Indian Institute of Technology Delhi

Operating System

ELL 783

Assignment 1

Name:	Atul Kumar Rana
Entry No:	2023EET2195
Course Instructor:	Dr. Smruti R. Sarangi

1 Installing and testing xv6

I. **Download xv6 Source Code:**

```
git clone https://github.com/mit-pdos/xv6-public.git
```

II. **Extract the Source Code:**

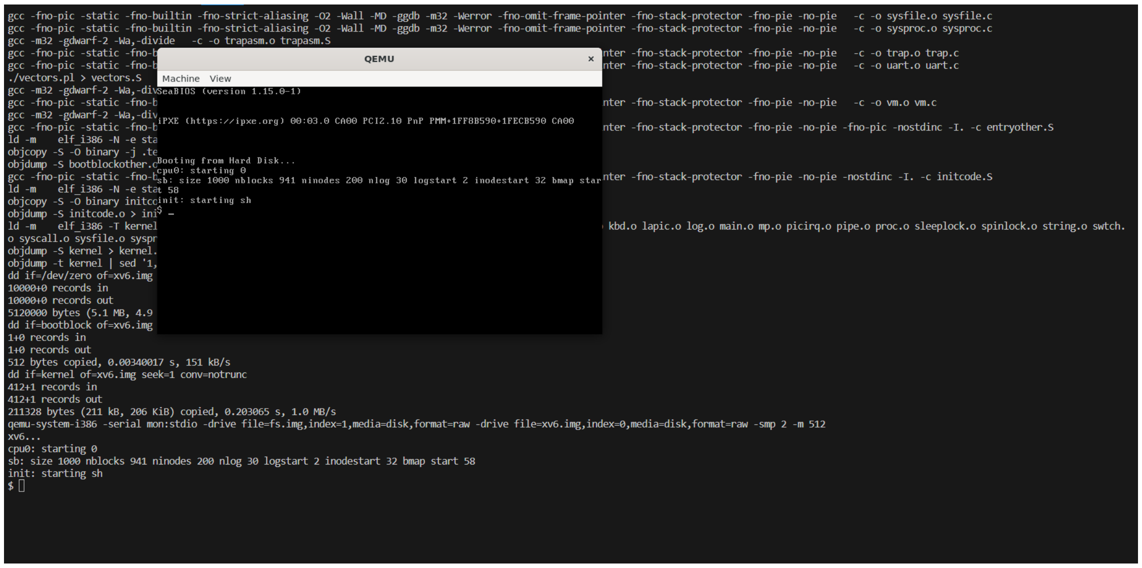
```
cd xv6-public
make
```

III. **Install Required Tools:** Ensure GCC, QEMU, and other necessary tools are installed.

IV. **Compile xv6:**

```
make qemu
```

V. **Run xv6:** This will launch xv6 in the QEMU emulator.



```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B590+1FECB590 CA00



Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ _
```

```
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -o sysfile.o sysfile.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie   -c -o sysproc.o sysproc.c
gcc -m32 -gdwarf-2 -Wa,-divide   -c -o trapasm.o trapasm.S
gcc -fno-pic -static -fno-b                                                    nter -fno-stack-protector -fno-pie -no-pie   -c -o trap.o trap.c
gcc -fno-pic -static -fno-b                                                    nter -fno-stack-protector -fno-pie -no-pie   -c -o uart.o uart.c
./vectors.pl > vectors.S
gcc -m32 -gdwarf-2 -Wa,-div
gcc -fno-pic -static -fno-b                                                    nter -fno-stack-protector -fno-pie -no-pie   -c -o vm.o vm.c
gcc -m32 -gdwarf-2 -Wa,-div
gcc -fno-pic -static -fno-b                                                    nter -fno-stack-protector -fno-pie -no-pie -fno-pic -nostdinc -I. -c entryother.S
ld -m    elf_i386 -N -e sta
objcopy -S -O binary -j .te
objdump -S bootblockother.o
gcc -fno-pic -static -fno-b                                                    nter -fno-stack-protector -fno-pie -no-pie -nostdinc -I. -c initcode.S
ld -m    elf_i386 -N -e sta
objcopy -S -O binary initco
objdump -S initcode.o > ini
ld -m    elf_i386 -T kernel                                                     kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.o string.o swtch.
o syscall.o sysfile.o syspr
objdump -S kernel > kernel.
objdump -t kernel | sed '1,
dd if=/dev/zero of=xv6.img
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9
dd if=bootblock of=xv6.img
1+0 records in
1+0 records out
512 bytes copied, 0.00340017 s, 151 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
412+1 records in
412+1 records out
211328 bytes (211 kB, 206 KiB) copied, 0.203065 s, 1.0 MB/s
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ 
```

2 System Call

To implement our method in xv6, the following files must be modified:

- **'sysproc.c'**: This file is where the actual implementation of our method should be added.

- **‘syscall.c‘**: In this file, a pointer to our system call function needs to be included. This file essentially serves as a collection of system call functions.
- **‘syscall.h‘**: Allocate a unique identifier for our system call in this header file. This identifier will be used to distinguish our system call from others.
- **‘user.h‘**: Ensure visibility of the system call to user programs by adding an entry in this file. This step is crucial for user-level code to utilize the newly added system call.
- **‘usys.S‘**: Add an entry to this assembly file, following conventions, so that the assembly stub can correctly move the system call number into register `eax` and make the system call.

Implement the System Call: Create a new file for your system call implementation and must include following header files

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

These changes collectively enable the integration of our custom system call into the xv6 operating system. Ensure that each file is modified according to the specified instructions to maintain consistency and functionality.

2.1 System Call ’toggle()’:

In `trap.c`, I have defined an integer variable `trace` and an integer array `system_call_count`. The variable `trace` is initialized to 0, and `system_call_count` is an array of 30 integers initialized to zeros.

```c
int trace = 0;
int system_call_count[30] = {0};
```

The variable `trace` undergoes toggling between 0 and 1 whenever the `toggle` function is called. It is utilized to track the tracing mode, distinguishing between `TRACE_ON` (1) and `TRACE_OFF` (0). If the tracing mode is `TRACE_ON`, the counts of all system calls are updated in the `system_call_count` array upon entering the `trap` function in `trap.c`.

```c
if (trace == 1) {
    int num = tf->eax;
    system_call_count[num] += 1;
}
```

2.2 System call 'print_count()':

In sysproc.c, the system calls sys_print_count and sys_toggle are defined, and the variables trace and system_call_count are declared as extern. The sys_print_count function prints the count of system calls in ascending alphabetical order using the system_call_count array. Counts are printed only if they are greater than 0. During the transition from 1 to 0 (TRACE_OFF), the system_call_count array is set to contain 0s to prevent count printing.

The sys_print_count function utilizes the alphabetical order of system call names to access the system_call_count array and prints both the system call name and the corresponding count accordingly.

```
int sys_print_count(void) {
  if (system_call_count[call_number] > 0) {
    cprintf("sys_call_name %d\n", system_call_count[call_number
       ]);
  }
   more system calls...
}
}
```

2.3 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.

```
int sys_add(void)
{
int a,b;
if(argint(0,&a)<0)
{
return -1;
}
if(argint(1,&b)<0)
{
return -1;
}
return a+b;
}
```

2.4 System call ps():

I have defined the sys_ps system call in sysproc.c. The sys_ps system call calls the list_process function defined in proc.c. The list_process function iterates over the ptable struct in proc.c and prints the pid and name of the process entry in ptable if the state is UNUSED.

```c
int sys_ps(void)
{
list_process();
return 0;
}
void list_process(void)
{
struct proc *p;
for(p=ptable.proc;p<&ptable.proc[NPROC];p++){
if(p->state!=UNUSED){
cprintf("pid:%d name:%s\n",p->pid,p->name);
}
}
return;
}
```

3 Inter Process Communication

3.1 Unicast Communication

First we have created a buffer structure as follows:

```c
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 each process, as the maximum number of processes in xv6 is 64). Each buffer has its own lock, and `"bufferstatus"` indicates whether the buffer is empty or not, signifying whether there is some data in the buffer that is yet to be read. To achieve this functionality, two system calls were implemented as follows:

Next, we define the send() function, which takes 2 integer and one char * argument, and stores the message in the buffer. Here the lock is used to ensure no other send/recv call can access the buffers during this time

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

Now, we implement the recv() call, which takes one char * argument and writes tha message to this location. Locks are implimented in a similar fashion. This is done using a blocking system call, so the receiving process will keep waiting for the message until it recieves it. This is done by continuously searching the buffer until a message appears for the calling process.

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

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

3.2 Multicast Communication

Here, a new system call, `sys_send_multi()`, is defined. It takes several parameters, including the sender's process ID, an array of receiver process IDs, the message as a `char*`, and the number of receivers. The implementation utilizes send() and recv() function instead of signal handler mechanism.The system call sys$_{s}end_{m}ulti()wasimplementedasfollows :$

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

In this, I proceed to send the message to each of the receiving processes using the `sys_send()` system call. This action places the message into the buffer of each receiving process, allowing it to be read. Simultaneously, it wakes up the receiving processes, effectively operating as a multicast communication model.

4 Distributed Algorithm

Here, the changes are made in assig1 8.c to implement the distributed algorithm to compute the sum of a 1000-element array. This program has two parts, one to compute the sum and another to compute the variance

First we initialised the number of processes (noOfProcesses), the parent process ID (pidParent), and determines the number of elements each process will handle (elmForEachProc). Memory is allocated for variables such as partialSumP and pmsg.

4.1 Calculating Sum

For calculating sum, the program enters a loop to fork 8 child processes, each responsible for calculating the partial sum of assigned array elements. Communication between child and parent processes occurs through the send system call, with the parent waiting for all child processes to complete before aggregating and printing the total sum.

```c
if (type == 0) {
        for (int i = 0; i < noOfProcesses; i++) {
            int cid = fork();
            if (cid == 0) {
                int curStart = i * elmForEachProc;
                int curEnd = (i + 1) * elmForEachProc;
                procData[i].partialSum = 0;
                for (int j = curStart; j < curEnd; j++) {
                    procData[i].partialSum += (int)arr[j];
                }

                procData[i].childId = getpid();
                send(getpid(), pidParent, (void *)&procData[i])
                   ;
                exit();
            } else {
                wait();
                struct sumData partialSumP;
                recv((void *)&partialSumP);
                tot_sum += partialSumP.partialSum;
            }
        }
    }
```

4.2 Calculating Variance

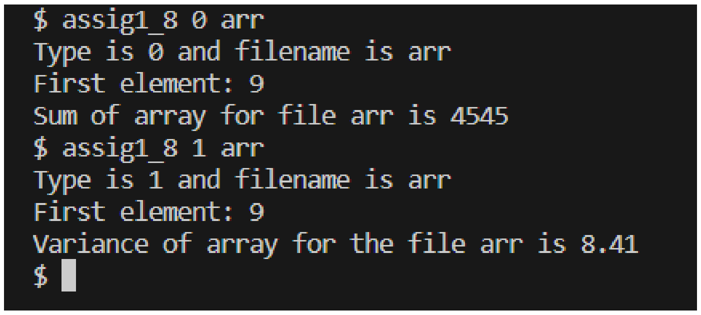
For calculating Variance, the initialization phase is repeated, and an array (indexArr) is created to determine the index range for each process. The program maintains an array (children) to keep track of child process IDs. In this model, child processes first calculate their partial sums, followed by two communication phases.

In the first phase, partial sums are sent to the parent process, which calculates the mean and multicasts it to all child processes. In the second phase, child processes unblock and compute the sum of squares of differences about the mean. The results are communicated back to the parent process, which then computes and prints the variance.

```
       else {
        struct sumData procData[noOfProcesses];

        int indexArr[noOfProcesses + 1];
        indexArr[0] = 0;
        indexArr[noOfProcesses] = size;
        int children[noOfProcesses];

        for (int i = 1; i < noOfProcesses; i++)
            indexArr[i] = indexArr[i - 1] + (size /
               noOfProcesses);

        for (int i = 0; i < noOfProcesses; i++) {
            int cid = fork();
            if (cid == 0) {
                int curStart = indexArr[i];
                int curEnd = indexArr[i + 1];
                procData[i].partialSum = 0;
                for (int j = curStart; j < curEnd; j++) {
                    procData[i].partialSum += (short)arr[j];
```

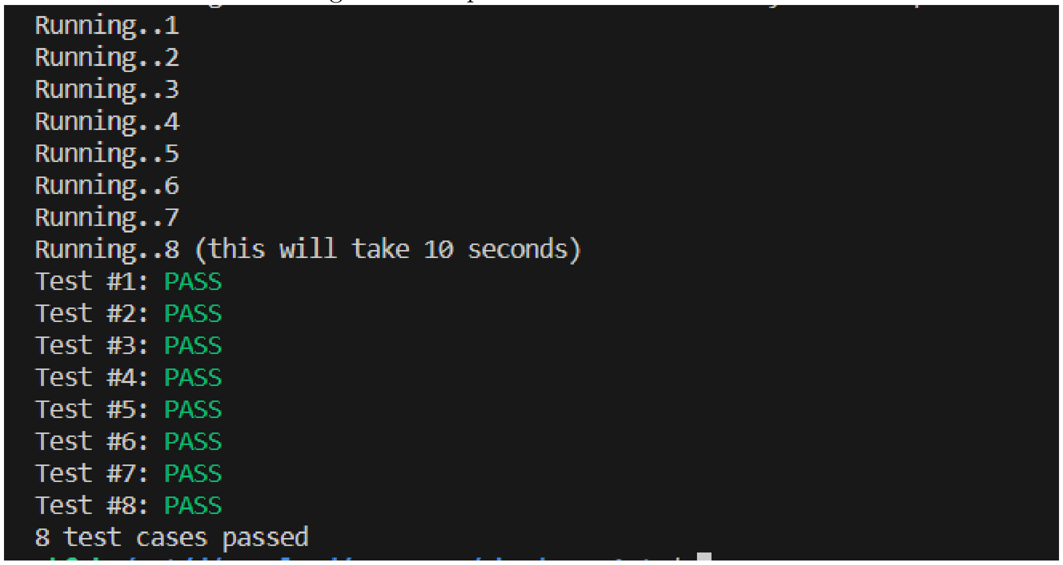
```c
                }

                procData[i].childId = getpid();
                send(getpid(), pidParent, (void *)&procData[i]);
                sleep(100 * (i + 1));

                struct MeanData mean;
                recv((void *)&mean);

                float var = 0;
                for (int k = curStart; k < curEnd; k++) {
                    var += (arr[k] - mean.mean) * (arr[k] - mean.mean);
                }

                sleep(100 * (i + 1));

                struct VarData ret_var;
                ret_var.var = var;
                ret_var.childId = getpid();
                send(getpid(), pidParent, (void *)&ret_var);
                exit();
            } else {
                children[i] = cid;
            }
        }

        sleep(10);

        for (int i = 0; i < noOfProcesses; i++) {
            struct sumData partialSumP1;
            recv((void *)&partialSumP1);
            tot_sum += partialSumP1.partialSum;
        }

        float p_mean = (float)tot_sum / size;
        struct MeanData multi_msg;
        multi_msg.mean = p_mean;
        multi_msg.childId = children[0];

        send_multi(getpid(), children, (void *)&multi_msg);

        struct VarData get_var;
        int v_v = 0;

        for (int i = 0; i < noOfProcesses; i++) {
            wait();
            recv((void *)&get_var);
            v_v += get_var.var;
        }

        variance = v_v / (float)size;
    }
```

5 Outputs and Extra Details



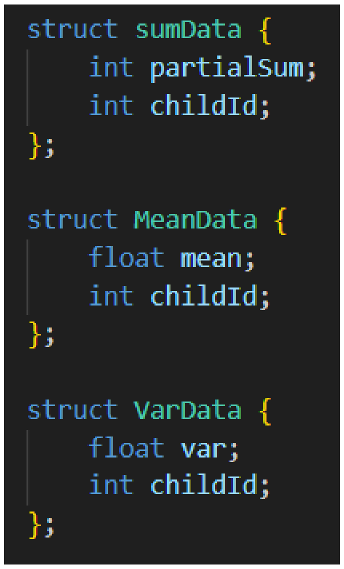
```
$ assig1_8 0 arr
Type is 0 and filename is arr
First element: 9
Sum of array for file arr is 4545
$ assig1_8 1 arr
Type is 1 and filename is arr
First element: 9
Variance of array for the file arr is 8.41
$
```

Figure 1: Output of Sum and Variance



```
Running..1
Running..2
Running..3
Running..4
Running..5
Running..6
Running..7
Running..8 (this will take 10 seconds)
Test #1: PASS
Test #2: PASS
Test #3: PASS
Test #4: PASS
Test #5: PASS
Test #6: PASS
Test #7: PASS
Test #8: PASS
8 test cases passed
```

Figure 2: Output of Checkscript



```c
struct sumData {
    int partialSum;
    int childId;
};

struct MeanData {
    float mean;
    int childId;
};

struct VarData {
    float var;
    int childId;
};
```

Figure 3: Data Structure used in `assig1_8`

The data structures `struct sumData`, `struct MeanData`, and `struct VarData` play crucial roles in facilitating effective communication between the parent process (coordinator) and its child processes. These structures are utilized for the organized exchange of partial sum values, mean calculations, and partial variances during the distributed computation of the array sum and variance.

- The `struct sumData` encapsulates information regarding the partial sum computed by each child process.
- The `struct MeanData` structure is employed for multicast communication of the mean value calculated by the coordinator to all child processes.
- The `struct VarData` structure is designed to convey partial variance values from child processes back to the coordinator.