COL331: Operating System
Assignment 1

Updated

Total: 100 Marks

The updates are written in blue color : 5 Feb 2019.

Key Points:

1. The assignment has to be done individually.
2. Penalty for late submission: 3-day buffer period. Thereafter, -15% penalty per day.
3. Please use an Ubuntu machine for doing the assignments. Mac and Windows operating system are **NOT** supported.
4. Instructions to install Ubuntu can be found here: `http://www.cse.iitd.ernet.in/~kumarsandeep/ta/col331/assig1/ubuntuinstructions.pdf`. These are the instructions for installing Ubuntu 16.04. They are similar for other Ubuntu versions from 14 to 18.

There are 5 parts of the assignment. Please note that we will use xv6 for this assignment.

1 Installing and Evaluating xv6 [1]: 10 Marks

- xv6 is available at : `https://pdos.csail.mit.edu/6.828/2018/xv6.html`.
- Instructions to build and install xv6 can be found here: `http://www.cse.iitd.ernet.in/~kumarsandeep/ta/col331/assig1/XV6instructions.pdf`.
- Build instructions for Qemu can be found here: `http://www.cse.iitd.ernet.in/~kumarsandeep/ta/col331/assig1/qemu_makefile_tutorial.pdf`.

2 System calls: 20 Marks

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

- System call trace:
  Your first task is to modify the xv6 kernel to print out a line for each system call invocation. Please note no new system call has to be added for this part. Print the name of the system call and the number of times it has been called in the following format:

  ```
  <System Call Name>  <count>
  eg:
  sys_fork 10
  ```

  The system calls issued on booting the xv6 and the number of times they are called, is same all the time. Hence, your implementation should print the same sequence every time starting from the boot time. We will use some other hidden test cases, whose output will be different to make sure that the implementation is correct and not just a bunch of *printf* statements.

- System call toggle (*sys_toggle()*):
  After you are done with the part *a* of the assignment, every time you boot or issue some command, you will see system trace printed on the screen, which makes it difficult to see the output of some other command. Hence, as part *b*, you will have to implement a sys_toggle() system call, which will toggle the system trace. If the system trace is ON, the system call should switch it OFF, and vice versa. By default, when xv6 boots, it should be ON.

  After the system call implementation is done, you need a user program to actually make the system call. It can be named anything. Assuming you want to add user_toggle user program to xv6, you need to create a file name user_toggle.c, whose contents will be like:

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

  int
  main(int argc, char *argv[])
  {
  // If you follow the naming convetion, system call
  // name will be sys_toggle and you
  // call it by calling
  // toggle();

  exit();
  }
  ```

  Adding a user program to xv6:

  - Create user_toggle.c in the xv6 directory.
  - Add a line "_user_togglëo the *Makefile*. Your Makefile should look like this

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

- Also make changes to the Makefile as following:

```
EXTRA=\
user_toggle.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 following commands:

```
make clean
make
```

- If you want to test the user program, launch xv6 using command

  `make qemu-nox`

  After xv6 has booted in the new terminal, you can type

  `ls #(this will be with system traces, implemented in part \(a\))`

  This will show the user_toggle in the list of available programs, call it using user_toggle To make sure it worked, try ls command again, which should be free of the system traces.

- Add system call: *sys_add()*
  In this part you have to add a new system call to xv6. This system call should take two integer arguments and return their sum.

  You should add a system call named, sys_add() to xv6 and then create a user-level program to test it

- Process List: *sys_ps()*
  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
```

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

3 IPC Communication: 45 Marks

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

3.1 IPC Syscalls: Unicast [25 Marks]

You need to implement a system call for this purpose, which should be named *sys_send*. The length of the message is fixed to **8 bytes**. The system call will 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 the message.

The return type is *int*, 0 means success and any other value means failure. The responsibility to check the messages should be with the receiver and not the sender. To achieve this you need to implement another system call, *sys_rec*. The system call signature will be similar to the send call.

~~int sys_rec(int sender_pid, int *rec_id, void *msg)~~

```
int sys_recv(int* myid , int *from, void *msg)
```

Changes in the name of the variables does not matter.

- myid: after the call will contain my pid.
- from: will contain the senders pid.
- msg: will contain the message from the sender.

In this system call *from* is used to store the pid of the process from where the message is sent. A process sends its *pid* in *myid* and a pointer to get the pid of the process which sent the message (to be filled in *from*) and a pointer to store the *msg*.

After a *fork()* call, calling *getpid()* in the child does not give the correct pid. However, this information is available in the *proc* structure of the child, which can be retrieved during the system call.

Here, *myid*, ~~*rec_id*~~ *from* and *msg* buffer will be initially empty, which will be filled after this system call is successful. Similar to previous call, return type is *int*, 0 means success and any other value means failure. This should follow a **polling model**, meaning the message will only be delivered when the receiver make a *sys_rec* call to the OS.

3.2 Multicast [20 Marks]

In the previous section you implemented a unicast model of communication. In this section, you need to implement a multicast model of communication. The basic unicast model, the message is from one sender to one receiver. In the multicast model, one sender sends a single message to multiple receivers in a single system_call. The send system call will change :

```
int sys_send(int sender_pid, int rec_pids[], void *msg)
```

Here, *rec_pids* is an int array which will contains the pids of the process to whom the message is to be sent. This should follow an **interrupt model**. The receiving processes will be notified about the message arrival by the OS using an interrupt.

User programs will need to implement an interrupt handler

Please explain the design choices you make to implement this in the report.

4 Distributed Algorithm: 15 Marks

In this part you will implement *array sum* in a distributed manner. Modern systems are multicore, and applications can benefit from parallelizing their operations so as to use the full capability of the hardware present in the system. The way to do this is left to you. You can use *shared memory*, *shared files*, or *sockets* to do this.

The task here is simple. Calculate the sum of an array, bounded by the following conditions:

- Total elements in the array $1 \leq x \leq 1,000,000$
- Value of each element is (0 to 9)
- There can be duplicates in the entry.
  A sample input dataset can be downloaded from here:
  `http://www.cse.iitd.ernet.in/~kumarsandeep/ta/col331/assig1/million_zero_nine.txt`.

IPC communications between processes incur overheads, also there are costs involved with process creation and context switching. This will impact the runtime of the algorithm. In the report, you need to plot how the performance changes with change in the number of processes. Typically number of processes is varied from 1 to 8.

Also use both the model of communication, to implement the algorithm and report the difference in the performance in your report.

5 Report: 10 Marks

Page limit: 10
The report should clearly mention the implementation methodology for all the parts of the assignment. Small code snippets are OK, additionally pseudocode should also suffice.

- Distributed Algorithm: How exactly does your algorithm work?
- IPC: Explain the implementation of the interrupt handler
- Any other details relevant to the implementation.

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.

We will release some testing scripts in few days, which can be used to check the implementation.

References

[1] xv6. `https://pdos.csail.mit.edu/6.828/2018/xv6.html`. (Accessed on 01/17/2019).