Operating Systems
Assignment 1 – *Easy*

Total: 40 Marks

Instructions:

1. The assignment has to be done individually.
2. You can use Piazza for any queries related to the assignment and avoid asking queries on the last day.

The assignment has 5 parts.

1 Installing and Testing xv6: 3 Marks

- xv6 is available at the following link .
- **Install xv6 version 11**.
- Instructions to build and install xv6 can be found here .
- Build instructions for Qemu can be found here.

2 System calls: 7 Marks

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.

- System call trace:
  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
```

- Toggling the tracing mode:
  Let us now add one more system call, *sys_toggle()* that toggles the state: if the state is *TRACE_ON* sets it to *TRACE_OFF*, and vice versa. A program to toggle the state looks like this:

```
#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 sys_toggle and you
    // call it by calling the function toggle();
    // toggle();

    toggle();
    exit();
}
```

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 the *Makefile*. The relevant part of your Makefile should look like this:

```
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.

- Also make changes to the Makefile as follows (the only change is on 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 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.

- 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. Create a system call, *sys_add()*, 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 Inter-Process Communication: 10 Marks

We implement the following communication models between the processes, namely:

1. Unicast : In this communication model, a process can communicate with another process using two system calls namely, *sys_send* and *sys_recv*. The *sys_send* system call is responsible for sending the data from a sender process to a receiver process. Similarly, the *sys_recv* system call is responsible

for reading the data sent from a sender process at the receiver process. The length of the message is fixed at **8 bytes**. The signatures of the system calls are:

```
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 8-byte 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 return type is *int*: 0 means success and any other value indicates an error.

2. Multicast : In contrast to the unicast communication model, the multicast model sends a data from a sender to multiple processes using the *sys_send_multi* system call. You need to implement this system call using interrupts. This will involve writing a signal handler that is called by the kernel. You need to make appropriate modifications to the kernel as you deem fit. Name the software interrupt *sys_msg_mcast*. The signature of the system call is:

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

- *sender_pid:* pid of the sender process.
- *rec_pids:* pids of the receiver processes.
- *msg:* pointer to the buffer that contains the 8-byte message.

4 Distributed Algorithm: 10 Marks

In this part, you will 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 and multicast 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 program has been provided. It needs to be completed. The program takes two command line arguments: $\langle type \rangle$ and $\langle input\_file\_name \rangle$. If the type is 0, you use unicast communication, and if it is 1, you use multicast based communication.

Some more points:

1. In the unicast version 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. In the multicast version, you need to extend the program to compute the variance. Divide your program into two phases. In the first phase, you have the standard unicast version, where all the processes send their partial sums to the coordinator. The coordinator then computes the mean and then **multicasts** it to all the worker processes. The worker processes then unblock themselves, compute the sum of the squares of the differences about the mean (second phase). Then the worker processes send the partial sums to the coordinator. The coordinator computes and prints the variance.

3. Limit the number of processes to 8. Note that your program should run for 8 processes.

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 alright, additionally, the pseudocode should also suffice.

- Distributed Algorithm: How exactly does your algorithm work?
- IPC: Explain the implementation of the interrupt handler
- Any other details that are relevant to the implementation.
- Submit a pdf containing all the relevant details.
- Say what you have done that is extra.