ELL405 Assignment 1

1. Installing Xv6

The following commands were used to download and install xv6 and run it.

```
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
```

2. Adding a System Call

To Add a system call we need to change following files -

1. syscall.h - to assign a unique number to our added system call
2. syscall.c - to add a function pointer to our implemented system call in the array `syscalls` for system to call and give a declaration to our system call function.
3. sysproc.c - actual implementation is done here for our system call.
4. usys.S - we add line SYSCALL(<name>) to allow user to call this.
5. user.h - we declare the user wrapper for user to use the system call.

3. sys_toggle

In `syscall.c` we define the following global variable to maintain the current state of toggle

```
extern int toggle_state;
```

And following snippet in `sysproc.c` to define the the system call and initialize the global state `toggle_state` itself -

```
int toggle_state = 0;

int
sys_toggle(void)
{
  for(int i=0;i<NELEM(sysCallName);i++){
    numSysCalls[i] = 0;
  }
  if(toggle_state==0){
    toggle_state=1;
  }else{
    toggle_state=0;
```

```
  }
  return 0;
}
```

Here if toggle_state == 1 implies the system is currently in state toggle_on else it is in toggle_off.
numSysCalls[] is an array which is declared in syscall.c and sysproc.c similar too toggle_state and maintains the number of calls of ith system call after toggle was last turned on. Also we clear numSysCalls everytime toggle is switched to avoid junk data.

4. sys_print_count

In syscall.c we define the following global to maintain the names of all system calls and count of system calls made after last toggle_on operation -

```
extern int numSysCalls[];
extern const char* sysCallName[];
```

Here `sysCallName` contains name of system calls as per their index and `numSysCall` maintains the number of calls of ith system call after toggle was last turned on.
And are implemented along with system call as following in sysproc.c

This `numSysCalls` is maintained by `sys_call()` in `syscall.c` and updates are done as the system call is processed thus the count is accurate.

`sys_print_count` merely reads this array and prints it. Further quick sort was used to ensure that output was sorted.

5. sys_add

- Here we implemented a system call `sys_add()` that takes two integers and returns their sum.
- The system call has to be passed arguments via xv6's built-in function `argint()`, as system calls can't be passed arguments in the usual way.
- The implementation of this system call is in `sysproc.c` at line 108:

```
int
sys_add(void)
{
  int num1, num2;
  argint(0, &num1);
  argint(1, &num2);
  return num1 + num2;
}
```

6. sys_ps

We implemented `process_analyzer()` function to perform sys_ps in proc.c and called it from sysproc.c. it acquires the lock for ptable to avoid any parallelism hazards and accesses all the processes from it to be printed in output.

7. sys_send and sys_recv

- Here we implemented a simple IPC protocol that lets a process send an 8-byte message to another process.
- We used the **shared memory** method to implement this, by implementing a bounded buffer that processes can access concurrently.
- To allow concurrent access of the bounded buffer, a spinlock was used to prevent multiple processes accessing the buffer at the same time.
- The structures involved with this protocol are defined in `sys_send_recv_structs.h`. Two structs are defined there, `message` and `buffer`.
- `message` struct encapsulates a single 8-byte message, along with the sender pid and the pid of the receiver process. `buffer` struct encapsulates the bounded buffer. It contains an array of 1024 `struct message` objects, as well as a spinlock associated with the buffer.
- In `sysproc.c`, an object of type `struct buffer`, named `buf` is declared at line 12. This is the common bounded buffer that all processes will share to send messages to each other.
- In `sys_send`, after gathering the arguments passed to the system call using `argint` and `argptr` calls, it first acquires the lock of the buffer `buf`. Then it scans the buffer's array (of type `struct message`) to find an index that is empty where it can put its message (a message is empty if its `sender_pid` field is set to -1, an invalid pid). Then it writes the provided message to the message object at that index, sets its `sender_pid` to the sender pid provided to it, and `rec_pid` to the receiver pid provided to it. Then it releases the lock and returns 0, indicating success.
  If it is not able to find an empty space in the buffer, it releases the lock and returns -1.
  Thus this is a non-blocking call, as it makes a single pass across the buffer, and if it finds empty space, it adds its message to the buffer, else it just returns -1.
  This function is implemented in `sysproc.c`, at line 238.
- In `sys_recv`, after gathering the pointer argument passed to it using an `argptr` call, it immediately enters an infinite loop. This ensures this is a blocking system call, and will return only when it receives one message from the buffer. Inside the loop, it first acquires the lock of the buffer `buf`. Then it scans the buffer's array (of type `struct message`) to find an index having a message object that has `rec_pid` equal to the pid of the process that called this system call. If it finds a message, it writes that message to the pointer provided to it, and sets `sender_pid` and `rec_pid` of the message it read

from the buffer to -1, to indicate that the field is now empty. Then it releases the lock and returns 0, indicating success.
If it is not able to find an empty space in the buffer, it releases the lock, and the loop's next iteration begins.
This function is implemented in `sysproc.c`, at line 261.

8. Distributed Algorithm

- This is implemented in the file `assig1_8.c`, which is then run as a user program in xv6. This uses the IPC protocol implemented in the previous section.
- Here a parent process spawns 7 child processes using `fork()` system call to have those child processes compute the partial sum of separate portions of a given array of 1000 integers, then send these computed partial sums back to the parent(using `sys_send`). The parent, after receiving these seven partial sums(using `sys_recv`) sums them all up to get the sum of all elements of the array.
  Each child process executes a function `partial_sum()`, that computes the sum of a portion of the array that is unique to each child process. The implementation of this function is given below(from line 5 of `assig1_8.c`):

```
int partial_sum(short arr[], int start_index, int step, int size_of_arr)
{
	int ret_sum = 0;
	int i = start_index;
	while(i < size_of_arr) {
		ret_sum += arr[i];
		i += step;
	}
	return ret_sum;
}
```

  Here `step` is set to 7, the total number of child processes working on the given array `arr`. This ensures no process takes a number that any other process has already taken into its partial sum.
- The `getpid()` system call is used to get the pids of the child and parent processes, to be used in `sys_send` by the child processes.