ASSIGNMENT 1 REPORT

VINIT CHANDAK

15/02/2023

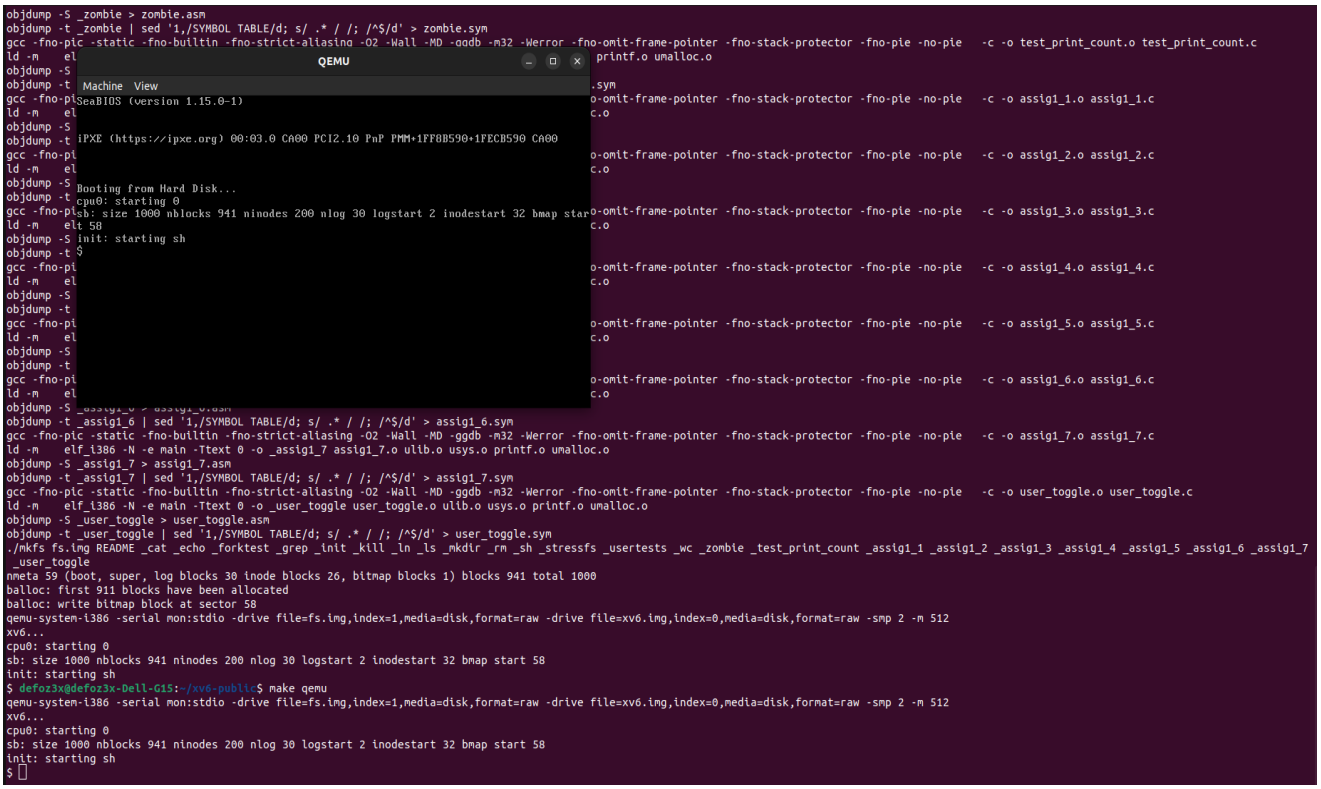
—

OPERATING SYSTEMS (ELL783)

—

Prof. Smruti Sarangi

1. Installing and Testing xv6



2. System Calls

How to add system calls to xv6?

- We need to make changes to the following files (at a minimum).
  - `sysproc.c`: Add the real implementation of our method here.
  - `syscall.c`: System call is a function. We need to add a pointer to our system call function in this file.
  - `syscall.h`: Allocate a unique id to our system call.
  - `user.h`: Add an entry so that the system call is visible to the user programs.
  - `usys.S`: We need to add an entry keeping with the convention so that assembly stub can move the system call number into reg eax and make the system call.

Any user mode program that wants to invoke a system call needs to include (atleast):

```
#include "types.h"
#include "stat.h"
#include "user.h"
```

1. System call toggle():

   Implemented sys_toggle() - changes the state from TRACE_OFF to TRACE_ON and vice versa. Created a trace_state data type as below.

   ```
   enum trace_state{TRACE_OFF=0, TRACE_ON=1}
   ```

   Using the trace_state data type, created an `int trace_flag = TRACE_OFF` which is initialized to TRACE_OFF by default. Initially, the count for all the system calls is 0 and whenever we transition from TRACE_ON to TRACE_OFF the count for all the system calls is reset to 0.

2. System call print_count():

   Implemented system call print_count() – prints a list of system calls and the number of times they were invoked since the last transition to TRACE_ON. The list is printed in alphabetical order. I have maintained three different arrays to implement this system call. One array (`int syscallcount []`) stores the count of the number of times each system call was invoked, one array (`static int alphabeticalsyscallmap []`) is used to print the system calls in alphabetical order and the last array (`static char* syscallnames []`) is used to print the system call names. For every system call at some point in its path, the function syscall() is called. If the tracing is on then I increment the count for the corresponding system call from there.

3. System call add():

   This system call takes two integer arguments and returns their sum. The implementation is straightforward. The main function body is in sysproc.c which calls a helper function defined in proc.c.

4. System call ps():

   Implemented ps() system call which prints the list of all current running processes in a specific format. The main function body is present in proc.c. I traverse over ptable (array of PCB's) and prints the

details of all the processes whose state is either 'RUNNING' or 'RUNNABLE'.

3. Inter Process Communication

1. Unicast Communication Model:

In the unicast model for IPC that I implemented, the processes communicate with each other using a shared buffer. I have created a buffer structure as follows:

```
struct buffer_struct {
        char buffer[MSGSIZE];
        int bufferstatus;
        int sender_id;
        struct spinlock bufferlock;
};
```

Then I created an array of 65 such buffers i.e `struct buffer_struct buffers[65]` (one for/associated with each process as the maximum number of processes in xv6 is 64). Every buffer has it's own lock. "bufferstatus" indicates whether the buffer is empty or not i.e. whether there is some data in the buffer that is yet to be read.

Two system calls as follows were implemented to achieve this:

```
int send(int s_id, int r_id, void *message)
```

In this call, the sending process first acquires the lock on the buffer associated with the receiver process. If the buffer is empty, it releases the lock and returns 1. Otherwise, it puts the message into the buffer, sets the sender id and buffer status fields appropriately, releases the lock and returns 0.

```
int recv(void *message)
```

In this function call, again the lock associated with the processes buffer is obtained first. Then the bufferstatus is check to determine if some data has been written to the buffer or not. If no data is written i.e. (bufferstatus == 0) is true, then the process acquires sleeplock and goes to sleep until it is woken up by a process(sender) after it (sender) has added some data to the buffer. It then goes on to read the data. Otherwise, the message is read from the buffer, bufferstatus is set to 1, bufferlock is released and 0 is returned.

Both the function's in sysproc.c call their helper functions in proc.c where all these things are implemented.

2. Multicast Communication Model

Since we were later allowed to use sys_send() and sys_recv() instead of the signal handler mechanism, the implementation of multicast model builds on top of the unicast model. The system call sys_send_multi() was implemented as follows:

```
int sys_send_multi (int, int *, void *);
```

I first take the arguments as input. Number of receiving process is a constant which is set to 8. This is because as array reduces to a pointer in C when passing to a function and it is not possible to find out the number of elements in the array using this pointer.

Then I simply send the message to each of the receiving process using sys_send() system call. This puts the message into the buffer of each receiving process from where it can be read and wakes up the receiving process thereby working as a multicast communication model.

4. Distributed Algorithm

Calculating Total Sum:

We need to calculate the sum of 1000 elements such that there is a coordinator process which divides the task between 8 processes, each of which compute their own partial sum and send it the coordinator process which then returns the total sum by adding all the partial sums.
First, I initialize some variables as follows:

```
int noOfProcesses = 8;                          - no of child processes
int pidParent = getpid();                       - pid of coordinator process
int i, j, cid;                                  - loop variables, temp variable to store child pid
int elmForEachProc = size/noOfProcesses;        - no of elements each process needs to add
void *partialSumP = (void *)malloc(MSGSIZE);    - used to store partial sum received by parent
void *pmsg = (void *)malloc(MSGSIZE);           - used to send the partial sum to the parent
int partialSum=0;                               - used to compute partial sum
```

Then, I have a **for** loop with number of iteration equal to the number of child processes. In each iteration of the **for** loop, I fork() the parent

process. Each of the child processes has a copy of parent processes address space (no copy_on_write() in xv6). This is essential to compute the partial sums. For each of the child process(one child does its job every iteration), *curStart* and *curEnd* is computed which indicates the start and end indexes of the array elements whose partial sum it needs to compute. Each child then computes the partial sum, sends it to the coordinator process using sys_send() system call and calls exit(). In the same iteration, the parent process first calls wait() and waits for the child process to exit(). It then reads the partial sum send by that child using sys_recv() system call and adds it to the total.

Thus, in each iteration, one child computes its partial sum, sends it to the coordinator process which adds it to the total sum. By the end of the **for** loop, the variable *tot_sum* contains the sum of all elements as required.

```
for(i=0; i<noOfProcesses; i++){
    cid = fork();
    if(cid == 0){
        int curStart = i*elmForEachProc;
        int curEnd = (i+1)*elmForEachProc;
        partialSum=0;
        for(j=curStart; j<curEnd; j++){
            partialSum += (short)arr[j];
        }
        //printf(1, "partial sum = %d ", partialSum);
        //printf(1, "%d %d ", curStart, curEnd);
        //printf(1, "%d \n",getpid());
        pmsg = (void *)(&partialSum);
        send(getpid(), pidParent, (void *)pmsg);
        //updating local copy?
        //curStart+=elmForEachProc;
        //curEnd+=elmForEachProc;
        exit();
    }
    else{
        wait();
        //printf(1, "%d ",getpid());
        recv((void *)partialSumP);
        //printf(1, "%d \n", *(short *)(partialSumP));
        tot_sum+=*(short *)partialSumP;
    }

}
```

Calculating Variance:

`int indexArr[noOfProcesses+1]` array is used to determine what elements a process needs to process. `int children[noOfProcesses]` array stores the pid's of the child processes. We need this array to send back the mean to child processes. I have a nested block of if-else statements. Briefly, the program works as follows:

We fork() the original parent process, the child executes some code and the parent recursively fork()'s into child processes until we have 8 children. **Since we don't have signaling mechanism, it is not possible to**

wake up a particular blocked process. Hence, we cannot use the for-loop approach as above. Therefore, we use the sleep and wait system calls with variable time period for sleep so that the coordinator process can get every processe's partial sum and read it.

Every child process computes its partial sum and sends it to the coordinator process. It then sleeps for some time with the sleep time gradually increasing as we go down the nested if-else block. This is needed as we don't have a queue and our buffer can only hold one message at a time. So, the coordinator process has to read every child's partial sum before it gets the next child's sum). This works as every time a child process sends its partial sum to the coordinator process, the send() system calls wakes up the receiving process(coordinator in this case). After the coordinator process computes the total sum and mean, it sends the mean to all the children processes using the **send_multi()** system call which again wakes up the child processes if they're asleep. Each of the child processes then compute the sum of squares of the differences about the mean and send it to the coordinator process. Again, before sending each child sleeps for some amount of time (increasing as we go down the if-else blocks). This along with wait() call properly synchronizes and gives the coordinator process enough time to read the values sent by every child. After sending, the child process exits by calling exit(). The coordinator process reads the value sent by the child and adds it to a temp variable. After getting the values from all the child processes, the variance is computed using this temporary variable.

All the critical regions are properly synchronized using spinlocks for all the system calls.

NOTE:- The program takes around 12 seconds to compute the variance.

Code for one of the child processes:

```
children[6]=cid7;
int cid8 = fork();
if(cid8 == 0){
    partialSum=0;
    for(j=indexArr[7]; j<indexArr[8]; j++){
        partialSum += (short)arr[j];
    }
    pmsg = (void *)(&partialSum);
    sleep(800);
    send(getpid(), pidParent, (void *)pmsg);
    void *mean = (void *)malloc(MSGSIZE);
    recv((void *)mean);
    float var=0;
    int k;
    for(k=indexArr[7]; k<indexArr[8]; k++){
        var += (arr[k]-*(float *)mean) * (arr[k]-*(float *)mean);
    }
    //printf(1, "%d ", var);
    void *ret_var = (void *)malloc(MSGSIZE);
    ret_var = (void *)(&var);
    sleep(800);
    //printf(1,"huh");
    send(getpid(), pidParent, (void *)ret_var);
    //printf(1,"huh");
    exit();
```

Code for the parent process:

```
children[7]=cid8;
sleep(10);
void *partialSumP1 = (void *)malloc(MSGSIZE);
void *partialSumP2 = (void *)malloc(MSGSIZE);
void *partialSumP3 = (void *)malloc(MSGSIZE);
void *partialSumP4 = (void *)malloc(MSGSIZE);
void *partialSumP5 = (void *)malloc(MSGSIZE);
void *partialSumP6 = (void *)malloc(MSGSIZE);
void *partialSumP7 = (void *)malloc(MSGSIZE);
void *partialSumP8 = (void *)malloc(MSGSIZE);
recv((void *)partialSumP1);
//printf(1, "%d \n", *(short *)(partialSumP1));
tot_sum+=*(short *)partialSumP1;
recv((void *)partialSumP2);
//printf(1, "%d \n", *(short *)(partialSumP2));
tot_sum+=*(short *)partialSumP2;
recv((void *)partialSumP3);
//printf(1, "%d \n", *(short *)(partialSumP3));
tot_sum+=*(short *)partialSumP3;
recv((void *)partialSumP4);
//printf(1, "%d \n", *(short *)(partialSumP4));
tot_sum+=*(short *)partialSumP4;
recv((void *)partialSumP5);
//printf(1, "%d \n", *(short *)(partialSumP5));
tot_sum+=*(short *)partialSumP5;
recv((void *)partialSumP6);
//printf(1, "%d \n", *(short *)(partialSumP6));
tot_sum+=*(short *)partialSumP6;
recv((void *)partialSumP7);
//printf(1, "%d \n", *(short *)(partialSumP7));
tot_sum+=*(short *)partialSumP7;
recv((void *)partialSumP8);
//printf(1, "%d \n", *(short *)(partialSumP8));
```

```
tot_sum+=*(short *)partialSumP8;
float p_mean = (float)tot_sum/(float)size;
void *multi_msg = (void *)malloc(MSGSIZE);
multi_msg=(void *)(&p_mean);
send_multi(getpid(), children, (void *)multi_msg);
//sleep(200);
void *get_var = (void *)malloc(MSGSIZE);
int v_v=0;
for(i=0; i<noOfProcesses; i++){
    wait();
    recv((void *)get_var);
    v_v += *(float *)get_var;
}
variance = v_v/(float)size;
```

Sum is coming out to be 4545 and variance is 8.41.

5. EXTRA STUFF AND NOTES

- Implemented a program to test multicast IPC model and system call send_multi(). It is present in the file "multicast_test.c". Uncomment the required code before running. Also set the no of receiving process in the function body of the system call send_multi() accordingly(set it to 3 if no changes are made in the code).

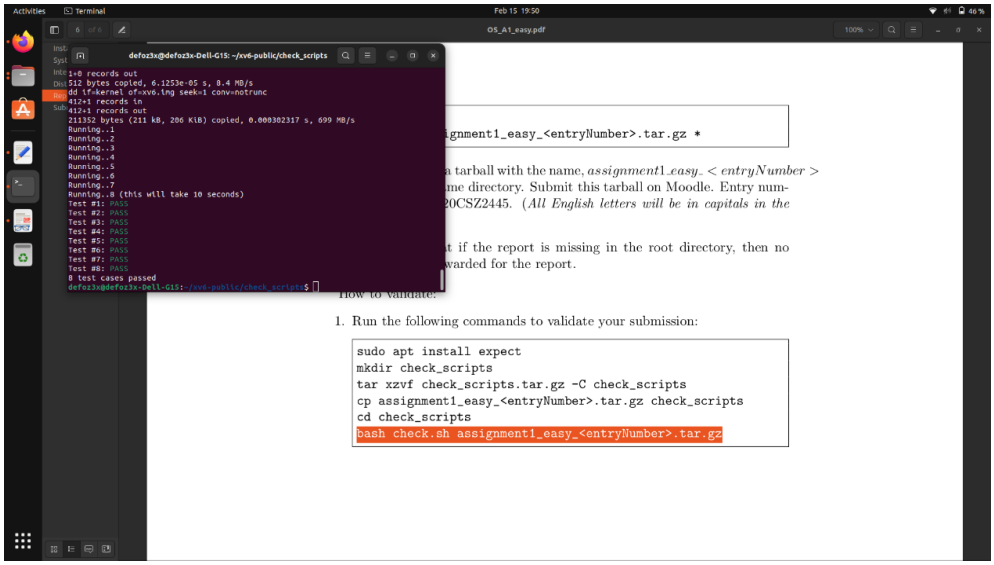
- For print_count() system call, I've ignored the counts of toggle() and print_count() as told on piazza. The functionality to print this can be easily added by making appropriate changes to the array's mentioned above in the print_count()'s section.

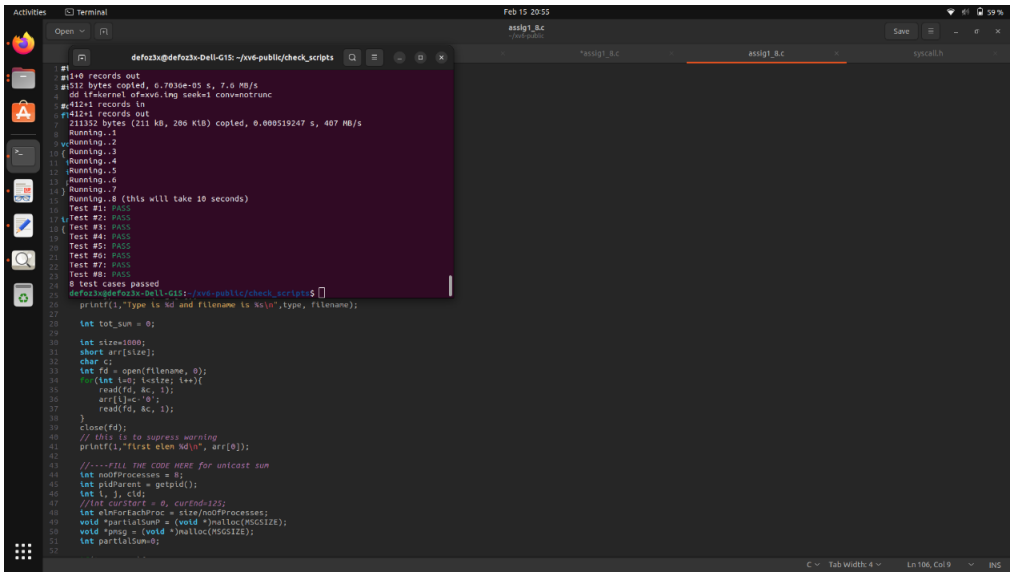
- I've implemented both blocking and non-blocking versions of the system call sys_recv(). Uncomment the appropriate code.

- A lot of code lines are commented, which can be used for looking at the control flow.

- The distributed program might take more than 10 seconds to compute the variance because of the sleep calls.

- All the test cases passed after running the script, I've attached the outputs below, screenshots taken at two different times and show my laptop/username(defoz3x):





Output of variance computation: