Indian Institute of Technology Delhi

COL331 Operating Systems : Assignment 1

Ishita Chaudhary: 2019CS10360

Contents

1 Installing and Testing xv6 **2**

2 System Calls **3**

3 Inter-Process Communication **6**

4 Distributed Algorithms **10**

5 Testing **11**

Installing and Testing xv6

Before doing the following parts, xv6 version 11 was installed from https://pdos.csail.mit.edu/6.828/2018/xv6.ht
To run the xv6 OS on QEMU virtual machine; I installed qemu on my local machine. It was tested by running the commands *make* and *make qemu*. The following parts were implemented and tested on the local machine (Ubuntu version 20.04) as well.

System Calls

To add a new system call of type *sys_name*, the following changes were made in the respective files:
1. usys.S: we add the line "SYS(name)"
2. user.h: add the line "int name(signature of parameters separated by comma)", this file contains the function signature of all system calls.
3. syscall.c: add "extern int sys_name(void)" and "[SYS_name] sys_name" where other system calls are defined in the syscalls[] array. This is where we give the pointer to the syscall function.
4. syscall.h: add "#define SYS_name *custom_integer*", this maps the system call to a unique number which becomes its id
5. sysproc.c: this is where the function of each system call is implemented. The syscall() function in syscall.c calls these functions. We will discuss the implementation later for each system call in detail.

Since in these system calls, we also need to make a user-level program to test the system call; we make a file "name.c" where we call the system call.
6. Makefile: to compile this user-level program, we add the flag "_name" in UPROGS and name.c in EXTRA flag of makefile.

To trace system calls, we implemented two system calls: toggle and print_count. We declared an extern variable "trace_on" which depicts if a trace is on or off, and an extern integer array "count_syscall" which maintains the count of each system call. These are declared in syscall.c because that is where the system calls are called. We use the keyword "extern" because we need to access them in sysproc.c (outside of syscall.c).
sys_toggle: This toggles the trace from ON to OFF and vice versa whenever it is called. We trace the system calls only when the trace is ON. Whenever we turn the trace OFF, we flush all the recorded system calls in *count_syscall*.

sys_print_count: To print the recorded system calls alphabetically, we record them in an alphabetical manner. In syscall.c, we increment the index of count_syscall for a system call in an alphabetical manner. We can do this by maintaining the mapping between the IDs of the system calls and their alphabetical number, as the IDs are fixed once when the kernel is booted. Whenever a system call is executed, the function syscall() in syscall.c is called. Hence, we do this inside the syscall function, after checking if the trace is ON. We do not record counts of "toggle" and "print_count" in this process.

sys_add: This is a basic implementation of the addition of two integers. We use "argint" function defined in xv6 to take kernel space input from user space.

sys_ps: This function prints out the process name and its pid for all processes that are created on the system, irrespective of the state they are in. In xv6, a process can be in one of the following states: UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE. We print all the processes which are not in UNUSED state. For this, we need to iterate over the table, where all processes are stored. And we can't access ptable outside of proc.c, hence we create a helper function in proc.c which is called in the sys_ps implementation of sysproc.c. We acquire the lock of the table, so that it is not edited by another process while we're iterating

```c
int sys_toggle(void)
{

    if(trace_on==1){
      trace_on=0;
      for(int i=0;i<28;i++) count_syscall[i]=0;
    }
    else if(trace_on==0) trace_on=1;
    return 0;
}
```

```c
int sys_add(int num1, int num2)
{
    //int num1, num2;
    argint(0,&num1);
    argint(1, &num2);

    //cprintf("%d - is the sum of the numbers:",num1+num2);
    return num1+num2;

}
```

over it. We release this lock once we're done. The syntax is taken from other system calls defined in xv6 which access the ptable.
Since we made a helper function in proc.c, we need to include the function signature in defs.h.

```c
//process list function, called in ps.c
int ps(void){
  struct proc *p;
  acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state == RUNNING || p->state == RUNNABLE || p->state == EMBRYO || p->state == ZOMBIE ||
      p->state == SLEEPING) cprintf("pid:%d name:%s\n", p->pid, p->name);
        //cprintf("pid:%d name:%s\n", p->pid, p->name);
    }
    release(&ptable.lock);
    return 0;
}
```

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
    if(trace_on==1){
      if(num==1) count_syscall[6]++;
      else if(num==2) count_syscall[4]++;
      else if(num==3) count_syscall[26]++;
      else if(num==4) count_syscall[14]++;
      else if(num==5) count_syscall[17]++;
      else if(num==6) count_syscall[9]++;
      else if(num==7) count_syscall[5]++;
      else if(num==8) count_syscall[7]++;
      else if(num==9) count_syscall[1]++;
      else if(num==10) count_syscall[3]++;
      else if(num==11) count_syscall[8]++;
      else if(num==12) count_syscall[19]++;
      else if(num==13) count_syscall[22]++;
      else if(num==14) count_syscall[25]++;
      else if(num==15) count_syscall[13]++;
      else if(num==16) count_syscall[27]++;
      else if(num==17) count_syscall[12]++;
      else if(num==18) count_syscall[24]++;
      else if(num==19) count_syscall[10]++;
      else if(num==20) count_syscall[11]++;
      else if(num==21) count_syscall[2]++;
      else if(num==22) count_syscall[0]++;
      else if(num==23) count_syscall[16]++;
      //else if(num==24) count_syscall[15]++; print_count
      //else if(num==25) count_syscall[23]++; toggle
      else if(num==26) count_syscall[20]++;
      else if(num==27) count_syscall[18]++;
      else if(num==28) count_syscall[21]++;
    }
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
```

Inter-Process Communication

In this part, we create system calls to communicate between processes. We have two models here: unicast and multicast. We declare and define system call signatures as described in the previous part. We do not make any changes to the makefile in this case, as we do not make any user-level programs to test these system calls.

Unicasting
In this model, one process can send a message to only one process at a given time using the "sys_send" and "sys_recv" calls. As the name suggests, the send system call takes two pids as input, the sender's and the receiver's, and another field that takes the pointer to the message which is to be sent from the sender pid to the receiver pid. Whereas the recv system call takes a buffer as an argument in which it will store the message.
There are a few constraints to the system, such as:
1. The length of the message cannot be more than 8 bytes.
2. In case a process receives multiple messages from different processes, it can only receive one message at a time. Hence, we maintain a queue for messages that cannot hold more than 10 messages.
These calls return 0 in case of success and -1 if it fails. We *assume* for the simplicity of implementation that the pids of the processes created will always be less than NPROC (which is 64 in xv6); otherwise, we would have to iterate over the ptable to find the appropriate position of a pid, which is dynamic.
In sysproc.c, we created a queue for all pids, with size =10, and each element will contain the sender pid and the message sent (a char array of size 8). We use a spinlock recv_queue_lock, while changing a queue so that any other process is not able to make any other changes simultaneously. We also maintain an integer array queue_size which keeps track of the size of the queue for each pid and gives us the index of the next available space.
We created a init_recv_queue() function, which allocates memory to the data structures described above. In this, we initialize the recv_queue_lock, set the default value of sender pid to -1 and messages to ", and queue_size of all pids to zero. We call this init_recv_queue() function in main.c inside its main() function, which is called when the kernel is booted. And since we want to call this function outside of sysproc.c, its signature is defined in defs.h.

In the send function, after various check assertions, if the queue for receiving pid is already full, we return -1. Otherwise, we acquire the recv_queue_lock; we update the message and sender pid to the next free space in queue, and increment queue_size. Finally, we release the lock.

In the case of recv function, we get the pid of the receiver using myproc()-¿pid, to help us get the message from the queue referred by pid. Since this is a blocking call, the initial recv flag is set to 0, and while this flag is zero, the function will repeatedly check if there's a message for this pid by checking the size of the queue for this pid. If there's a message, we acquire the recv_queue_lock, flip the recv flag to 1, and write the first message to the buffer, which the function takes as input. We shift all other messages from position i to i-1, decrease the size of the queue by one and release the lock.

Multicasting
In this model, a sender pid can send a message to multiple pids at the same time. We implement another system call "sys_send_multi", which takes the pid of the sender, a pointer to the message

```c
int sys_send(int sender_pid, int rec_pid, void* msg)
{
    // check assertions
    if(argint(0,&sender_pid)<0) return -1;
    if(argint(1, &rec_pid)<0) return -1;
    // Assumption: The pids are less than NPROC
    if(sender_pid>=NPROC || rec_pid>=NPROC) return -1;
    if(sender_pid<0 ||rec_pid<0) return -1;

    // check if the queue is full
    if (queue_size[rec_pid]==max_queue_size) return -1;


    char* complete_message = (char*)msg;
    if (argptr(2, &complete_message, max_msg_size)<0)
      return -1;
    acquire(&recv_queue_lock);
    char* qmsg = recv_queue[rec_pid][queue_size[rec_pid]].msg;
    do {
      *qmsg++ = *complete_message;

    } while (*complete_message++ != '\0');
    recv_queue[rec_pid][queue_size[rec_pid]].sender_pid=sender_pid;
    queue_size[rec_pid]++;
    // cprintf("sender_pid: %d, rec_pid: %d, msg: %s\n",sender_pid, rec_pid, qmsg);
    // cprintf("queue_size: %d\n",queue_size[rec_pid]);
    release(&recv_queue_lock);
    return 0;
}
```

to be sent, and an integer array of the receiver's pids as its parameters. I did not implement it with an interrupt (as mentioned on piazza) but used sys_recv system call to receive messages.

In addition to the previous constraints, the send_multi system call cannot send a message to more than 8 pids at a time. Since a valid pid is denoted by a positive integer, we iterate over the receiving pids and if a pid is positive, we call the helper function "multi_helper" inside "sys_send_multi" function. We did not use the "sys_send" function itself because it takes argument from user space, but in this case we are providing the inputs from the kernel space. Hence, the "multi_helper" function is identical to "sys_send" function, except the fact that in the helper function we're providing the arguments from the kernel space.

```
int sys_recv(void* msg)
{
    char* recv_msg = (char*)msg;
    if(argptr(0, &recv_msg, max_msg_size)<0) return -1;
    int rec_pid=myproc()->pid;
    // cprintf("I am %d\n", rec_pid);
    int recv=0;
    char* to_write = recv_msg;
    while(!recv){
      acquire(&recv_queue_lock);
      // cprintf("Q size %d\n",queue_size[rec_pid]);
      if(queue_size[rec_pid]!=0){

        recv=1;

        char* qmsg = recv_queue[rec_pid][0].msg;
        do {
          *to_write++ = *qmsg;
        } while (*qmsg++ != '\0');

        for(int i=0;i<queue_size[rec_pid]-1;i++){
          recv_queue[rec_pid][i]=recv_queue[rec_pid][i+1];
        }
        queue_size[rec_pid]--;
      }
      release(&recv_queue_lock);
    }

    return 0;

}
```

```c
int multi_helper(int sender_pid, int rec_pid, void* msg)
{
    // Assumption: The pids are less than NPROC
    if(sender_pid>=NPROC || rec_pid>=NPROC) return -1;
    // if(argptr(2, (void*)&msg, max_msg_size)<0) return -1;
    if(sender_pid<0 ||rec_pid<0) return -1;
    // check if the queue is full
    if (queue_size[rec_pid]==max_queue_size) return -1;
    char* complete_message = (char*)msg;
    acquire(&recv_queue_lock);
    char* qmsg = recv_queue[rec_pid][queue_size[rec_pid]].msg;
    do {
      *qmsg++ = *complete_message;
    } while (*complete_message++ != '\0');
    queue_size[rec_pid]++;
    release(&recv_queue_lock);
    return 0;
}

int sys_send_multi(int sender_pid, int rec_pids[], void *msg)
{
  if(argint(0,&sender_pid)<0) return -1;
  if (argptr(1, (void*)&rec_pids, 8 * sizeof(int)) < 0) return -1;
  char *broadcast_msg = (char*)msg;
  if (argptr(2, &broadcast_msg, max_msg_size) < 0) return -1;
  for(int i = 0; i < 8; i++){
    if (rec_pids[i]<0) continue;
    multi_helper(myproc()->pid, rec_pids[i], broadcast_msg);
  }
  return 0;
}
```

Distributed Algorithms

In this part, we 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.
The task is to calculate the sum of 1000 elements stored in an array each of which is from 0 to 9. We implement two versions for this task which are explained below.

Unicast Version (type = 0): In this version, the main/coordinator process forks 8 worker processes and then uses the *send* system call to assign a fraction of the array to each worker process. The worker processes first receive their assigned partition index using the *recv* system call, calculate the partial sums corresponding to assigned partitions and then send the partial sums back to the coordinator process using *send* system call. The workers can now exit.
The coordinator process then uses the *recv* system call to receive the partial sums sent by each of the worker processes and adds them to obtain the total sum of the given array. The coordinator process must also call *wait()* for each of the children to avoid formation of Zombie processes.

Multicast Version (type = 1): In this version, we perform all the steps done in the unicast version but the workers do not exit after sending their partial sums. The controller process calculates the mean and then uses the *send_multi* system call to multicast the mean to all worker processes. The worker processes again use the *recv* system call to receive the mean, compute the sum of the squares of the differences about the mean and send back the computed partial sum of squares to the coordinator process using *send* system call. The workers can now exit.
The coordinator finally uses the *recv* system call to receive all the partial sum of squares, calculates the variance and prints it to console. Similar to the previous version, the coordinator process calls *wait()* for each of the children to avoid formation of Zombie processes.

Testing

As instructed, we use user programs to validate the system calls implemented as part of the asssignment. The outputs of all the test cases, assig1_1 to assig1_8 were found to be matching with the expected outputs as provided.