COL331 Operating Systems

Assignment 1 (Easy)

Pratik Nimbalkar (2020CS10607)

1. Installing and Testing xv6:
   I successfully installed the xv6 11 operating system and built qemu using the instructions given in the assignment pdf.
   The complete assignment of mine is built using xv6 and qemu. The task description of all parts is as follows:

2. System Calls

   a) System trace:
   I have implemented the system call sys_print_count in the sysproc.c file. It will print the list of system calls that have been invoked since the last transition to the TRACE ON state with their counts. Whenever the state changes from TRACE_OFF to TRACE_ON, sys_print_count will keep a count of the number of times a system call has been invoked since the state changed to TRACE_ON. This change of Trace from off to on and vice versa can be done by toggle() system call about which I have mentioned in the next part.

```
int
sys_print_count(void)
{
        for(int j=0; j<=27; j++)
        {
                if( sysc_count[alternate[j]-1] != 0 && alternate[j]!=23)
                        cprintf("sys_%s %d\n", sysc_name[alternate[j]-1], sysc_count[alternate[j]-1]);
        }


  return 0;
}

```

   This is my code for sys_print_count.
   This function prints the system trace, being maintained by the array defined inside the syscall.c file.
   Alternate is an array of 28 integers declared by me in sysproc.c as follows:
   extern int alternate[28];
   It is defined in syscall.c

The use of this alternate array is to print the system calls in alphabetically increasing order.

```
int sysc_count[28] = {0};
char sysc_name[28][20] = {"fork", "exit", "wait", "pipe", "read", "kill", "exec", "fstat", "chdir", "dup", "getpid", "sbrk", "sleep", "uptime", "open", "write", "mknod", "unlink", "link", "mkdir",
    "close", "toggle", "print_count", "add", "ps", "send", "recv", "send_multi"};

int alternate[28] = {24,9,21,10,7,2,1,8,11,6,19,20,17,15,4,23,25,5,27,12,26,28,13,22,18,14,3,16};
```

The sysc_name and sysc_count contains the list of syscalls and how many times a particular syscall is called respectively. They are defined in syscall.c as shown in the screenshot.

There are 28 system calls defined by me in syscall.h. Hence the alternate array is of 28 integers which show the system call numbers as per syscall.h in alphabetically increasing order, i.e. alternate[1] is 24 since the system call number of add (which is lexicographically smallest) and so on.

Also for each system call defined we need to declare it in syscall.h, syscall.c (in sysc_name array), usys.h, user.h.

1. Add the following in syscall.c.

```
extern int sys_print_count ( void ) ;
```

2. Add the following to the array of functions in syscall.c.

```
[ SYS_print_count ] sys_print_count
```

3. Add the following in syscall.h.

```
# define SYS_print_count 22
```

4. Add this is USYS.s

```
SYSCALL(print_count)
```

5. Add this is user.h

int print_count(void);

b) Toggling the tracing mode:

For this, we have to declare and define the sys toggle system call.

toggle() will be the user function that will call the system call.

sys toggle() that toggles the state:

if the state is TRACE ON sets it to TRACE OFF, and vice versa. As mentioned above, in all system calls we need to make changes in 4 files mentioned, hence I am mentioning only the other changes made by me in sysproc.c

Add the following in sysproc.c.

```
int
sys_toggle(void)
{
  if(trace==0) {
    for(int i=0; i<=27 ; i++)
        sysc_count[i] = 0;
    trace = 1;
    }
  else{
    trace = 0;
   }
  return trace;
}
```

If the trace is off, we make it on and make all system calls' occurrences as 0, i.e. sysc_count as 0 for all system calls.

If it is on, we make it off.

c) Add system call: sys add()

We define the system call sys_add. add() will be the user function that will call the system call. This system call takes two integer arguments and returns their sum. This is done as follows.

Add the following in sysproc.c

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

d) Process List: sys ps()

We define the sys_ps() system call in sysproc.c. Here we only add the ps() function that will print the list of all the current running processes. I have defined ps() in proc.c

This is done as follows.

1. Let's define the sys ps system call. The definition will be given in sysproc.c.

```
int
sys_ps(void)
{
        return ps();
}
```

2. The sys ps system call used a function ps() which does the main job of printing the process name and id. The definition will be given in proc.c.

```c
int
ps()
{
  struct proc *curr;
  acquire(&ptable.lock);
  for(curr=ptable.proc; curr < &ptable.proc[NPROC]; curr++)
  {
    if(curr->pid!=UNUSED)
      cprintf("pid:%d name:%s\n", curr->pid, curr->name);
  }
  release(&ptable.lock);
  return 0;
}
```

Here I am also using acquire lock before starting the loop and release lock so that it does not get interrupted by any other process. Hence the usage of lock is to avoid smooth run of ps().

2. Inter Process Communication

   a. Unicast

      In this section, we implement the unicast method for Inter-Process Communication. For this we have two system calls sys send() and sys recv() in sysproc.c

      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.

```c

int
sys_send(void)
{
  int sender_pid, receiver_pid;
  char *msg;

  if((argint(0, &sender_pid) < 0)||(argint(1, &receiver_pid) < 0)||(argstr(2, &msg) < 0)){
    return -1;
  send_msg(sender_pid,receiver_pid, msg);
  return 0;

}

int
sys_recv(void)
{
  char *msg;

  if(argstr(0, &msg) < 0){
    return -1;
}

  rcv_msg(msg);
  return 0;

}
```

      The sys_send system call is taking the sender pid, receiver pid and the message which needs to be sent as the input.

In the if condition, I am checking the validity of these 3 input parameters, i.e. if pid <0 (invalid pid) or invalid message, then return -1.
Then I am calling the send_msg function which is defined in proc.c
The sys_recv system call is only taking the message as input. After validating it, I am calling the recv_msg function.

The send_msg and rcv_msg functions are defined in proc.c

```
void
send_msg(int sender_pid, int rec_pid, char *msg)
{


  strncpy(messages[rec_pid], msg, 8);

  present[rec_pid]=1;


  struct proc *p;


    acquire(&ptable.lock);
    for(p=ptable.proc;p<&ptable.proc[NPROC];p++)
    {
        if(p -> pid==rec_pid)
        {
        break;
        }
    }
    if(p->state==SLEEPING)
    p->state = RUNNABLE;
    release(&ptable.lock);


 }
void
rcv_msg(char *msg)
{

  int rec_pid = myproc()->pid;
  if(present[rec_pid]!=1)
  {

  struct proc *p;
    p = &ptable.proc[rec_pid];

    acquire(&ptable.lock);
    sleep(&p, &ptable.lock);
    release(&ptable.lock);
  }



  strncpy(msg, messages[rec_pid], 8);
  present[rec_pid]=0;


}
```

The strncpy is used to copy the message (store the message in a buffer) which will later be used by the receiver to receive.
I am using the acquire lock and release lock functionality here as well so that it does not get interrupted by any other process.
Basically the lock is used to ensure no other send/recv call can access the buffers during this time. Hence the usage of lock is to avoid smooth run of the functions.

Send_msg function takes receiver pid, sender pid and message as input. It copies the messages to a buffer.
The rec() function takes one char * argument.
The message is written to this location. Locks are implemented here as well.

b. Multicast
The multicast model sends data from a sender to multiple processes using the sys send multi system call. I have implemented the multicast communication model using the
sys recv system call instead of a signal handler.
The signature of the system call is:
int sys_send_multi(int sender_pid, int rec_pids[], void *msg)

We define the system call sys_send_multi in sysproc.c which takes the following arguments:
Sender pid - integer
Array of receiver pids - array of integers
Message which needs to be sent - char
Argint, argptr are used as usual.
The number of receivers have been set as 8 (according to piazza post), thus argptr(1, &rec_multi, 8) and the message sent is 8 bytes (according to the assignment), thus argptr(2, &msg, 8).
Then we call the send_msg_multi function which is defined in proc.c

```
int
sys_send_multi(void)
{

  int sender_pid;
  int *rec_pids;
  char *msg;

  argint(0, &sender_pid);
  argptr(2, &msg, 8);
  void* rec_multi;
  argptr(1, &rec_multi, 8);
  rec_pids = (int*) rec_multi;


  send_msg_multi(sender_pid, rec_pids,msg);
  return 0;
}
```

The send_msg_multi function:

It takes the sender pid, receiver pid and message as input.

Here we know that the bytes of a message which is sent from receiver to sender = 8 bytes.

Hence we loop from int i =0 to 7.

If the receiver is valid, we copy the message to the messages buffer mapping the message to the accurate receiver using strncpy.

Then again the lock is used to ensure no other send/recv call can access the buffers during this time.

```
void
send_msg_multi(int sender_pid, int *rec_pid, char *msg)
{


  for(int i=0; i<8;i++){
  if(*(rec_pid+i)>=0){
  strncpy(messages[*(rec_pid+i)], msg, 8);

  present[*(rec_pid+i)]=1;


  struct proc *p;


    acquire(&ptable.lock);
    for(p=ptable.proc;p<&ptable.proc[NPROC];p++)
    {
        if(p -> pid== *(rec_pid+i))
        {
        break;
        }
    }
    if(p->state==SLEEPING)
    p->state = RUNNABLE;
    release(&ptable.lock);

    }
  }

 }
```

- Other:
  I have mentioned the additions made by me in various files here:

  1. In syscall.h:

```
# define SYS_toggle 22
# define SYS_print_count 23
# define SYS_add 24
# define SYS_ps 25
# define SYS_send 26
# define SYS_recv 27
# define SYS_send_multi 28
```

  2. In syscall.c:

```
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 sys_send_multi ( void ) ;
// inside the array
[ SYS_toggle ] sys_toggle ,
[ SYS_print_count ] sys_print_count ,
[ SYS_add ] sys_add ,
[ SYS_ps ] sys_ps ,
[ SYS_send ] sys_send ,
[ SYS_recv ] sys_recv ,
[ SYS_send_multi ] sys_send_multi ,
};
```

  3. In usys.s

```
SYSCALL ( print_count )
SYSCALL ( toggle )
```

SYSCALL (add)
SYSCALL (ps)
SYSCALL (send)
SYSCALL (recv)
SYSCALL (send_multi)

4. In makefile

```
//  ( in UPROGS =\)
_assig1_1 \
_assig1_2 \
_assig1_3 \
_assig1_4 \
_assig1_5 \
_assig1_6 \
_assig1_7 \
_assig1_8 \
____ _____ ______ __
\\  ( fs . img :)
fs . img : mkfs README arr $ ( UPROGS )
./ mkfs fs . img README arr $ ( UPROGS )

// ( in EXTRA =\)
_assig1_1 . c _assig1_2 . c _assig1_3 . c _assig1_4 . c _assig1_5 . c
_assig1_6 . c _assig1_7 . c _assig1_8 . c \
```

5. In user.h

```
// under " system calls "
int print_count ( void ) ;
int toggle ( void ) ;
int add ( int , int ) ;
int ps ( void ) ;
int send ( int , int , void *) ;
int recv ( void *) ;
int send_multi ( int , int * , void * , int ) ;
```

- Output:
  Outputs for all tasks:

  1. Part 1
  sys_fork 1
  sys_write 18

  2. Part 2
  sys_close 1
  sys_open 1

  3. Part 3
  sum of 2 and 3 is : 5

  4. Part 4
  sum of 10 and -6 is : 4

  5. Part 5
  pid :1 name : init
  pid :2 name : sh
  pid :3 name : assig1_5

  6. Part 6
  pid:1 name:init
  pid:2 name:sh
  pid:9 name:assig1_6
  pid:10 name:assig1_6
  pid:1 name:init
  pid:2 name:sh
  pid:9 name:assig1_6
  pid:10 name:assig1_6

  7. Part 7
  1 PARENT : msg sent is : P
  2 CHILD : msg recv is : P