CS344 : Operating systems Lab

Assignment - 2

Group M10

Group Details

Member - 1

Name: Dhruvesh Bhure

Roll No: 200123018

Member - 2

Name: Soham Roy

Roll No: 200123055

Member - 3

Name: Dev Sandip Shah

Roll No: 200123074

Part A: Implementing following new system calls in xv6.

1. **getNumProc() and getMaxPid() -**

   To create the system calls **getNumProc()** and **getMaxPID()**, a procedure similar to the one used to create a system call in the previous assignment is used. We have created user programs named **testNumProc** and **testMaxPid** to use the above system calls.Two functions namely, **getNumProc** and **getMaxPID** are implemented in **proc.c** which help us in achieving the desired functionalities.

   - **getNumProc() -** returns the total number of active processes in the system (either in embryo, running, runnable, sleeping, or zombie states).

```
870 int getNumProc(void){
871   int ans = 0;
872   struct proc *p;
873   acquire(&ptable.lock);
874   for(p=ptable.proc;p<&ptable.proc[NPROC];++p){
875     if(p->state!=UNUSED)
876       ++ans;
877   }
878   release(&ptable.lock);
879   return ans;
880 }
```

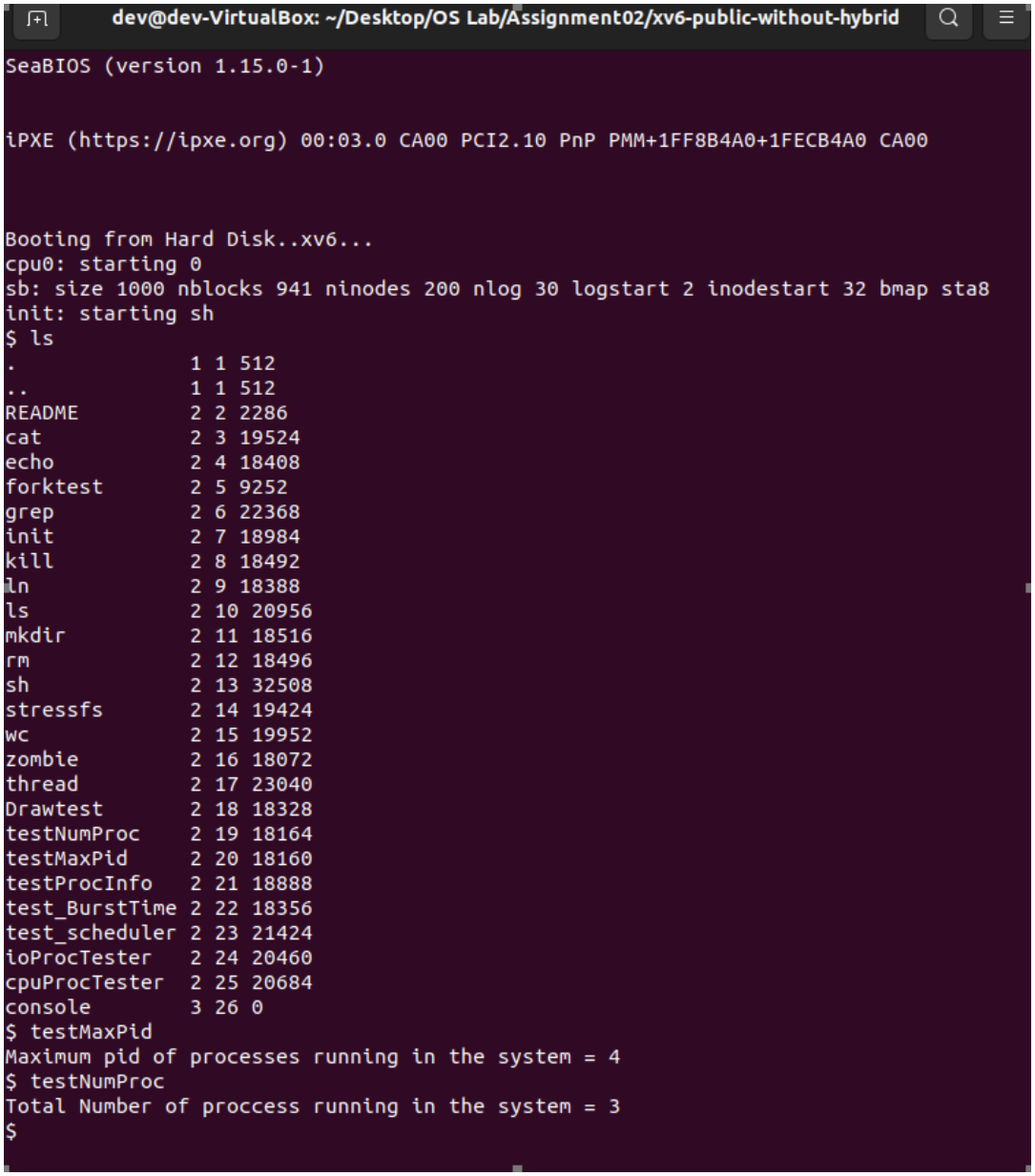
   - **getMaxPid() -** returns the maximum PID amongst the PIDs of all currently active processes in the system.

```
882 int getMaxPid(void){
883   int ans = -1;
884   struct proc *p;
885   acquire(&ptable.lock);
886   for(p = ptable.proc;p<&ptable.proc[NPROC];++p)
887     if(p->pid>ans)
888       ans = p->pid;
889   release(&ptable.lock);
890   return ans;
891 }
```

   As can be seen, the functions access **ptable** by acquiring its lock and then loop through it to carry out their respective tasks.

The output obtained on calling the user programs is attached below:

First, a **ls** command is run which shows the list of user programs available. This process is run with process ID 3. Because process ID 1 and 2 are allotted to system processes because of which the next available process ID is 3 which is allotted to ls. After completion of ls process, **testMaxPid** is run. The next available process ID is 4 which is allotted to **testMaxPid**. At this time, 3 processes (ls has already terminated) are currently running on the xv6 OS, the two system processes with PID 1 and PID 2 and **testMaxPid** with PID 4. Hence, the output is 4. Similarly, when **testNumProc** is run, 3 processes are running (2 system processes and 1 **testNumProc**). Hence the output is 3.



2. **getProcInfo(pid, &processInfo) -**
    - ➢ This system call takes as arguments an integer PID and a pointer to a structure processInfo. This structure is used for passing information between user and kernel mode.
    - ➢ This function call returns the information about the process which includes the **parent PID**, the number of times the process was **context switched** in by the scheduler, and the **process size in bytes**.

We followed the same common process to make a system call as done in previous labs. Till now none of the system calls we implemented had any parameters. This is the first system call in the OS lab in which we have to pass arguments. In xv6, there is a different process to pass the arguments. Here we pass the parameters to **syscall** using **argptr** which is a predefined system call which serves our purpose.

We added the last 3 lines to the **struct proc** in **proc.h** to maintain the information of each process where -

- ➢ numSwitches - maintains the number of context switches of each process.
- ➢ Burst_time - this variable maintains the burst time of each process.
- ➢ Runtime - this variable maintains the amount of time for which the process was in the running state.

```
37 // Per-process state
38 struct proc {
39   uint sz;                     // Size of process memory (bytes)
40   pde_t* pgdir;                // Page table
41   char *kstack;                // Bottom of kernel stack for this process
42   enum procstate state;        // Process state
43   int pid;                     // Process ID
44   struct proc *parent;         // Parent process
45   struct trapframe *tf;        // Trap frame for current syscall
46   struct context *context;     // swtch() here to run process
47   void *chan;                  // If non-zero, sleeping on chan
48   int killed;                  // If non-zero, have been killed
49   struct file *ofile[NOFILE];  // Open files
50   struct inode *cwd;           // Current directory
51   char name[16];
52   int isThread;
53   int numSwitches;               // Process name (debugging)
54   int burst_time;                //Added by us
55   int run_time;
56 };
```

Now we need to initialise **numSwitches** for every process. We do this by setting **p→numSwitches** to 0 in **allocproc()** function in **proc.c** ,since

before any process goes in running state, it is allocated (assigned a place in ptable) using **allocproc**. The next thing we do is add the statement

(p->numSwitches)++ to **scheduler().** This ensures that every time a process is scheduled, the number of context switches are updated accordingly.

We create a dummy process **defaultParent** and set it as parent of every process using **p→parent=&defaultParent** in **alloc proc(). fork()** replaces it with the original parent after the 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:

File : sysproc.c

```
144 int sys_getProcInfo(void){
145   int pid;
146   struct processInfo *info;
147   argptr(0,(void *) &pid,sizeof(pid));
148   argptr(1,(void *) &info,sizeof(info));
149   struct processInfo tempInfo = getProcInfoHelp(pid);
150   if(tempInfo.ppid==-1)
151     return -1;
152   info->ppid = tempInfo.ppid;
153   info->psize = tempInfo.psize;
154   info->numberContextSwitches = tempInfo.numberContextSwitches;
155   return 0;
156 }
```

File : proc.c

```
908 struct processInfo getProcInfoHelp(int pid){
909   struct proc *p;
910   struct processInfo temp = {-1,0,0};
911   acquire(&ptable.lock);
912   for(p = ptable.proc;p<&ptable.proc[NPROC];++p)
913   {
914     if(p->state!=UNUSED)
915     {
916       if(p->pid==pid){
917         temp.ppid = p->parent->pid;
918         temp.psize = p->sz;
919         temp.numberContextSwitches = p->numSwitches;
920         release(&ptable.lock);
921         return temp;
922       }
923     }
924   }
925       release(&ptable.lock);
926       return temp;
927 }
```

As can be seen above, we use **temp** (a dummy variable of type processInfo) to indicate that there exists no process with the given PID. The output obtained on running **testProcInfo** is as follows :

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 19524
echo           2 4 18408
forktest       2 5 9252
grep           2 6 22368
init           2 7 18984
kill           2 8 18492
ln             2 9 18388
ls             2 10 20956
mkdir          2 11 18516
rm             2 12 18496
sh             2 13 32508
stressfs       2 14 19424
wc             2 15 19952
zombie         2 16 18072
thread         2 17 23040
Drawtest       2 18 18328
testNumProc    2 19 18164
testMaxPid     2 20 18160
testProcInfo   2 21 18888
test_BurstTime 2 22 18356
test_scheduler 2 23 21424
ioProcTester   2 24 20460
cpuProcTester  2 25 20684
console        3 26 0
$ testProcInfo 3
No process has this PID.
$ testProcInfo 2
PPID:  1
Size: 20480
Number of Context Switches: 22
$ testProcInfo 1
PPID: No parent Process
Size: 16384
Number of Context Switches: 25
```

testProcInfo takes process ID as a command line parameter which is then passed to **getProcInfo** which then passes it to getProcInfoHelp. **getProcInfoHelp** iterates over ptable to find out the desired information and then returns the obtained values in the form of a **processInfo** variable. From this, we extract the desired information.

3. **set_burst_time() and get_burst_time()-**
   - ➢ **set_burst_time(n) -** This is a system call to set the burst time of the current process to n. In the question it is given that n will be a random number from 1 to 20.

```
893 int set_burst_timeAssist(int burst_time){
894         struct proc *p = myproc();
895         p->burst_time = burst_time;
896   yield();
897
898         return 0;
899 }
```

   - ➢ **get_burst_time() -** This system call returns the burst time we set for the current process.

```
901 int get_burst_timeAssist(){
902         struct proc *p = myproc();
903
904         return p->burst_time;
```

For this part, an additional attribute namely **burst_time** has to be added to **struct proc** in **proc.h**. We have already defined an attribute named **burst_time** but we need to set it to a default value before any changes are made. We do this by setting **p→burst_time** to 0 (we took the default value of burst time to be 0) in **allocproc()**.

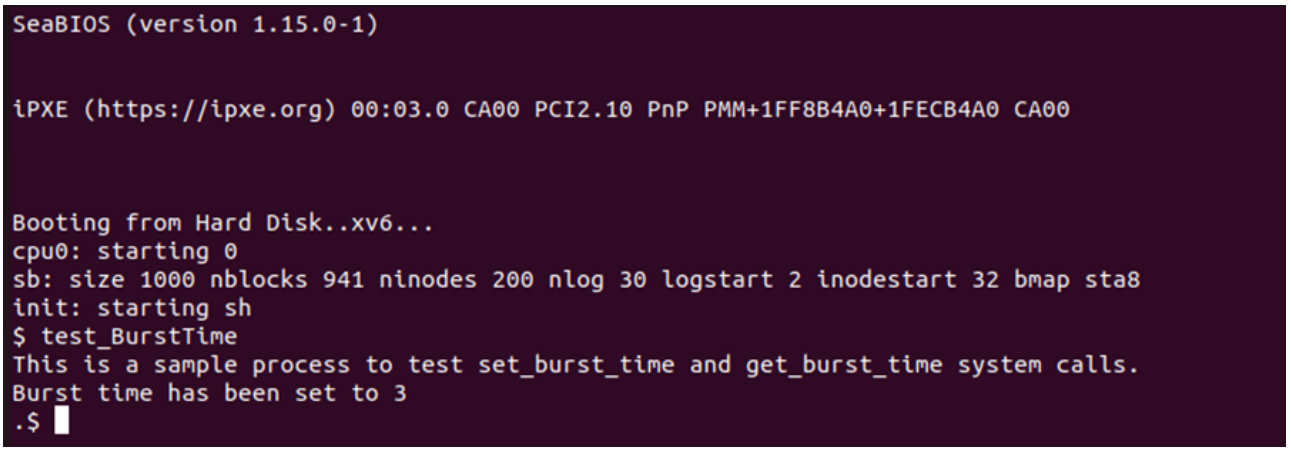
To access the current process there is a predefined function in xv6 called myproc() which returns the pointer to the struct proc of the current process.Now we change the burst time of the current process.

To test the above processes, we created a user program named **test_BurstTime**. It is as follows -

```
1 #include "types.h"
2 #include "stat.h"
3 #include "user.h"
4
5
6 int main(void)
7 {
8         printf(1,"This is a sample process to test set_burst_time and get_burst_time system calls.\n");
9         set_burst_time(3);
10        printf(1,"Burst time has been set to %d\n.", get_burst_time());
11        exit();
12 }
```

Here we have set the burst time of the current process to 3.

Output of the above program is as follows -



As we can see, the burst time we got in output is 3 which is the same as the one we set in the **test_BurstTime**. It changes from the default burst time 0 to 3 indicating that the above system calls are working correctly.

Part B: Modifying the default Round Robin scheduler in xv6.

The current scheduler in xv6 is an unweighted round robin scheduler. Now we will modify this scheduler to take into account user-defined process burst time and implement a shortest job first scheduler.

4. **Scheduler Implementation -**
   We are asked to implement the shortest job first scheduling algorithm. Keeping the burst times in mind, we have implemented a Shortest **Job First(SJF)** scheduler to replace the default **Round Robin Scheduler** in **xv6.**We have to do the following things to implement **SJF -**
   - ➢ In the default round robin scheduler, the forced preemption of the current process with every clock tick is being handled in the trap.c file. In order to implement SJF we removed the preemption of the current process on every OS clock tick so the current process completely finishes first.We simply comment the below code in trap.c to remove the preemption.

```
106  // Comment below 2 lines to implement SJF
107  // UnCommnent them to implement RR
108
109  if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER)
110    yield();
```

   - ➢ We changed 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 **extractMin().** We made the following changes to **proc.c**
     - **Declare priority queue** :

```
18 //implementation of Priority Queue
19 struct {
20         int sze;
21         struct spinlock lock;
22         struct proc* proc[NPROC+1];
23 } priorityQ;
```

We Implement the following functions in proc.c

➢ **InsertIntoPQ** - Inserts a given process into the priority queue

Implementation is as follows -

```
50 void insertIntoPQ(struct proc *p){
51         if(isFull())
52                 return;
53
54         acquire(&priorityQ.lock);
55         priorityQ.sze++;
56         priorityQ.proc[priorityQ.sze]=p;
57         int curr=priorityQ.sze;
58         while(!(curr<=1) && ((priorityQ.proc[curr]->burst_time)<(priorityQ.proc[curr/2]->burst_time))){
59                 struct proc* temp=priorityQ.proc[curr];
60                 priorityQ.proc[curr]=priorityQ.proc[curr/2];
61                 priorityQ.proc[curr/2]=temp;
62                 curr/=2;
63         }
64         release(&priorityQ.lock);
65
66
67 }
```

➢ **isEmpty** - Checks if priority queue is empty or not

Implementation is as follows -

```
26 int isEmpty(){
27         acquire(&priorityQ.lock);
28         if(priorityQ.sze == 0){
29                 release(&priorityQ.lock);
30                 return 1;
31         }
32         else{
33                 release(&priorityQ.lock);
34                 return 0;
35         }
36 }
```

➢ **isFull** - Checks if priority queue is full or not

Implementation is as follows -

```
38 int isFull(){
39         acquire(&priorityQ.lock);
40         if(priorityQ.sze==NPROC){
41                 release(&priorityQ.lock);
42                 return 1;
43         }
44         else{
45                 release(&priorityQ.lock);
46                 return 0;
47         }
48 }
```

➢ **extractMin** - removes process at the front of the queue and returns it

Implementation is as follows -

```
106 struct proc * extractMin(){
107
108         if(isEmpty())
109                 return 0;
110
111         acquire(&priorityQ.lock);
112         struct proc* minimum=priorityQ.proc[1];
113         if(priorityQ.sze==1)
114         {
115                 priorityQ.sze=0;
116                 release(&priorityQ.lock);
117         }
118         else{
119                 priorityQ.proc[1] = priorityQ.proc[priorityQ.sze];
120                 priorityQ.sze--;
121                 release(&priorityQ.lock);
122
123                 heapify(1);
124         }
125         return minimum;
126 }
```

➢ **changeKey** - Changes the burst time of a process with a given PID in the priority queue and updates the queue accordingly

Implementation is as follows -

```
128 void changeKey(int pid, int newBT){
129
130         acquire(&priorityQ.lock);
131
132         struct proc* p;
133         int curr=-1;
134         for(int i=1;i<=priorityQ.sze;i++){
135                 if(priorityQ.proc[i]->pid == pid){
136                         p=priorityQ.proc[i];
137                         curr=i;
138                         break;
139                 }
140         }
141
142         if(curr==-1){
143                 release(&priorityQ.lock);
144                 return;
145         }
146
147
148         if(curr==priorityQ.sze){
149                 priorityQ.sze--;
150                 release(&priorityQ.lock);
151         }
152         else{
153                 priorityQ.proc[curr]=priorityQ.proc[priorityQ.sze];
154                 priorityQ.sze--;
155                 release(&priorityQ.lock);
156
157                 heapify(curr);
158         }
159
160         p->burst_time=newBT;
161         insertIntoPQ(p);
162
163 }
```

➢ **heapify** - basically converts the array into min heap assuming that left subtree and the right subtree of the root are already min heaps

Implementation is as follows -

```
70 void heapify(int curr){
71
72         acquire(&priorityQ.lock);
73         while(curr*2<=priorityQ.sze){
74                 if(curr*2+1<=priorityQ.sze){
75                         if((priorityQ.proc[curr]->burst_time)<=(priorityQ.proc[curr*2]-
   >burst_time)&&(priorityQ.proc[curr]->burst_time)<=(priorityQ.proc[curr*2+1]->burst_time))
76                                 break;
77                         else{
78                                 if((priorityQ.proc[curr*2]-
   >burst_time)<=(priorityQ.proc[curr*2+1]->burst_time)){
79                                         struct proc* temp=priorityQ.proc[curr*2];
80                                         priorityQ.proc[curr*2]=priorityQ.proc[curr];
81                                         priorityQ.proc[curr]=temp;
82                                         curr*=2;
83                                 } else {
84                                         struct proc* temp=priorityQ.proc[curr*2+1];
85                                         priorityQ.proc[curr*2+1]=priorityQ.proc[curr];
86                                         priorityQ.proc[curr]=temp;
87                                         curr*=2;
88                                         curr++;
89                                 }
90                         }
91                 } else {
92                         if((priorityQ.proc[curr]->burst_time)<=(priorityQ.proc[curr*2]-
   >burst_time))
93                                 break;
94                         else{
95                                 struct proc* temp=priorityQ.proc[curr*2];
96                                 priorityQ.proc[curr*2]=priorityQ.proc[curr];
97                                 priorityQ.proc[curr]=temp;
98                                 curr*=2;
99                         }
100                }
101        }
102        release(&priorityQ.lock);
103
104 }
```

➢ Change the scheduler so that it uses the priority queue to schedule the next process (**priority queue locks are acquired and released in the priority queue functions**):

Implementation is as follows -

```
488 void scheduler(void){
489   defaultParent.pid = -2;
490   struct proc *p;
491   struct cpu *c = mycpu();
492   c->proc = 0;
493   for(;;){
494     // Enable interrupts on this processor.
495     sti();
496     // Default RR scheduling
497     // Loop over process table looking for process to run.
498     /*acquire(&ptable.lock);
499     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
500       if(p->state != RUNNABLE)
501         continue;
502       // Switch to chosen process.  It is the process's job
503       // to release ptable.lock and then reacquire it
504       // before jumping back to us.
505       c->proc = p;
506       switchuvm(p);
507       p->state = RUNNING;
508       (p->numSwitches)++;
509       swtch(&(c->scheduler), p->context);
510       switchkvm();
511       // Process is done running for now.
512       // It should have changed its p->state before coming back.
513       c->proc = 0;
514     }
515     release(&ptable.lock);*/
516
517     // NEW Shortest Job Scheduler
518     acquire(&ptable.lock);
519         if((p = extractMin()) == 0){release(&ptable.lock);continue;}
520         if(p->state!=RUNNABLE)
521                 {release(&ptable.lock);continue;}
522         c->proc = p;
523         switchuvm(p);
524         p->state = RUNNING;
525         (p->numSwitches)++;
526         swtch(&(c->scheduler), p->context);
527         switchkvm();
528         c->proc = 0;
529     release(&ptable.lock);
530   }
531 }
```

➢ Insert processes into the priority queue as and when their state becomes **RUNNABLE** This happens in five functions – **yield, kill, fork, userinit, 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:

Implementation is as follows -

```
375   acquire(&ptable.lock);
376
377   short check = (np->state!=RUNNABLE);
378   np->state = RUNNABLE;
379
380   //Insert Process Into Queue
381   if(check)
382         insertIntoPQ(np);
383
384   release(&ptable.lock);
385
386   return pid;
387 }
```

➢ Insert a **yield** call into **set_burst_time**. This is because when the 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.

Implementation is as follows -

```
879 int set_burst_timeAssist(int burst_time){
880         struct proc *p = myproc();
881         p->burst_time = burst_time;
882   yield();
883
884         return 0;
885 }
```

Runtime Complexity analysis -

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.

Corner cases handling and safety -

➢ The priority queue functions are robust and don't lead to a situation where a segmentation fault would occur.

➢ 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.

- 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 changes state, this measure protects the operating system.
- In order to maintain data consistency, a lock is always used when accessing **priorityQ**.This lock is created especially for **priorityQ** and is initialised in **pinit** -

```
175 void
176 pinit(void)
177 {
178   initlock(&ptable.lock, "ptable");
179   initlock(&priorityQ.lock, "priorityQ");
180 }
```

- 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**.
- If the queue is empty, **extractMin** returns 0 after which the scheduler doesn't schedule any process.
- If the priority queue is full, **insertIntoPQ** rejects the new process and simply returns so no new process is inserted into the queue by removing an older process.

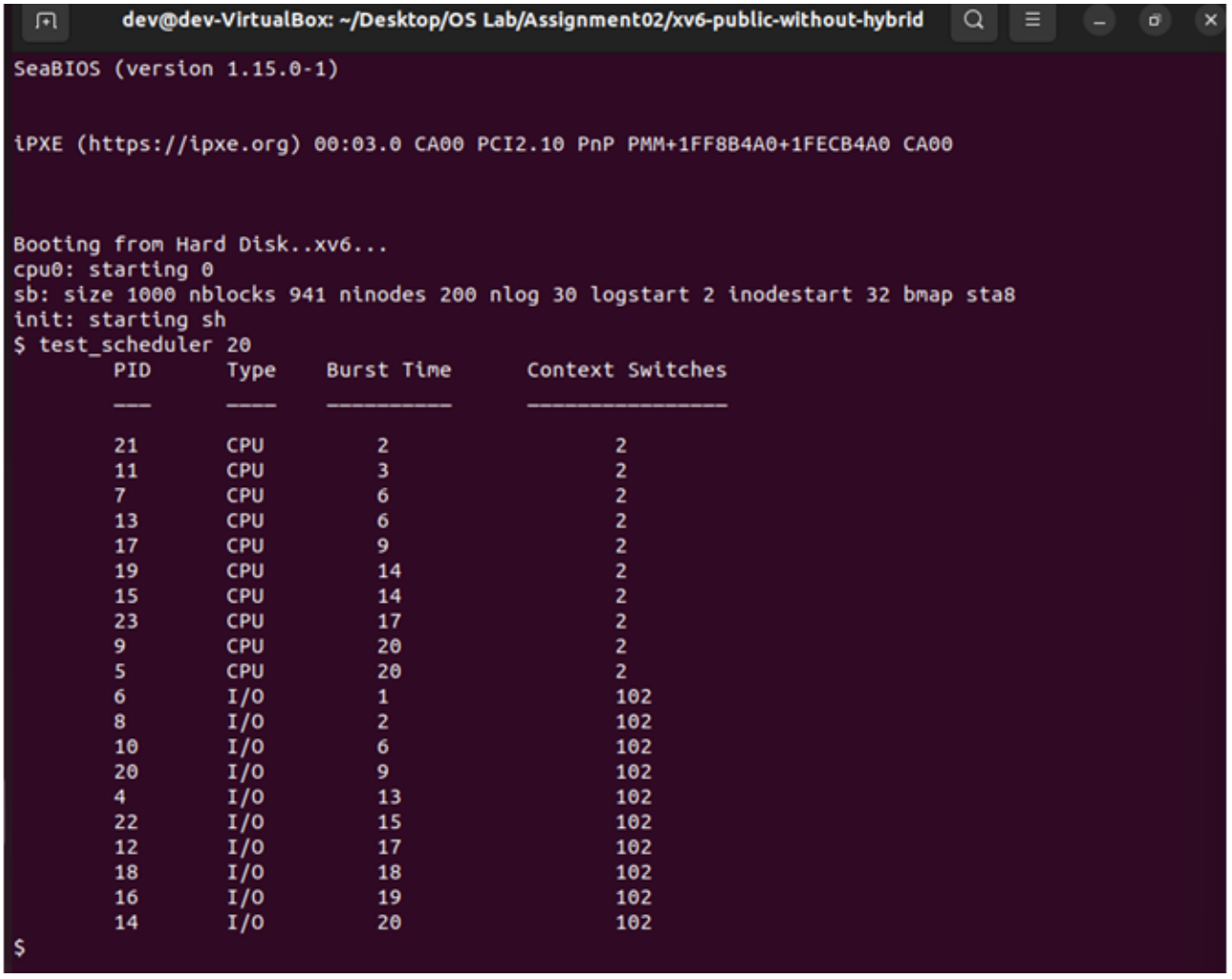
5. Test_scheduler Program -

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.



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).

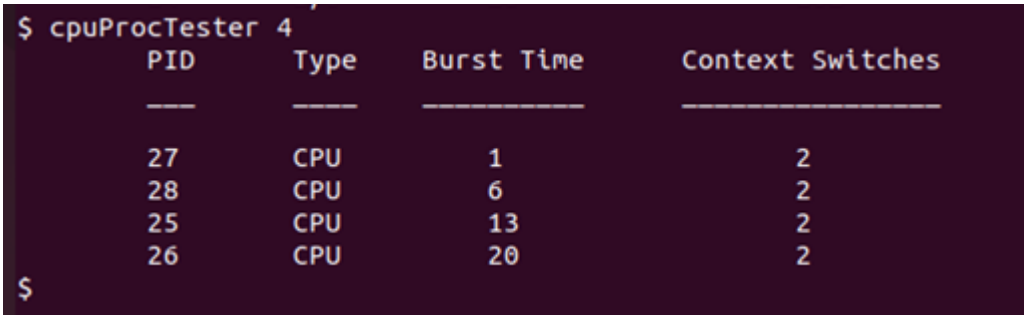
This is in contrast to the default **Round Robin Scheduler**.We created two special programs called **cpuProcTester** and **ioProcTester** to compare the **Round Robin Scheduler** with the **SJF Scheduler**. **cpuProcTester** runs CPU processes to simplify the comparison. **ioProcTester** only runs I/O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

Output with the **Round Robin (RR) Scheduler** is as follows **-**

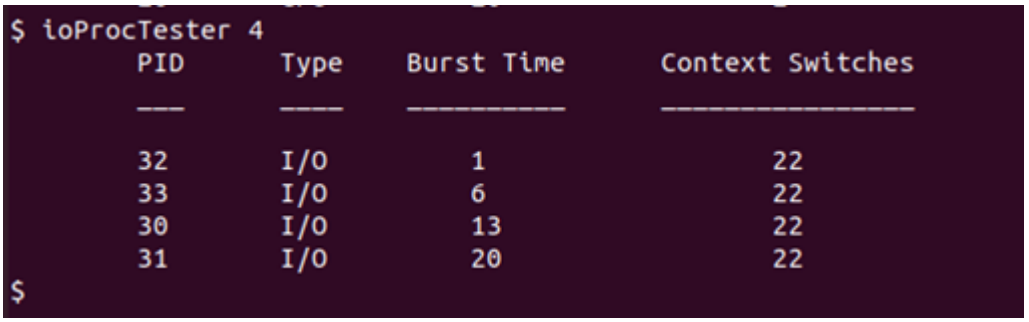
```
$ cpuProcTester 4
        PID      Type    Burst Time      Context Switches
        ---      ----    ----------      ----------------

        5        CPU        20                24
        4        CPU        13                25
        7        CPU        6                 25
        6        CPU        1                 27
```

```
$ ioProcTester 4
        PID      Type    Burst Time      Context Switches
        ---      ----    ----------      ----------------

        9        I/O        13                22
        10       I/O        20                22
        11       I/O        1                 22
        12       I/O        6                 22
$
```

Output with the **Shortest job first (SJF) Scheduler** is as follows **-**



```
$ cpuProcTester 4
        PID      Type    Burst Time      Context Switches
        ---      ----    ----------      ----------------

        27       CPU         1                2
        28       CPU         6                2
        25       CPU         13               2
        26       CPU         20               2
$
```



```
$ ioProcTester 4
        PID      Type    Burst Time      Context Switches
        ---      ----    ----------      ----------------

        32       I/O         1                22
        33       I/O         6                22
        30       I/O         13               22
        31       I/O         20               22
$
```

As you can see, since the **Round Robin (RR) 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 points to be noted -

- ➢ Burst times are generated by the random number generator created in the file **random.c.** We made this file a user library.
- ➢ **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.

6. **Hybrid Scheduling -**
   We implemented a scheduler that behaves as a **hybrid** between **Round Robin** and **SJF** schedulers. We did this by modifying the **trap.c, proc.c** and **defs.h** files**.**

   We first create the logic for setting up a time quantum. In order to do this, we declare an **extern int** variable in **defs.h** so it can be initialised in **proc.c**. This variable is called **quant.** It is initialised to 1000 by default as the burst times are between 1 and 1000. The assignment asks us to make the time quantum equal to the burst time of the process with the shortest burst time in the priority queue. However, the default burst time is zero. If the burst time of a process is zero, we do not know how long the process will run and hence we assign a default burst time of zero. Therefore, the **quant variable is modified in the set_burst_time function.** If the burst time to be set is less than the **quant** value, **quant's** value is set to burst time.

   Next, we create another priority queue called **priorityQ2** and the corresponding functions for this priority queue. Functions like **insertIntoPQ, heapify, extractMin** etc were created corresponding to **priorityQ.** The corresponding functions for **priorityQ2** are **insertIntoPQ2, extractMin2, heapify2** etc. The functions are the same as the original ones except they modify priorityQ2 instead of priority.

   We then modified the **clock tick interrupt handler** in the **trap.c** file. At every clock tick, we increment the running time (**added parameter in struct proc – run_time** which is initialised to zero when proc is created in **allocproc**) field of the current process (**myproc()**). By default, processes have burst time zero. If the burst time of a process isn't manually set, we don't want to kill the process as soon as its first clock tick is observed since that may seriously affect the functioning of the OS. So, before we check if the current running time is equal to the burst time of the process, we check if the burst time is zero. We only make the equivalence check between **run_time** and **burst_time** if the burst_time value is non zero. If the equivalence is true, we **exit()** the current process. Otherwise, we make the next check - we check if the running time of the current process is divisible by the time quantum, **quant.** If true, we preempt

the current process and insert into the other priority queue **priorityQ2.**

```
103   // Force process to give up CPU on clock tick.
104   // If interrupts were on while locks held, would need to check nlock.
105   if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER){
106      (myproc()->run_time)++;
107      if(myproc()->burst_time != 0){
108         if(myproc()->burst_time == myproc()->run_time)
109                 exit();
110      }
111      if((myproc()->run_time)%quant == 0)
112         new_yield();
113   }
```

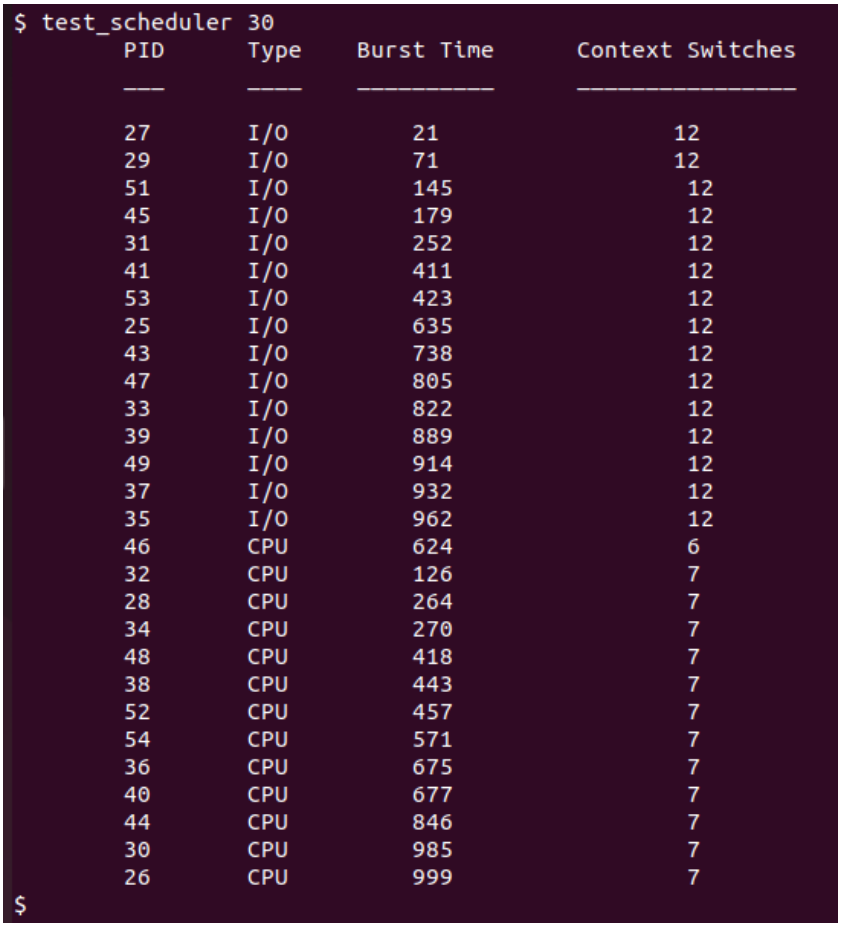
```
740 void new_yield(void){
741   acquire(&ptable.lock);
742
743   myproc()->state = RUNNABLE;
744
745   insertIntoPQ2(myproc());
746
747   sched();
748   release(&ptable.lock);
749
750 }
```

```
629 void scheduler(void){
630   defaultParent.pid = -2;
631   struct proc *p;
632   struct cpu *c = mycpu();
633   c->proc = 0;
634   for(;;){
635     // Enable interrupts on this processor.
636     sti();
637     // New Hybrid Scheduler
638     acquire(&ptable.lock);
639     if(isEmpty()){
640       if(isEmpty2()){ goto label;}
641       while(!isEmpty2()){
642         if((p = extractMin2()) == 0){release(&ptable.lock);break;}
643         insertIntoPQ(p);
644       }
645     }
646     label:
647         if((p = extractMin()) == 0){release(&ptable.lock);continue;}
648         if(p->state!=RUNNABLE) {release(&ptable.lock);continue;}
649         c->proc = p;
650         switchuvm(p);
651         p->state = RUNNING;
652         (p->numSwitches)++;
653         swtch(&(c->scheduler), p->context);
654         switchkvm();
655         c->proc = 0;
656     release(&ptable.lock);
657   }
658 }
```

The final part is the testing which is done using three user programs - **test_scheduler, cpuProcTester and ioProcTester.** In the code files corresponding to these programs, we did the following:

- In **test_scheduler.c,** half of the forked processes are I/O bound processes and the other half are CPU bound processes. All forked processes are assigned a random burst time between 1 and 1000. CPU bound processes consist of loops that run for 10^9 iterations and I/O bound processes consist of calling sleep(1) 10 times.
- In **cpuProcTester.c,** we just made all the forked processes CPU bound. Burst times are assigned randomly between 1 and 1000. A loop of 10^9 iterations is run (takes some time - approximately 200 ticks).
- In **ioProcTester.c,** every forked process is I/O bound. In order to simulate I/O, we are simply calling **sleep(1)** 10 times. Burst times are assigned at random with values between 1 and 1000.

Output obtained from the above tests are as follows -



```
$ test_scheduler 30
        PID     Type    Burst Time      Context Switches
        ___     ____    __________      ________________

        27      I/O         21              12
        29      I/O         71              12
        51      I/O         145              12
        45      I/O         179              12
        31      I/O         252              12
        41      I/O         411              12
        53      I/O         423              12
        25      I/O         635              12
        43      I/O         738              12
        47      I/O         805              12
        33      I/O         822              12
        39      I/O         889              12
        49      I/O         914              12
        37      I/O         932              12
        35      I/O         962              12
        46      CPU         624              6
        32      CPU         126              7
        28      CPU         264              7
        34      CPU         270              7
        48      CPU         418              7
        38      CPU         443              7
        52      CPU         457              7
        54      CPU         571              7
        36      CPU         675              7
        40      CPU         677              7
        44      CPU         846              7
        30      CPU         985              7
        26      CPU         999              7
$
```

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

        42       CPU         67                4
        29       CPU         71                4
        32       CPU         126                4
        31       CPU         252                4
        28       CPU         264                4
        34       CPU         270                4
        41       CPU         411                4
        38       CPU         443                4
        25       CPU         635                4
        36       CPU         675                4
        40       CPU         677                4
        43       CPU         738                4
        33       CPU         822                4
        44       CPU         846                4
        39       CPU         889                4
        37       CPU         932                4
        35       CPU         962                4
        30       CPU         985                4
        26       CPU         999                4
$
```

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

        48       I/O         21                22
        63       I/O         67                22
        50       I/O         71                22
        53       I/O         126                22
        52       I/O         252                22
        49       I/O         264                22
        55       I/O         270                22
        62       I/O         411                22
        59       I/O         443                22
        46       I/O         635                22
        57       I/O         675                22
        61       I/O         677                22
        64       I/O         738                22
        54       I/O         822                22
        65       I/O         846                22
        60       I/O         889                22
        58       I/O         932                22
        56       I/O         962                22
        51       I/O         985                22
        47       I/O         999                22
$
```

We got the results as expected.

Observations -

- ➢ Not all CPU bound processes were actually completed. This is because some of them had a burst time lower than their actual execution time and were killed before they printed anything. This proves that when the **run_time** value of the process becomes equal to the **burst_time** value of that process, the process is actually being exited.
- ➢ There was a considerably higher number of context switches with this new hybrid scheduling in the CPU bound processes. This is because the CPU processes are being preempted every quant clock ticks.
- ➢ The I/O bound processes weren't affected by the pre-emption and they behaved just like they did in SJF scheduling. This is because they are sleeping most of the time. Their actual execution time is very low. The likelihood of them experiencing quant clock ticks is very low since quant is expected to be a 2-3 digit number (burst times are chosen randomly between 1 and 1000 which affects quant). Hence, they didn't get forcefully pre-empted at regular intervals. They just went to sleep repeatedly. Hence, their number of context switches remained the same as they would be in SJF scheduling.
- ➢ When we are using a combination of CPU and I/O bound processes, some CPU bound processes are getting preempted more than others since I/O bound processes are returning from the SLEEPING state and forcing the currently running CPU bound processes to get pre-empted. This leads to out of order execution of some CPU bound processes.