ELL 405: Operating Systems
Assignment 1 Report

Lakshya Kumar Tangri 2018EE10222
Somanshu Singla 2018EE10314

March 26, 2021

Assignment 1

Note: *LaTeX template courtesy of Prof. Lalan Kumar IITD Electrical dept.*

1 Objectives

1. Installing and Testing xv6
2. Implementing simple system calls in xv6
3. Creating system calls for communication between processes
4. Using multiple processes to compute the sum of the elements in an array

2 System Specifications

In this assignment a virtual machine with Ubuntu 20.04 LTS on a windows machine has been used for all implementations and testing.

3 Theory in brief

System call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for user programs to interact with the operating system. A user program makes a system call when it makes a request to the operating system's kernel. System call provides the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system.

Interprocess communication is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another. There can be multiple ways of implementing IPC, the one which we look at and implement in this assignment is called *message passing*. This involves two processes a sender and a receiver, the sender sends a message to receiver's *mailbox* which the receiver can access once it looks at it's mailbox.

4 Procedure and Implementations

4.1 Installing

We just followed the instructions given on Assignment statement and MIT's page regarding installation of xv6 and qemu.

4.2 Adding System calls

We need to the following steps for making a new system call mySysCall

1. usys.S: add a new line SYSCALL(mySysCall) to the end of file
2. syscall.h :append new line at end of file #define SYS_mySysCall 23
3. user.h :add function prototype for mySysCall
4. syscall.c : add extern int sys_mySysCall(void); line
5. syscall.c : add [SYS_mySysCall] sys_mySysCall, line
6. In some file, define the sys_mySysCall function. for example either in Sysproc.c, or proc.c etc.

4.2.1 Tracing the system call i.e sys_toggle() and sys_printcount()

After doing the above steps:
We had to define two states within the kernel: trace on and trace off.

```
extern int sys_toggle(void);
extern int sys_print_count(void);
extern int sys_add(void);
extern int sys_ps(void);
extern int sys_send(void);
extern int sys_recv(void);
extern int trace_off;
```

Listing 1: extern definitions in syscall.c

So, trace_off variable is added to take care of the state within the kernel trace_off=1 means the state is trace off and vice versa.

```
extern int count_calls[27];
// an external declaration as i would need to accesss count_calls in sysproc.c as
    well
void
syscall(void)
{

  int num;
  struct proc *curproc = myproc();
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
      if(trace_off==0){ count_calls[(num-1)]++;} //increase count of call if
    trace is on
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

```
}
```

Listing 2: maintaining the count of system calls in syscall.c

Now for keeping count of system calls a count calls array is declared. Each system call whenever executed goes through the void syscall(void) in function in syscall.c so this was the obvious choice for maintaining the count of system calls, as array is 0-indexed and sys calls is 1 indexed, so we increment the count at index "num-1".

In xv6 the implementation of all system calls are done in a file by the name *sysproc.c*. Function definitions in sysproc.c:

```
int
sys_toggle(void)
{
if(trace_off==1)
  {
      trace_off=0;
  }
else
  {
    for(int i =0 ; i<27;i++)
    {
      count_calls[i]=0;     /// so that the perivous count gets erased
    }
    trace_off=1;
  }
return 0;
}

int
sys_print_count(void)     // hard coded as the number of systesm calls won't change once a kernel is booted
{
if(count_calls[23]!=0){
    cprintf("%s %d\n","sys_add",count_calls[23]);
 }

 if(count_calls[8]!=0){
    cprintf("%s %d\n","sys_chdir",count_calls[8]);
 }
if(count_calls[20]!=0){
    cprintf("%s %d\n","sys_close",count_calls[20]);
 }
if(count_calls[9]!=0){
    cprintf("%s %d\n","sys_dup",count_calls[9]);
 }
if(count_calls[6]!=0){
    cprintf("%s %d\n","sys_exec",count_calls[6]);
 }
if(count_calls[1]!=0){
    cprintf("%s %d\n","sys_exit",count_calls[1]);
 }
if(count_calls[0]!=0){
    cprintf("%s %d\n","sys_fork",count_calls[0]);
 }
if(count_calls[7]!=0){
    cprintf("%s %d\n","sys_fstat",count_calls[7]);      // this is just a snippet of the actual
 }                                                     // implementation
if(count_calls[10]!=0){                                // complete code not uploaded for this function
    cprintf("%s %d\n","sys_getpid",count_calls[10]);
```

```
 }
if(count_calls[5]!=0){
    cprintf("%s %d\n","sys_kill",count_calls[5]);
 }
if(count_calls[18]!=0){
    cprintf("%s %d\n","sys_link",count_calls[18]);
 }
if(count_calls[19]!=0){
    cprintf("%s %d\n","sys_mkdir",count_calls[19]);
 }

return 0;
}
```

Listing 3: implementation of toggle and print_count in sysproc.c

Here if state is toggled, apart from changing state of kernel the entire count array is cleared if the final state is trace-off because we don't won't to carry forward the counts. For the print count function it is trivial to see that we have just accessed and printed the count of each and every system call made until now in lexicographical order. Also we have hard coded the print count function as the number and name of system calls won't change once kernel is booted.

4.2.2 Implementing sys_add

```
int
sys_add(int a, int b)
{
  int c =0;     // argint() is a method to take in input parameters inherent to
    xv6
   if(argint(0, &a) < 0){ // a check just as the xv6 does
     return -1;}
    if(argint(1, &b) < 0){  // a check just as the xv6 does
        return -1;}
  c=a+b;
return c;
}
```

Listing 4: implementation of add in sysproc.c

This is a very simple implementation, the only thing to note here is that for taking the input parameters from user space to kernel space one needs to use *argint* method.

4.2.3 Implementing sys_ps

```
void
sys_ps(void)
{
  ps();    // helper function as one needs to access ptable which can only be
    done in proc.c
}
```

Listing 5: implementation of sys_ps using ps helper function

```
//a helper function to help ps() system call because printing information about
    all processes requires access to ptable where in I can access each processe's
    PCB and this is only possible within
// proc.c file so had to declare a extern ps function in sysproc.c (where the
    system call needs to be implemented) and defined it here. The idea was
    suggested in a stack overflow post.
void
ps(void)
{
```

```
  struct proc *p;     // all these syntax taken from exit() system call inherent
    to xv6
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
     if( p->pid!=0)
     {
          cprintf("pid:%d ",p->pid);
      cprintf("name:%s\n",p->name);
       }
        }
  release(&ptable.lock);

}
```

Listing 6: implementation of ps helper function in proc.c

Printing information about all processes requires access to ptable where in we can access each processe's PCB and this is only possible within proc.c so we had to declare a *extern ps function* in sysproc.c (where the system call needs to be implemented) which is implemented in proc.c. The *extern* keyword in C allows you to do this, defining a function in one file and declaring it in another file.

The idea was suggested in a stack overflow post and looping through the table for each process we check if its id !=0 and then print it .All the syntax was taken from exit() system call inherent to xv6.

4.3 Implementing Inter Process Communication

We had to create system calls for communication between processes. In sysproc.c we implemented the system calls and we did that by calling helper after checking arguments

```

int
sys_send(int sender_pid, int rec_pid, void *msg)
{
  char* ptr = (char*)msg;
  if(argint(0, &sender_pid) < 0){  // a check just as the xv6 does
                return -1;}
  if(argint(1, &rec_pid) < 0){  // a check just as the xv6 does
                return -1;}
  if(argstr(2,&ptr) < 0){  // a check just as the xv6 does
                return -1;}
  return  sendp(sender_pid,rec_pid,ptr);     // helper function implemented in
    proc.c
}


int
sys_recv(void *msg)
{
   char* ptr = (char*)msg;
   if(argstr(0,&ptr) < 0){  // a check just as the xv6 does
                return -1;}
          return recvp(ptr);   // helper function implemented in proc.c
}
```

Listing 7: IPC methods implemented in sysproc.c

In proc.c we defined messages as message queue array i.e each row is a string array acting as mailbox for each process . Now string array in c is implemented as character matrix. We have also num_msg array which counts messages in mailbox for each process and also the index for next message in msg queue

```
//Adding mailboxes
char messages[NPROC][1][8];
int limit =1;                    // the mail box of different processes.
int num_msg[NPROC] = {0};
```

Listing 8: mailbox definition in proc.c

Now in the send process (sendp) ,we check if the msg queue is not full for receiving process and then add the msg in msg queue. Since the send is non-blocking we will not do block the sender process and if asleep wakeup the receiving process.

```
int
sendp(int send_pid , int rec_pid, char *msg){

     if(num_msg[rec_pid]==limit)
     {
      return -1;                          // this is the bounded buffer limit.
     }
     else
     {
  for (int i =0 ; i<MSGSIZE; i++)
  {
    messages[rec_pid][num_msg[rec_pid]][i] = *(msg+i);   // storing the message
    in the mailbox
  }
  num_msg[rec_pid]++;  // the next message to be stored in next location
  struct proc *pr= &ptable.proc[rec_pid];   // one needs to wakeup the recieving
    process if asleep
      wakeup(pr);                                  // wakeup function handles if the
     process wasn't sleeping
  return 0;
     }
}

int
recvp(char *msg)
{

  struct proc *curproc = myproc();     // This is used to get pid
  int rpid = curproc->pid;             // of the current process, learnt by grep "
    pid" *.c

  if(num_msg[rpid] == 0)
  {
    struct proc *pr;                        // we need to get current process from
    process table as sleep and wakeup should have same chan. no.
      pr = &ptable.proc[rpid];               // and as seen in sleep calls in
    other functions in this c file we normaly do that by passing the address of
    the process
    acquire(&ptable.lock);                  // and to make sure we are using the
    same pointer we access the process from ptable.
        sleep(pr, &ptable.lock);              // Sleep didm't work without
    acquiring locks expilicty although is does it internally - don't know why
          release(&ptable.lock);            // for correct usage of sleep refer -
    https://pdos.csail.mit.edu/6.828/2011/lec/l-coordination.html
  }
                                             // can't use else beacuse then
    the else block won't be able to execute and our message won't get stored in
    the return pointer
  num_msg[rpid]--;
  for(int i=0; i<MSGSIZE; i++)
   {
    *(msg+i)= messages[rpid][num_msg[rpid]][i];         // this is just saving
    the stored mesage into our input pointer
```

```
41    }
42    return 0;
43 }
```

Listing 9: implementation of send and recv helpers in proc.c

The receive is blocking it has to blocked until a msg is received that is if no msg is in msg queue then the kernel must remove the receiving process from running queue i.e we make the process sleep else we store the most recent msg from the queue and decrement the count

4.4 Distributed Algorithm using IPC

We had to create and use multiple processes to compute the sum of the elements in an array in a distributed manner. We had to create (assign) a coordinator process for collecting the partial sums from the rest of the processes, compute and print the sum.

```
1  void
2  sys_store(int a, int b)
3  {
4    if(argint(0,&a)<0){}
5    if(argint(1,&b)<0){}
6    partial_sum[a]=b;
7  }
8
9
10 int
11 sys_sums(void)
12 {
13   int res=0;
14   for(int i=0;i<7;i++)
15   {
16     res+=partial_sum[i];
17   }
18   return res;
19 }
```

Listing 10: helper system calls implemented in sysproc.c

For this reason 2 new system calls were added one which store partial sums in an array used by the non coordinator processes and finally one to compute entire sum used by the coordinator process.

Now in the user program the parent process is the main coordinating process Now we have used fork thrice so that the main process is the coordinating process and all other 7 processes are sub-processes .

For each sub-process an if -else block is there .Now in each of these blocks the sub-process calculates sum of the part of array assigned to it and then sends a message to coordinating process, implying that it's done before sending this message it stores the sum obtained by it in a shared memory using 'store' system call (implemented above) and then the sub-process exits.

In the first if-else block for parent the coordinating process waits for message of each of sub-process so there are 7 receives(blocking receive) in total for main process so once the process wakes up after 7 receives it means the shared memory is populated with the correct sums of each of sub-process so it uses 'sums' system call to get the final sum. In each block appropriate wait system calls have been added to prevent zombies.

```
1  int pid_par = getpid();
2          int a = fork();
3    int b = fork();
4    int c2 = fork();
5
6    if(a>0 && b>0 && c2>0)
7    {       // parent process -> also coordinating process
```

```
8     char *msg = (char *)malloc(MSGSIZE);
9     recv(msg);
10    recv(msg);                                                    // here the
      mechanism implemented is that the main process is the coordintaing process and
       all other 7 processes are
11    recv(msg);                                                    //  sub-processes
      each of the subprocess calculates sum of the part of array assigned to it and
      then sends a message to
12    recv(msg);                                                    // coordinating
      process, implying that it's done before sending this message it stores the sum
       obtained by it in a shared
13    recv(msg);                                                    // memory using '
      store' system call and then the subprocess exits so here the coordinating
      process waits for message of each
14    recv(msg);                                                    // of subprocess
      so total 7 recieves after recieving 7 such messages it means the shared memory
       is populated with the correct
15    recv(msg);                                                    // sums of each of
       subprocess so it uses 'sums' system call to get the final sum.
16    tot_sum = sums();
17    wait();
18    wait();                 // parent of 3 processes ... to prevent zombies
19    wait();
20  }
21  else if (a>0 && b>0 && c2==0)
22  {
23    int sum =0;   // partial sum
24    for (int i =0 ; i<143;i++)
25    {
26      sum += arr[i];
27    }
28    //printf(1," %d \n",sum);
29    char *msg_child = (char *)malloc(MSGSIZE);
30                msg_child = "1";
31    store(0,sum);
32                send(getpid(),pid_par,msg_child);                                //
      message acts as a signal to the coordintaing process that this subprocess is
      done, store stores the sum obtained by this
33    exit();                                                       // process.
34  }
35  // the other sub processes have similar sections of their own not shown in this
       snippet.
36
37      if(type==0){ //unicast sum
38    printf(1,"Sum of array for file %s is %d\n", filename,tot_sum);
39  }
40  exit();
```

Listing 11: implementation of distributed algorithm in assig1_8.c

4.5 Some specifications about implementations

- For the print_count function we have hardcoded the implementations as the name and number of system calls is not going to change and doing this manually can save the execution time spent on sorting (although it will not create much of a difference).
- For getting the active processes in the ps method we have considered all processes with pid $\neq 0$ as suggested in the help session.
- In the message passing section we have used a finite size mailbox which has a limit of storing 8 messages, which means that if the mailbox is full the *send* system call can even fail.

- For the distributed algorithm part we implemented two new system call that are not a part of problem statement to ease the implementation of the required functionality.
- For the distributed algorithm part the unicast primitives without any new system calls alone could have done the jab but that would have needed us to implement some string processing methods, so instead we choose this approach.