the *exec* system call resets the process priority to 2 (default priority).
- Returning from the SLEEPING mode (in our case I/O) increases the process's priority to the highest priority.
- Yielding the CPU manually keeps the priority the same.
- Running the whole quanta will result in a decrease of priority by 1.

1.3.4. Testing the code

Create two user programs to test the impact of each scheduling policy.

- The first user program called schedule_user tests all three scheduling policies and prints the output statistics. This program gets a number n as the argument and forks m processes and will wait till all of them finish and print the statistics for each child (n should be greater than 5). Hint: you may fork the processes along the lines of:



```
for (i = 0; i < n; i++){
    pid = fork();
}
```

All the m processes are one of the three types:

- CPU bound process (CPU): process with (pid mod 3 = 0):
    - For these processes, run a dummy loop 100000 times
- Short tasks based on CPU bound process (S-CPU): process with (pid mod 3 = 1)
    - For these processes, run a dummy loop 100 times, with each dummy loop running for 100000 iterations.
    - After every dummy loop (of the 100), yield the CPU.
- I/O bound process (IO): process with (pid mod 3 = 2)
    - For these processes, run dummy sleep calls for 1000 times: sleep(1)

<u>Printing the Statistics:</u>

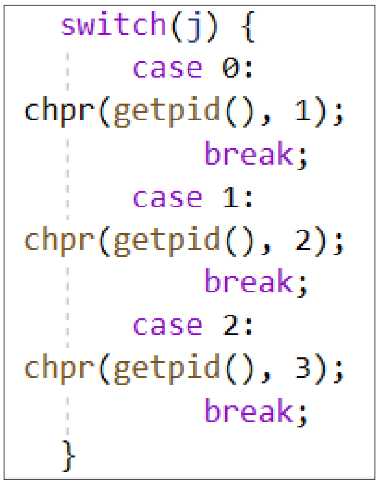
You have to print the statistics for each terminated process and the average values of all the m processes.

- Print the pid, process type (CPU/ S-CPU/ IO), ready time, run time, sleep time, and turnaround time for each process.
- At the termination of all the m processes, print the following average statistics of each of the process types (CPU/ S-CPU/ IO):
- Average Ready time: Average time the processes were in the ready state.
- Average Run time: Average time the processes were in the ready state.
- Average Sleep time: Average time the processes were in the sleep state
- Average Turnaround time: Average time for the processes to complete (ready time + run time + sleep time).

Analysis:

• Run the test cases for different scheduling policies and compare the results. Provide the justifications, explaining the correctness of your implementation and which policy is better.

• The second user program called MLQ_user_check tests the scheduling order of the processes in the multi-level queue scheduling policy. Fork n number of CPU bound processes (say 30), give each process a different priority, and then print the statistics for each terminated process and the average values of all the processes. The priorities can be assigned along the lines of:



```
   switch(j) {
       case 0:
chpr(getpid(), 1);
           break;
       case 1:
chpr(getpid(), 2);
           break;
       case 2:
chpr(getpid(), 3);
           break;
   }
```

2. Introduction to Linux Kernel Modules

In this part, you will learn how to create a kernel module and load it into the Linux kernel.

2.1. Kernel module writing

- You have to write a basic kernel module that prints the "Kernel Module Loaded" message when loaded and "Kernel Module Removed" when unloaded.
- By default, *printk()* writes the message to the */var/log/messages* file; you have to write the module in such a way that the messages are printed to the current terminal window.
- Inserting the module in one window and removing it in another should print the messages in the respective window.

2.2. Listing the running tasks

- You need to write a kernel module that lists all the currently running tasks in a Linux system. In particular, output the task command, state, and process id of each task.
- Write this code in the module entry point so that its contents will appear in the kernel log buffer, which can be viewed using the *dmesg* command. To verify that your code is working correctly, compare the contents of the kernel log buffer with the output of the following command, which lists all the tasks in the system: ***ps -el*** .

3. Report

The report should clearly mention the implementation methodology for all the assignment components. The report should contain a detailed description of your new scheduler. Small code snippets are alright; additionally, the pseudo-code should also suffice. The snapshots of the outputs should also be there.

• Details that are relevant to the implementation.

• Say what you have done that is extra (this should be the last section in the document).

• Limit of 10 pages (A4 size) and must be in the PDF format (name: report.pdf).

Submission Instructions

• We will run MOSS on the submissions. We will also include last year's submissions and resources on the web. Any cheating will result in a **zero** in the assignment, a penalty as per the course policy, and possibly much stricter penalties (including a fail grade and/or a DISCO).

• There will be NO demo for assignment 1. Your code will be evaluated based on the test cases you provided and the hidden test cases checking the corner cases, and marks will be awarded based on that. We will read through your report to understand your design and the testing. We will also read your code to ensure that you have adhered to the problem specification in your implementation. Please make sure that the output of your test program is clean enough to understand the results of the tests.

How to submit:

1. Copy all your code for the Linux kernel module part to a folder named *kernelModule.*

2. Copy the folder *kernelModule* and your report to the xv6 root directory. Donot forget to include your test program.
3. Then, in the root directory run

```
make clean
tar czvf assignment1_<entryNumber1_entryNumber2>.tar.gz *
```

   This will create an assignment1_<entryNumber1_entryNumber2>.tar.gz tar ball in the same directory. Submit this tar ball on Moodle. Entry number format: 2017ANZ8353
4. Please note that if the report is missing in the root directory, then no marks will be awarded for the report.

References

1. xv6. https://pdos.csail.mit.edu/6.828/2018/xv6.html. (Accessed on 01/18/2022).