CS344 – Operating Systems Lab Assignment-2

200123030 – Kura Priyanka

200123054 – Siddam Shetty Sahithi Shresta

200123070 – Vemulapati Sai Lakshmi Swarupa

Part A:

1)To implement getNumProc() and getMaxPID() :

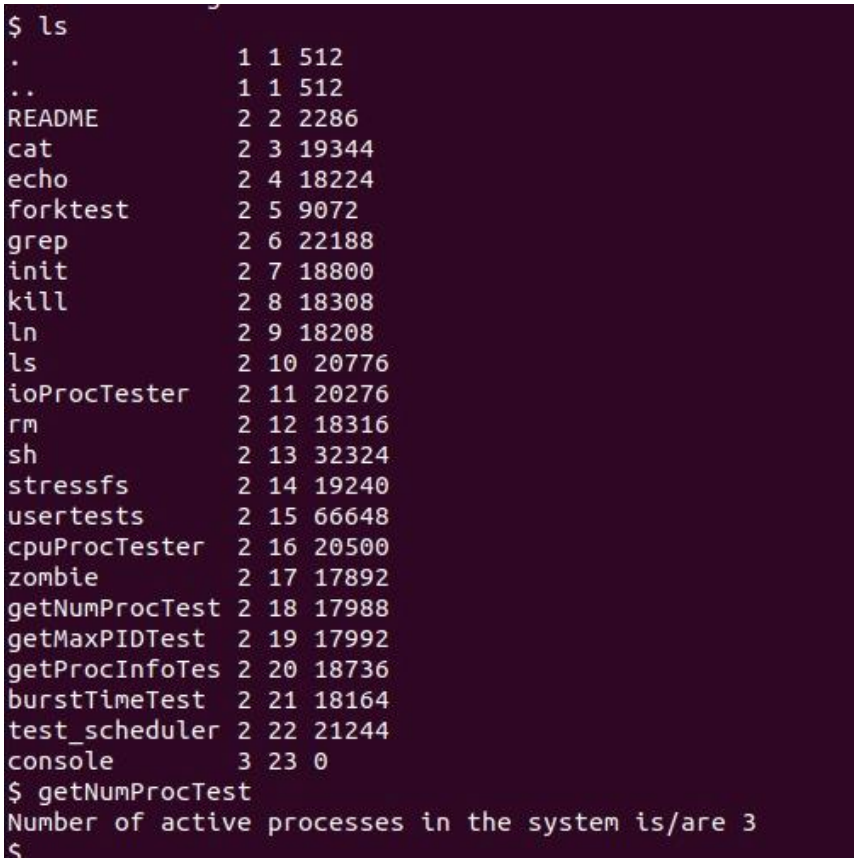
- Made system calls for these functions.
- In proc.c, 2 assistant functions named getNumProcAssist and getMaxPIDAssist are implemented to help us achieve the implementations.

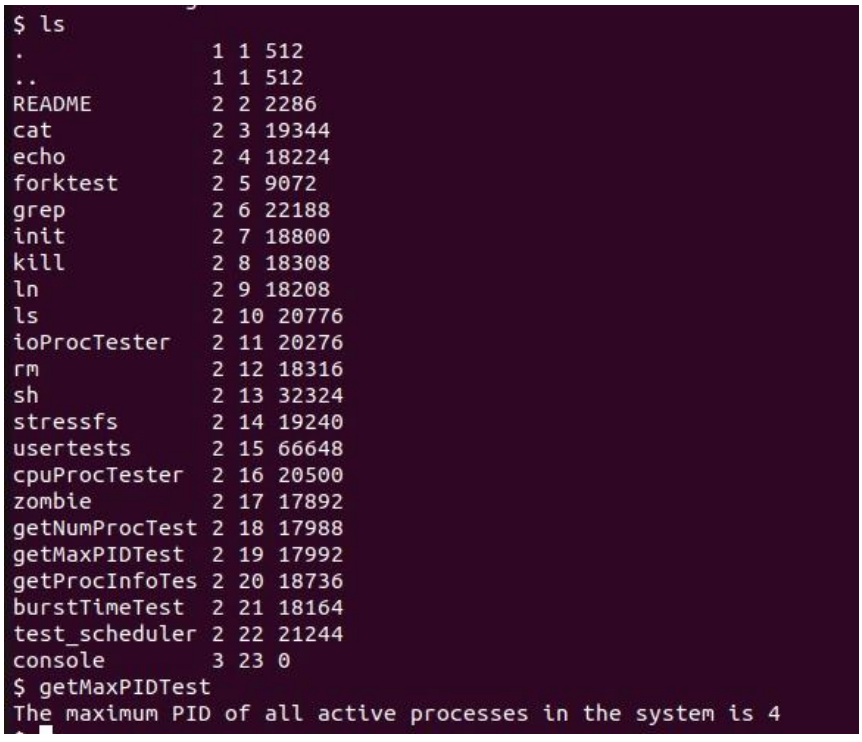
```
int getNumProcAssist(void){

        int ans=0;
        struct proc *p;

        acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != UNUSED)
        ans++;
    }
    release(&ptable.lock);

    return ans;
}
```

```
int getMaxPIDAssist(void){

        int max=0;
        struct proc *p;

        acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != UNUSED){
        if(p->pid > max)
                max=p->pid;
      }
    }
    release(&ptable.lock);

    return max;
}
```

The functions acquires the lock to ptable first and then loops through all processes and achieves the specified task.

Output of the functions-





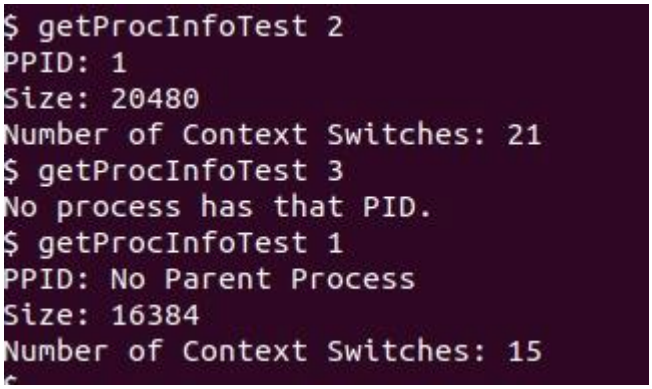
- getNumProc : Process ID 1 and 2 are allocated to system processes, so first we typed ls which displays all available user programs, here ls is assigned process ID 3. After completion of ls process, getNumProcTest is run, which is allotted the next available Process ID, which is 4. At this point of time, only 3 processes are running namely 2 system processes and getNumProcTest, so output of 3 is printed as observed.

- getMaxPID: Same as above and here Process ID 4 is assigned to getMaxPIDTest, so the maximum of the process ID's allotted is 4.

2) The first few steps to implement the system call getProcInfo is identical to those of previously made system calls. But here we also have to pass some arguments to the system call. We solve the problem of passing parameters to syscall using argptr which is a predefined system call which serves our purpose. To store the number of context switches for every process,we will modify struct proc to include additional member named nocs which stores the number of context switches. We will set nocs for every process by initializing it to 0 in allocproc() as before running a process we will first allocate place to it in ptable using allocproc. Next, we will increase nocs of a process by 1 in scheduler (), as whenever this process is scheduled, number of context switches will update accordingly. We create a dummy process defaultParent and set it as parent of every process using p->parent=&defaultParent in allocproc(). fork() replaces it with original parent after process is allocated. We set the PID of defaultParent to -2 in scheduler(). Using defaultParent, we instantly know if the process has a parent or not and if it has, we get its PID using p->parent->pid. The implementations are given below:

```
struct processInfo getProcInfoAssist(int pid){

        struct proc *p;
        struct processInfo temp = {-1,0,0};

        acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != UNUSED){
        // printf(1, "%d\n", p->pid);
        if(p->pid == pid) {
                temp.ppid = p->parent->pid;
                temp.psize = p->sz;
                temp.numberContextSwitches = p->nocs;
                release(&ptable.lock);
                return temp;
        }
      }
    }
    release(&ptable.lock);

    return temp;

}
int
sys_getProcInfo(void){
        int pid;

        struct processInfo *info;
        argptr(0,(void *)&pid, sizeof(pid));
        argptr(1,(void *)&info, sizeof(info));

        struct processInfo temporaryInfo = getProcInfoAssist(pid);

        if(temporaryInfo.ppid == -1)return -1;

        info->ppid = temporaryInfo.ppid;
        info->psize = temporaryInfo.psize;
  info->numberContextSwitches = temporaryInfo.numberContextSwitches;
        return 0;

}
```

Output of the function:



getProcInfoTest takes the PID as an argument and then iterates through the whole ptable for that process, if it finds a process with the given PID, it checks for its parent's PID using p->parent->pid, if this value is -2 then it means that it doesn't has a parent else it prints the parent's PID and the size of the process, number of context switches that process had.

3) We added an attribute burst_time to proc structure. We have also implemented 2 system calls set_burst_time(this will set burst time of the current process to the given value) and get_burst_time(this will retrieve the burst time of the current process) and then we initialized the burst_time to 0 in allocproc().Now, the next problem we face is how to access the current process without any given info such as process ID etc. The solution is to use a predefined function in xv6 namely myproc() which returns the pointer to proc structure of current process. Using this, we can easily access and change the burst times of current process.

The implementation for both the functions:

```
int
set_burst_timeAssist(int burst_time)
{
        struct proc *p = myproc();
        p->burst_time = burst_time;
  yield();

        return 0;
}
```

```
int
get_burst_timeAssist()
{
        struct proc *p = myproc();

        return p->burst_time;

}
```

To test this, we created a user program named burst_time_test, which will first set the burst time of the current process to 5 and then print the burst time of the process by retrieving it using the above-mentioned functions.

Output:

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

int
main(void){
        printf(1,"Dummy [rocess to test set_burst_time and get_burst_time system calls.\n");
        set_burst_time(5);
        printf(1, "Burst time is: %d\n", get_burst_time());
        exit();
}
```

```
$ burstTimeTest
Dummy process to test set_burst_time and get_burst_time system calls.
Burst time is: 5
```

Part B:

Keeping the burst times in mind, we have implemented a 'Shortest Job First' (SJF) scheduler to replace the previously used 'Round Robin' scheduler. In order to do this, we had to do two things: 𝕏 Remove the preemption of the current process (yield) on every OS clock tick so the current process completely finishes first. In the given round robin scheduler, the forced preemption of the current process with every clock tick is being handled in the trap.c file. We simply remove the following lines from trap.c to fix this issue:

```c
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
     yield();
```

Change the scheduler so that processes are executed in the increasing order of their burst times. In order to do this, we implemented a priority queue (min heap) using a simple array which sorts processes by burst time. Of course, this heap is locked. The queue at any particular time would contain all the 'RUNNABLE' processes on the system. When the scheduler needs to pick the next process, it simply chooses the process at the front of the priority queue by calling extract min. We had to make the following changes (All changes made in proc.c):

Declare priority queue:

```c
struct {
  struct spinlock lock;
  int siz;
  struct proc* proc[NPROC+1];
} pqueue;
```

Implement the following functions in the priority queue (All implementations done in proc.c):

insertIntoHeap (Inserts a given process into the priority queue):

```
void insertIntoHeap(struct proc *p){
        if(isFull())
                return;

        acquire(&pqueue.lock);

        pqueue.siz++;
        pqueue.proc[pqueue.siz]=p;
        int curr=pqueue.siz;
        while(curr>1 && ((pqueue.proc[curr]->burst_time)<(pqueue.proc[curr/2]->burst_time))){
                struct proc* temp=pqueue.proc[curr];
                pqueue.proc[curr]=pqueue.proc[curr/2];
                pqueue.proc[curr/2]=temp;
                curr/=2;
        }
        release(&pqueue.lock);


}
```

isEmpty (Checks if the priority queue is empty or not):

```
int isEmpty(){
        acquire(&pqueue.lock);
        if(pqueue.siz == 0){
                release(&pqueue.lock);
                return 1;
        }
        else{
                release(&pqueue.lock);
                return 0;
        }
}
```

isFull (Checks if the priority queue is full or not):

```
int isFull(){
        acquire(&pqueue.lock);
        if(pqueue.siz==NPROC){
                release(&pqueue.lock);
                return 1;
        }
        else{
                release(&pqueue.lock);
                return 0;
        }
}
```

extractMin (removes the process at the front of the queue and returns it):

```
struct proc * extractMin(){

        if(isEmpty())
                return 0;

        acquire(&pqueue.lock);
        struct proc* min=pqueue.proc[1];
        if(pqueue.siz==1)
        {
                pqueue.siz=0;
                release(&pqueue.lock);
        }
        else{
                pqueue.proc[1] = pqueue.proc[pqueue.siz];
                pqueue.siz--;
                release(&pqueue.lock);

                fix(1);
        }
        return min;
}
```

changeKey (Changes the burst time of a process with a given PID in the priority queue and updates the queue accordingly)

```
void changeKey(int pid, int x){

        acquire(&pqueue.lock);

        struct proc* p;
        int curr=-1;
        for(int i=1;i<=pqueue.siz;i++){
                if(pqueue.proc[i]->pid == pid){
                        p=pqueue.proc[i];
                        curr=i;
                        break;
                }
        }

        if(curr==-1){
                release(&pqueue.lock);
                return;
        }


        if(curr==pqueue.siz){
                pqueue.siz--;
                release(&pqueue.lock);
        }
        else{
                pqueue.proc[curr]=pqueue.proc[pqueue.siz];
                pqueue.siz--;
                release(&pqueue.lock);

                fix(curr);
        }

        p->burst_time=x;
        insertIntoHeap(p);

}
```

fix (performs Heapify on priority queue - basically converts the array into min heap assuming that the left subtree and the right subtree of the root are already min heaps.):

```
void fix(int curr){

        acquire(&pqueue.lock);
        while(curr*2<=pqueue.siz){
                if(curr*2+1<=pqueue.siz){
                        if((pqueue.proc[curr]->burst_time)<=(pqueue.proc[curr*2]->burst_time)&&(pqueue.proc[curr]->burst_time)<=(pqueue.proc[curr*2+1]->burst_time))
                                break;
                        else{
                                if((pqueue.proc[curr*2]->burst_time)<=(pqueue.proc[curr*2+1]->burst_time)){
                                        struct proc* temp=pqueue.proc[curr*2];
                                        pqueue.proc[curr*2]=pqueue.proc[curr];
                                        pqueue.proc[curr]=temp;
                                        curr*=2;
                                } else {
                                        struct proc* temp=pqueue.proc[curr*2+1];
                                        pqueue.proc[curr*2+1]=pqueue.proc[curr];
                                        pqueue.proc[curr]=temp;
                                        curr*=2;
                                        curr++;
                                }
                        }
                } else {
                        if((pqueue.proc[curr]->burst_time)<=(pqueue.proc[curr*2]->burst_time))
                                break;
                        else{
                                struct proc* temp=pqueue.proc[curr*2];
                                pqueue.proc[curr*2]=pqueue.proc[curr];
                                pqueue.proc[curr]=temp;
                                curr*=2;
                        }
                }
        }
        release(&pqueue.lock);

}
```

Change the scheduler so that it uses the priority queue to schedule the next process (**Note that the priority queue lacks acquired and required and released in the priority queue functions**):

```
void
scheduler(void)
{

        defaultParent.pid = -2;
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    acquire(&ptable.lock);

    //NEW SJF SCHEDULER

        if((p = extractMin()) == 0){release(&ptable.lock);continue;}

        if(p->state!=RUNNABLE)
                {release(&ptable.lock);continue;}

        c->proc = p;
        switchuvm(p);

        p->state = RUNNING;
        (p->nocs)++;

        swtch(&(c->scheduler), p->context);

        switchkvm();

        c->proc = 0;
        release(&ptable.lock);
```

Insert processes into the priority queue as and when their state becomes RUNNABLE. This happens in five functions - **yield, kill, fork, userinit and wakeup1**. The code from the **fork** function is given below. The rest of the instances are identical. The variable check is created to check if the process was already in the RUNNABLE state in which case it is already in the priority queue and shouldn't be inserted again:

Insert a **yield** call into **set_brust_time**. This is because when burst time of a process is set, its scheduling needs to be done on the basis of the new burst time. **Yield** switches the state of the current process to **RUNNABLE**, inserts it into the priority queue and switches the context to the scheduler.

```
int
set_burst_timeAssist(int burst_time)
{
        struct proc *p = myproc();
        p->burst_time = burst_time;
  yield();

        return 0;
}
```

Run Complexity: The runtime complexity of the scheduler is O(logn) because extractMin has a O(logn) time complexity and that is the dominating part of the scheduling process. The rest of the statements run in O(1) time. (Refer to the scheduler function shown in an above picture).

Corner case handling and safety:

- If the queue is empty, extractMin returns 0 after which the scheduler doesn't schedule any process. (See scheduler function)
- If the priority queue is full, insertIntoHeap rejects the new process and simply returns so no new process is inserted into the queue by removing an older process.
- When inserting a process into the priority queue, it is always checked whether the element was already in the priority queue or not. This is done by checking the state of the process prior to it becoming RUNNABLE. If it was already runnable, it was already in the queue.
- The priority queue functions are robust and don't lead to situations where a segmentation fault would occur
- Although the priority queue is expected to have only RUNNABLE processes, our scheduler checks if the process at the front is RUNNABLE or not. If not, the scheduler doesn't schedule this process. If the process somehow changed state, this measure protects the operating system
- When ZOMBIE child processes are freed, the priority queue is also checked for the processes and these processes are removed from there too using changeKey and extractMin.
- In order to maintain data consistency, a lock is always used when accessing pqueue. This lock is created specially for pqueue and is initialised in pinit.

```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&pqueue.lock, "pqueue");
}
```

Testing

Testing was done to make sure our new scheduler is robust and works correctly in every case. In order to do this, we **forked** multiple processes and gave them different burst times. Roughly half of the processes are **CPU bound processes** and the other half are **I/O bound processes**.

- CPU bound processes consist of loops that run for many iterations (10^8). An interesting fact we learned was that the loops are ignored by the compiler if the information computed in the loop isn't used later. Hence, we had to use the information computed in the loop later.
- I/O bound processes were simulated by calling **sleep(1)** 100 times. '**sleep'** changes the state of the current process to sleeping for a given number of clock ticks, which is something that happens when processes wait for user input. When one I/O bound process is put to sleep, the context is switched to another process that is decided by the scheduler.

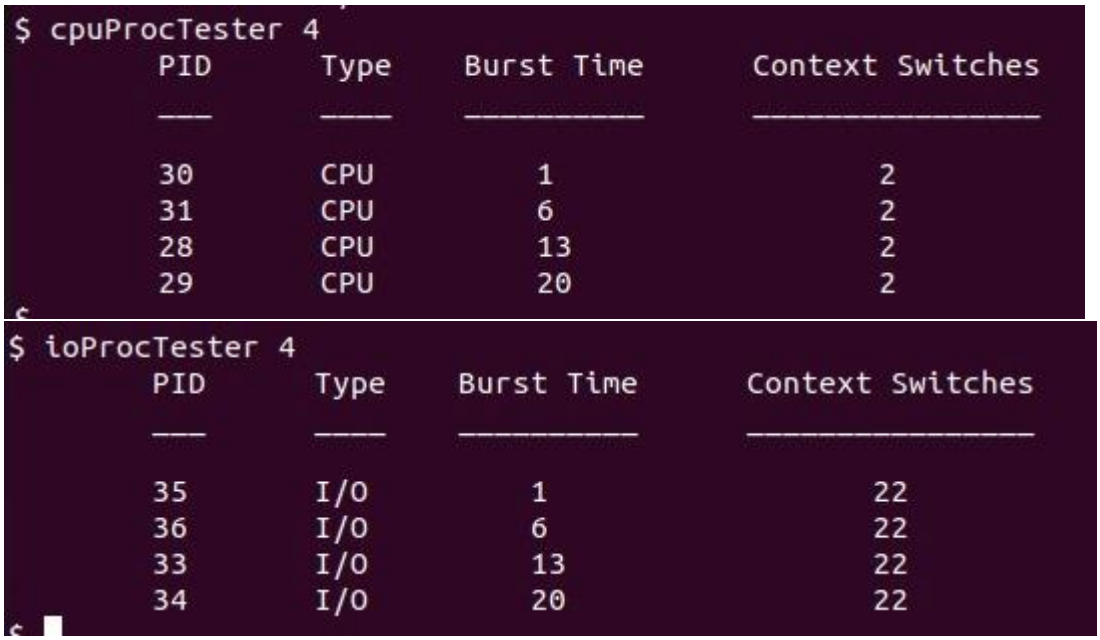
We first made a program called **test_scheduler** to check if the **SJF** scheduler is working according to burst times. It takes an argument equal to the number of forked processes and returns stats of each executed process:

```
$ test_scheduler 20
        PID      Type    Burst Time      Context Switches
        ___      ____    __________      ________________

        24       CPU         2                2
        14       CPU         3                2
        10       CPU         6                2
        16       CPU         6                2
        20       CPU         9                2
        22       CPU         14               2
        18       CPU         14               2
        26       CPU         17               2
        12       CPU         20               2
        8        CPU         20               2
        9        I/O         1                102
        11       I/O         2                102
        13       I/O         6                102
        23       I/O         9                102
        7        I/O         13               102
        25       I/O         15               102
        15       I/O         17               102
        21       I/O         18               102
        19       I/O         19               102
        17       I/O         20               102
$
```

As you can see, all CPU bound processes and I/O bound processes are sorted by their burst times and CPU bound processes finish first. The CPU bound processes finish first because I/O bound processes are blocked by the '**sleep**' system call. Since the processes are sorted by

burst time, we can say that the SJF scheduler is working perfectly. The context switches are also as expected. In the case of **CPU bound processes** first the process is switched in after which set_burst_time is called because of which the process is yielded and the next process is brought in. Finally, when the earlier process is chosen again by the scheduler, it finishes. In the case of **I/O bound processes**, they are also put to sleep 100 times. Hence, the processes have 100 additional context switches (they are brought back in 100 more times).

This is in contrast to the default **Round Robin Scheduler**. We created two special programs called **cpuProcTester** and **ioProcTester** to compare the **Round Robin Scheduler** with the SJF **Scheduler.cpuProcTester** runs CPU processes to simplify the comparison. **ioProcTester** only runs I/O bound processes: This is the outputs with the **Round Robin Scheduler**:



This is the output with the **SJF Scheduler (cpuProcTester):**

As you can see, since the Round Robin scheduler uses an FCFS queue, the order of execution is highly related to the PID of the process whereas in the SJF scheduler, the scheduling is happening by the burst times. Also, the number of context switches in the RR scheduler is very high. This is because of forced preemption on every clock tick.

Some Notes Regarding testing:

- Burst times are generated by the random number generator created in the file UaQdRP.c. We made his file a user library
- **IMPORTANT**: We removed wc and mkdir from the Makefile because we couldn't have more than 20 user programs in UPROGS. The OS wasn't compiling with a large number of user programs due to some virtual hard drive issue.
- **set_burst_time** yields the current process as mentioned in an above point. This is so that the new burst times are used in scheduling.