ELL 783/ELL 405: Operating Systems
Assignment-1

Total: 50 Marks
Released: 19th Feb 2021

1. The assignment has to be done in a group of two. Only one person need to submit it.
2. **Deadline: TBA**
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 issue with this.

1 Installing and Testing xv6 [1]: 5 Marks

- xv6 is available at :
  `https://pdos.csail.mit.edu/6.828/2018/xv6.html`
- **Install xv6 version 11**. `http://www.cse.iitd.ernet.in/~kumarsandeep/ta/ell405/assig1/xv6-rev11.tar.gz`
- Follow these commands to install Qemu and xv6:

```
sudo apt-get install qemu
sudo apt-get install libc6-dev:i386
wget http://www.cse.iitd.ernet.in/~kumarsandeep/ta/ell405/assig1/
xv6-rev11.tar.gz --no-check-certificate

tar xzvf xv6-rev11.tar.gz
cd xv6-public
make
make qemu
```

- Information on Qemu can be found here: `http://www.cse.iitd.ernet.in/~kumarsandeep/ta/ell405/assig1/qemu_makefile_tutorial.pdf`

- The check scripts can be downloaded from here: `http://www.cse.iitd.ernet.in/~kumarsandeep/ta/ell405/assig1/check_scripts.tar.gz`
- The README file inside the check scripts tar ball contains the instructions to run the check scripts.

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.

2.1 Tracing the system call: 10 Marks

Here, you will be tracing the system calls executed. For this, let us define two states within the kernel: *trace_on* and *trace_off*. If the state is equal to *trace_off*, which is the default, then nothing needs to be done. However, it is possible to set the state to *trace_on* with a special system call, which we shall see in the next bullet point. The state can be subsequently reset (set to *trace_off*) as well.

Whenever the state changes from *trace_off* to *trace_on*, you need to enable a custom form of system call tracing within the kernel. This will keep a count of the number of times a system call has been invoked **since the state changed** to *trace_on*.

Let us define a new system call called *sys_print_count*. It will print the list of system calls that have been invoked since the last transition to the *trace_on* state with their counts. A representative example is shown below. The format is as follows. There are two columns separated by a single space. The first column contains the name of the system call, and the second column contains the count. There are no additional lines; the system call names are **sorted alphabetically in ascending order**.

```
sys_fork 10
sys_read 20
sys_write 0
```

2.2 Toggling the tracing mode: 5 Marks

Let us now add one more system call, *sys_toggle()* that toggles the state: if the state is *trace_on* set it to *trace_off*, and vice versa. A sample program to toggle the state is shown in Listing 1.

Listing 1: "Sample code calling *sys_toggle*"

```
1 #include "types.h"
2 #include "user.h"
3 #include "date.h"
4 int main(int argc , char *argv [])
5 {
```

```
    // If you follow the  naming convention , the system call
    // name will be sys_toggle and you
    // call it by calling the function toggle ();
    toggle ();
    exit();
}
```

Instructions to add User Programs to xv6:

- Create two files called, "user_toggle.c", and "print_count.c" in the root directory.
- Add a line "_user_toggle" and "_print_count" in *Makefile* (already present in the root directory). Listing 2 shows the part where the changes are to be done.sudo apt-get install qemu sudo apt-get install libc6-dev:i386

Listing 2: "Makefile with relevant changes"

```
UPROGS=\
_cat\
_echo\
_forktest\
_grep\
init\
_kill\
_ln\
_ls\
_mkdir\
_rm\
_sh\
_stressfs\
_usertests\
_wc\
_zombie\
_user_toggle\
_print_count\
```

The only change in this part is the last two lines (17 and 18).

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

```
EXTRA=\
user_toggle.c print_count.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 *user_toggle* and *print_count* in the list of available programs. If it does not, then there is some error either in the code or in setting up the *Makefile*.

2.3 Add system call: *sys_add*: 5 Marks

In this part, you have to add a new system call to xv6. The first system call, *sys_add*, should take two integer arguments and return their sum. Create a user-level program to test it.

2.4 Process List: *sys_ps*: 5 Marks

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

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

Similarly for this part, create a new user program, which should in turn call your *sys_ps* system call.

3 Inter-Process Communication: 5 Marks

In this part, you have to create system calls for communication between processes. You are not allowed to use *sys_pipe* for this purpose (which is already present in the xv6 code).

3.1 IPC system calls for unicast communication

You need to implement two system calls here. First one is *sys_send*. It should take three arguments.

```
int sys_send(int sender_pid, int rec_pid, void *msg)
```

- *sender_pid*: pid of the sender process.
- *rec_pid*: pid of the receiver process.
- *msg*: pointer to the buffer that contains a 8-byte message.
- This is a **non-blocking** call.

The return type is *int*: 0 means success and any other value indicates an error.

Similarly we have another function for the receiving the message.

```
int sys_recv(void *msg)
```

- msg: Pointer to the buffer where the message from the sender is stored(after *sys_recv* returns).
- This is a **blocking** call.
- The process calling this should only receive messages that were sent to it (using its pid). The kernel should ensure that messages meant for other processes are not received by it.

As you may have noticed, a sender process must have the pid of the receiver process before it can send a message to it. This part is left open, and you are free to implement this as you may want it. However, you are not allowed to "hard-code" the pid values.

4 Distributed Algorithm: 5 Marks

In this part, you will use multiple processes to compute the sum of the elements in an array in a distributed manner. Modern systems have multiple cores, and applications can benefit from parallelizing their operations so as to use the full capability of the hardware present in the system. Use your unicast primitives. The task here is simple. Calculate the sum of an array, subject to the following conditions:

- Total number of elements in the array: 1000.
- Elements are from 0 to 9.
- A sample input dataset can be downloaded from here: `http://www.cse.iitd.ernet.in/~kumarsandeep/ta/ell405/assig1/arr`.
  To add this file to the xv6 file system, modify these lines in the Makefile:

```
1 fs.img: mkfs README arr $(UPROGS)
2    ./mkfs fs.img README arr $(UPROGS)
```

A sample program has been provided. It needs to be completed. The program takes two command line arguments: < *type* > and <input_file_name>. **Set** < *type* > **to 0**. It is required for grading purposes and has no impact on the functionality.

Some more points:

1. Create (assign) a coordinator process. This will collect the partial sums from the rest of the processes, compute and print the sum. The format of the output is specified in the sample program.
2. Limit the number of processes to 8. Note that your program should run for 8 processes.

5 Report: 10 Marks

The report should clearly mention the implementation methodology for all the parts of the assignment. Small code snippets are alright, additionally, the pseudo code should also suffice.

- Details that are relevant to the implementation.
- Submit an OpenOffice presentation.
- 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 PDF format (name: report.pdf).

6 Submission Instructions

- We will run MOSS on the submissions. We will also include last year's submissions. 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 using a check script (check.sh) on hidden test cases and marks will be awarded based on that.

How to submit:

1. Copy your report inside the xv6 root directory.
2. Then, in the root directory run

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

This will create a tar ball with name, assignment1_<entryNumber1_entryNumber2>.tar.gz in the same directory. Submit this tar ball on Moodle. **Entry number format: 2017ANZ8353**

3. Please note that if the report is missing in the root directory then no marks for the report.

References

1. xv6. `https://pdos.csail.mit.edu/6.828/2018/xv6.html`. (Accessed on 01/17/2021).