CS344 Operating Systems Lab
Assignment 2

GROUP 20
Anjali Godara -180101008
Niharika Bhamer 180101048
Varhade Amey Anant 180101087
Tanmay Jain 180123050

Part A

1. getNumProc() and getMaxPid()

To add the system calls we start by adding the following lines to the file ***syscall.h***

```
#define sys_getNumProc 22
 #define sys_getMaxPid 23
```

Then we add a pointer to the ***syscall.c*** file This file contains an array of function pointers that uses the above-defined numbers (indexes) as pointers to system calls, which are defined in a different location.

```
[SYS_getNumProc] sys_getNumProc
 [SYS_getMaxPid] sys_getMaxPid
```

Next we need to add the function prototype in ***syscall.c***,

```
extern int sys_getNumProc(void);
extern int sys_getMaxPid(void);
```

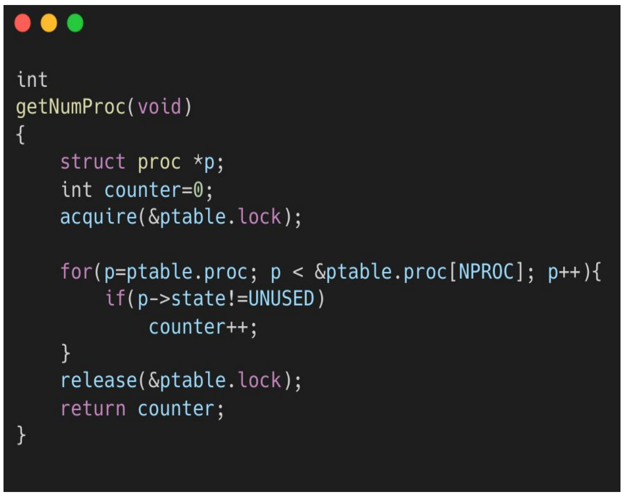
In order for a user program to call the system call, we added the following interface in the ***usys.S*** file

```
SYSCALL(getNumProc)
SYSCALL(getMaxPid)
```

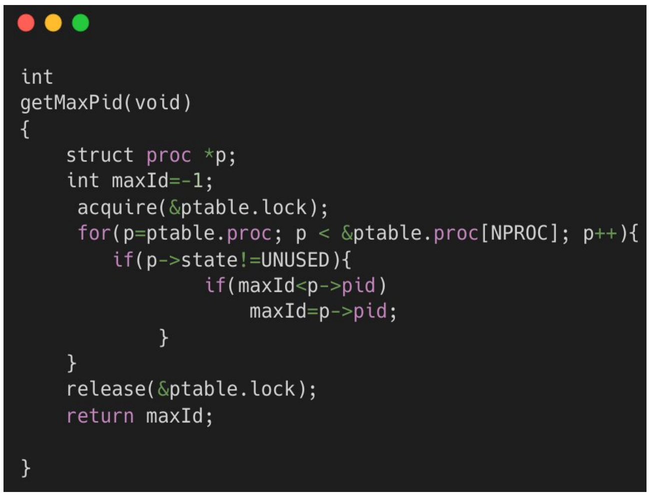
Then we need to add the function declaration in ***user.h*** file

```
int getNumProc(void);
int getMaxPid(void);
```

We need to add the function prototype in the file ***proc.c***



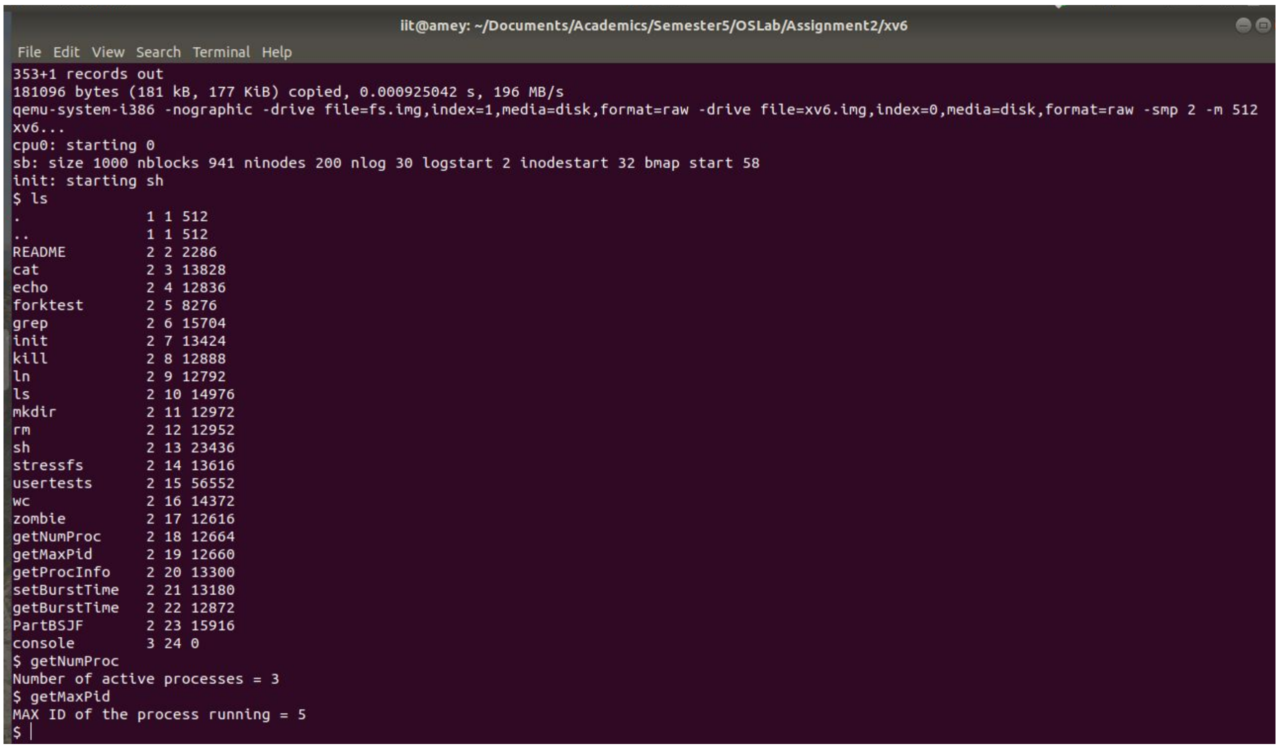
```
int
getNumProc(void)
{
    struct proc *p;
    int counter=0;
    acquire(&ptable.lock);

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



```
int
getMaxPid(void)
{
    struct proc *p;
    int maxId=-1;
     acquire(&ptable.lock);
     for(p=ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state!=UNUSED){
                if(maxId<p->pid)
                    maxId=p->pid;
            }
    }
    release(&ptable.lock);
    return maxId;

}
```

Changes to the Makefile are made at appropriate places.

The below image contains the implementation of both these system calls



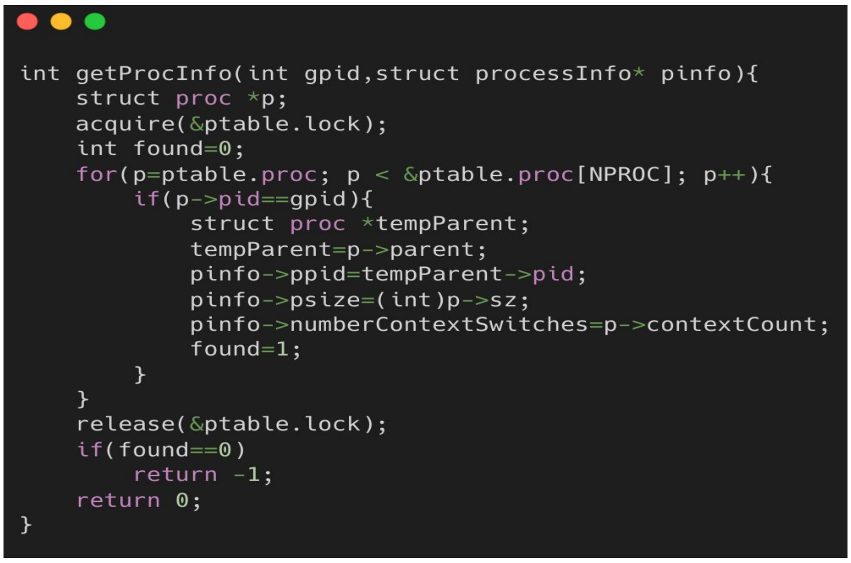
```
353+1 records out
181096 bytes (181 kB, 177 KiB) copied, 0.000925042 s, 196 MB/s
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 13828
echo           2 4 12836
forktest       2 5 8276
grep           2 6 15704
init           2 7 13424
kill           2 8 12888
ln             2 9 12792
ls             2 10 14976
mkdir          2 11 12972
rm             2 12 12952
sh             2 13 23436
stressfs       2 14 13616
usertests      2 15 56552
wc             2 16 14372
zombie         2 17 12616
getNumProc     2 18 12664
getMaxPid      2 19 12660
getProcInfo    2 20 13300
setBurstTime   2 21 13180
getBurstTime   2 22 12872
PartBSJF       2 23 15916
console        3 24 0
$ getNumProc
Number of active processes = 3
$ getMaxPid
MAX ID of the process running = 5
$ |
```

2. getProcInfo(int pid, struct processInfo* pInfo)

Apart from the files modified in the **PartA1** here, we need to modify the struct proc in the file ***proc.h*** by adding an integer variable `ContextCount` which will be used to return the ***numberContextSwitch*** variable in the *structure processInfo*

We increment the ContextCount variable every time a process changes state from running to runnable in the *scheduler()* function in ***proc.c*** file

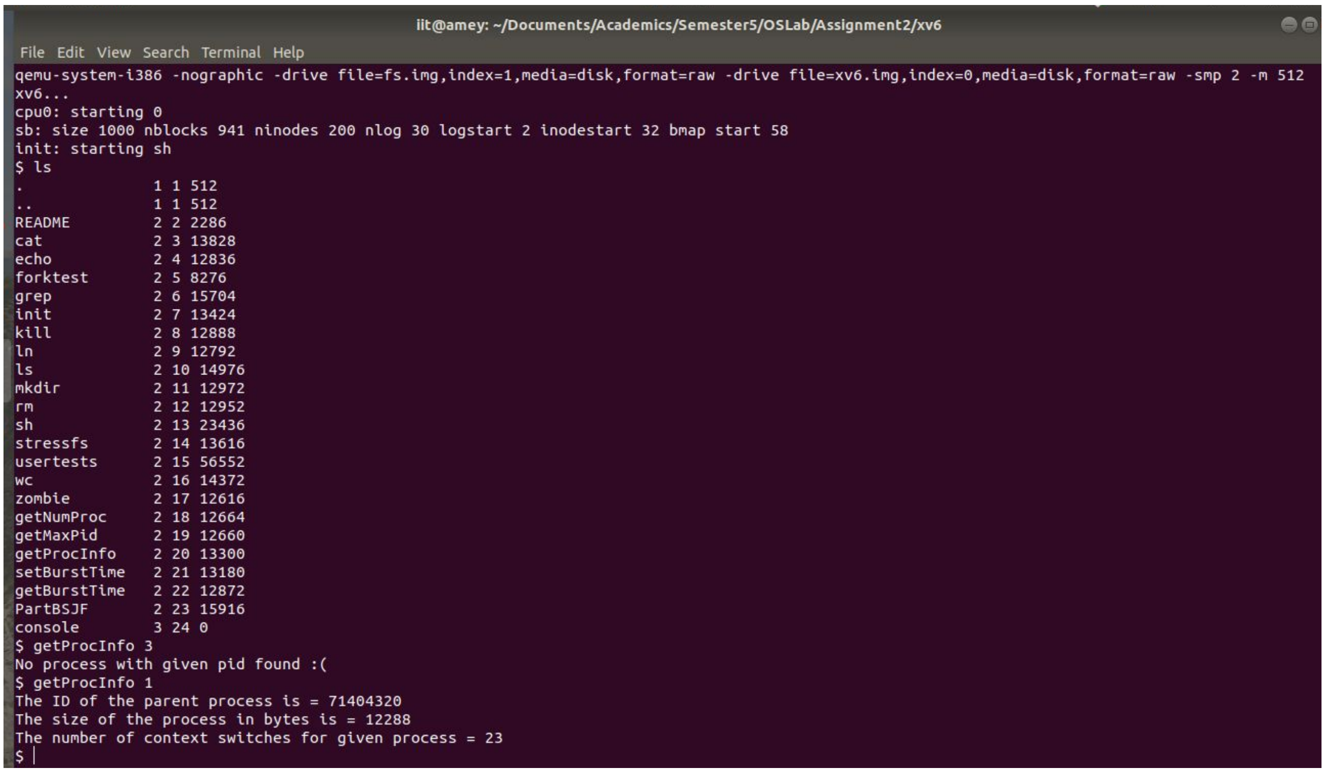
Code for **getProcInfo** can be found in **proc.c** file. Following is the code snippet :



```
int getProcInfo(int gpid,struct processInfo* pinfo){
    struct proc *p;
    acquire(&ptable.lock);
    int found=0;
    for(p=ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid==gpid){
            struct proc *tempParent;
            tempParent=p->parent;
            pinfo->ppid=tempParent->pid;
            pinfo->psize=(int)p->sz;
            pinfo->numberContextSwitches=p->contextCount;
            found=1;
        }
    }
    release(&ptable.lock);
    if(found==0)
        return -1;
    return 0;
}
```

.

Following is the output for the getProcInfo system call :


iit@amey: ~/Documents/Academics/Semester5/OSLab/Assignment2/xv6
File Edit View Search Terminal Help

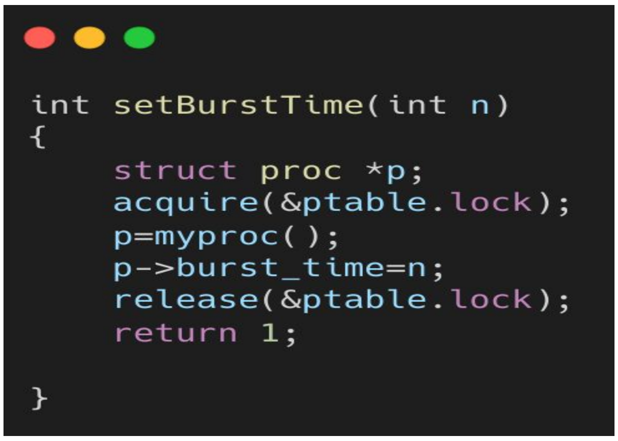
```
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 13828
echo           2 4 12836
forktest       2 5 8276
grep           2 6 15704
init           2 7 13424
kill           2 8 12888
ln             2 9 12792
ls             2 10 14976
mkdir          2 11 12972
rm             2 12 12952
sh             2 13 23436
stressfs       2 14 13616
usertests      2 15 56552
wc             2 16 14372
zombie         2 17 12616
getNumProc     2 18 12664
getMaxPid      2 19 12660
getProcInfo    2 20 13300
setBurstTime   2 21 13180
getBurstTime   2 22 12872
PartBSJF       2 23 15916
console        3 24 0
$ getProcInfo 3
No process with given pid found :(
$ getProcInfo 1
The ID of the parent process is = 71404320
The size of the process in bytes is = 12288
The number of context switches for given process = 23
$ |
```

3. get_burst_time(int n) and set_burst_time()

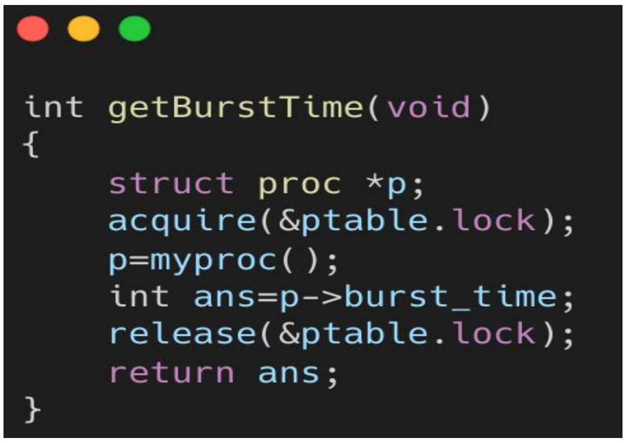
The *set_burst_time(n)* function is used to allot burst time [here between 1 to 20] to the current/running process when invoked through the user program. The current running process is provided to the function through the myproc() function. The attribute burst_time included instruct processInfo does not represent the actual burst time of a process, i.e the actual time on CPU needed by the process, but is rather a

custom attribute added in it by us. This is later used as a substitute for the processes priority values while implementing Shortest Job First scheduling.
The *get_burst_time()* is used to verify the working of *set_burst_time()* as it returns the burst time set through it.

Below is the code to implement to above functions
Both the system calls are invoked through just one file getBurstTime.c

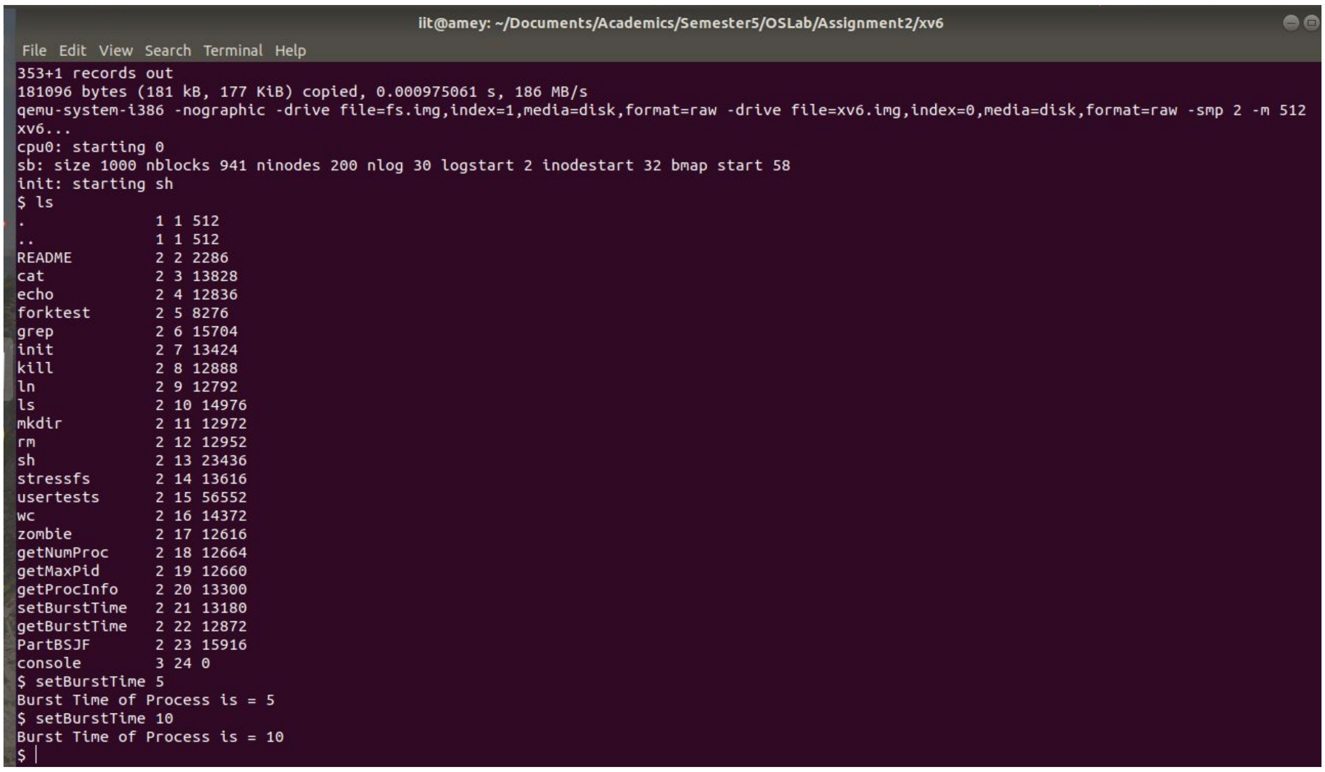


```
int setBurstTime(int n)
{
     struct proc *p;
     acquire(&ptable.lock);
     p=myproc();
     p->burst_time=n;
     release(&ptable.lock);
     return 1;

}
```



```
int getBurstTime(void)
{
     struct proc *p;
     acquire(&ptable.lock);
     p=myproc();
     int ans=p->burst_time;
     release(&ptable.lock);
     return ans;
}
```

Output for the above system calls:


iit@amey: ~/Documents/Academics/Semester5/OSLab/Assignment2/xv6
File Edit View Search Terminal Help

```
353+1 records out
181096 bytes (181 kB, 177 KiB) copied, 0.000975061 s, 186 MB/s
qemu-system-i386 -nographic -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
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 13828
echo           2 4 12836
forktest       2 5 8276
grep           2 6 15704
init           2 7 13424
kill           2 8 12888
ln             2 9 12792
ls             2 10 14976
mkdir          2 11 12972
rm             2 12 12952
sh             2 13 23436
stressfs       2 14 13616
usertests      2 15 56552
wc             2 16 14372
zombie         2 17 12616
getNumProc     2 18 12664
getMaxPid      2 19 12660
getProcInfo    2 20 13300
setBurstTime   2 21 13180
getBurstTime   2 22 12872
PartBSJF       2 23 15916
console        3 24 0
$ setBurstTime 5
Burst Time of Process is = 5
$ setBurstTime 10
Burst Time of Process is = 10
$ |
```

Part B

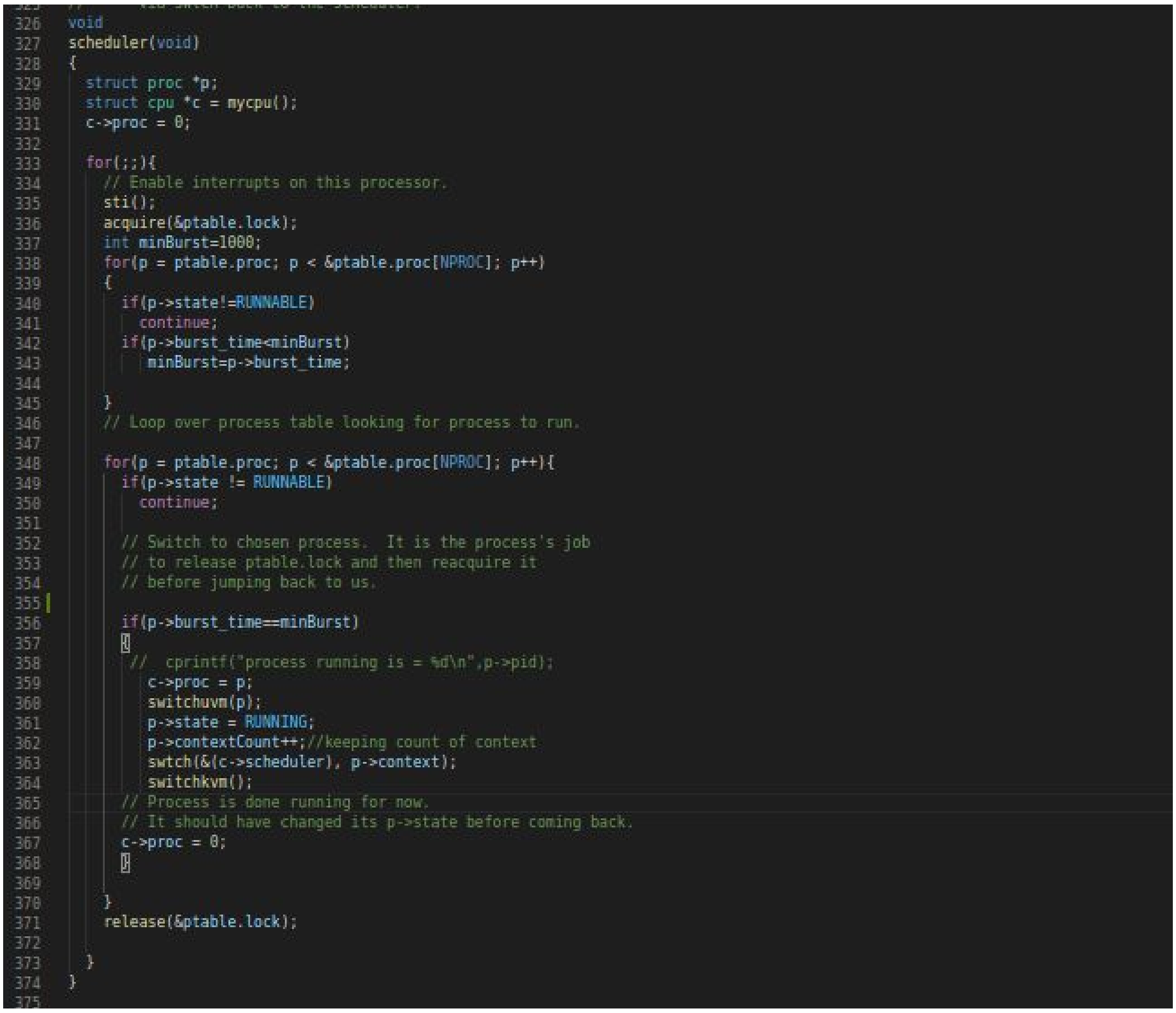
1. Shortest Job First(SJF)

Here we have implemented the Shortest Job First (SJF) Scheduling Algorithm. The default scheduling algorithm in xv6 is unweighted robin round. We have modified the *scheduler(void)* algorithm which now considers the job with the minimum burst_time among all the processes in the `RUNNABLE` state and then schedules it.

Design and Implementation:

We are using a linear search (linear in terms of number of steps) at each step/iteration of scheduling to select the next shortest job and hence the overall Asymptotic complexity of the Algorithm is O(n^2)
The average per-process time complexity is O(n) as each step involves a single linear search in SJF (Each step takes O(1) in Round Robin)

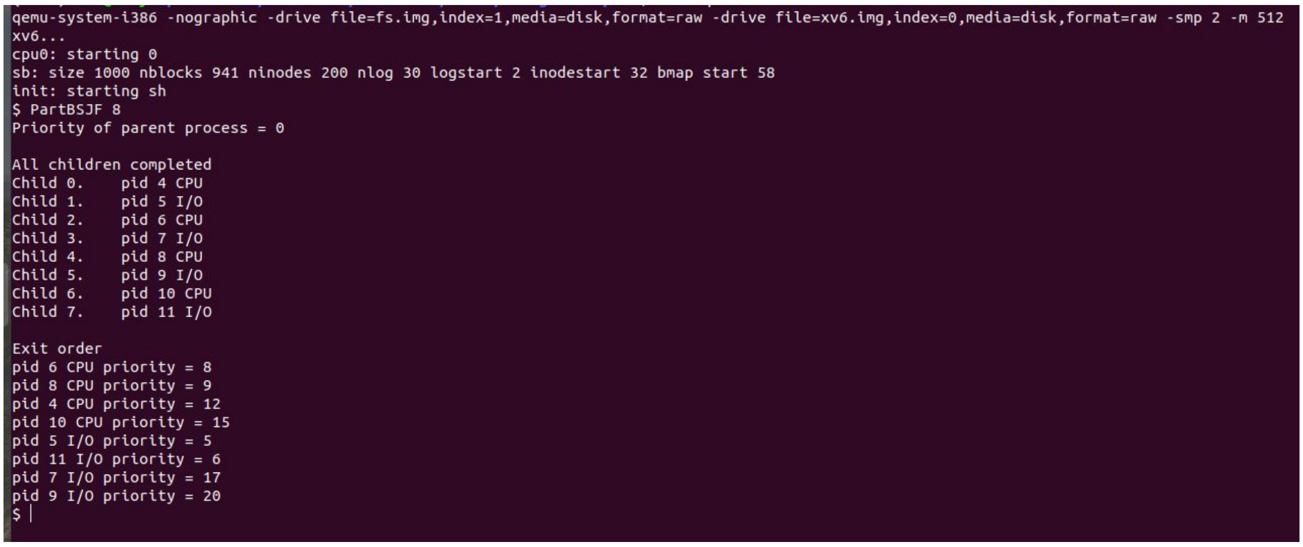
The below image details out the SJF Scheduler code



```c
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    acquire(&ptable.lock);
    int minBurst=1000;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if(p->state!=RUNNABLE)
        continue;
      if(p->burst_time<minBurst)
         minBurst=p->burst_time;

    }
    // Loop over process table looking for process to run.

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.

      if(p->burst_time==minBurst)
      {
       //  cprintf("process running is = %d\n",p->pid);
         c->proc = p;
         switchuvm(p);
         p->state = RUNNING;
         p->contextCount++;//keeping count of context
         swtch(&(c->scheduler), p->context);
         switchkvm();
      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
      }

    }
    release(&ptable.lock);

  }
}
```

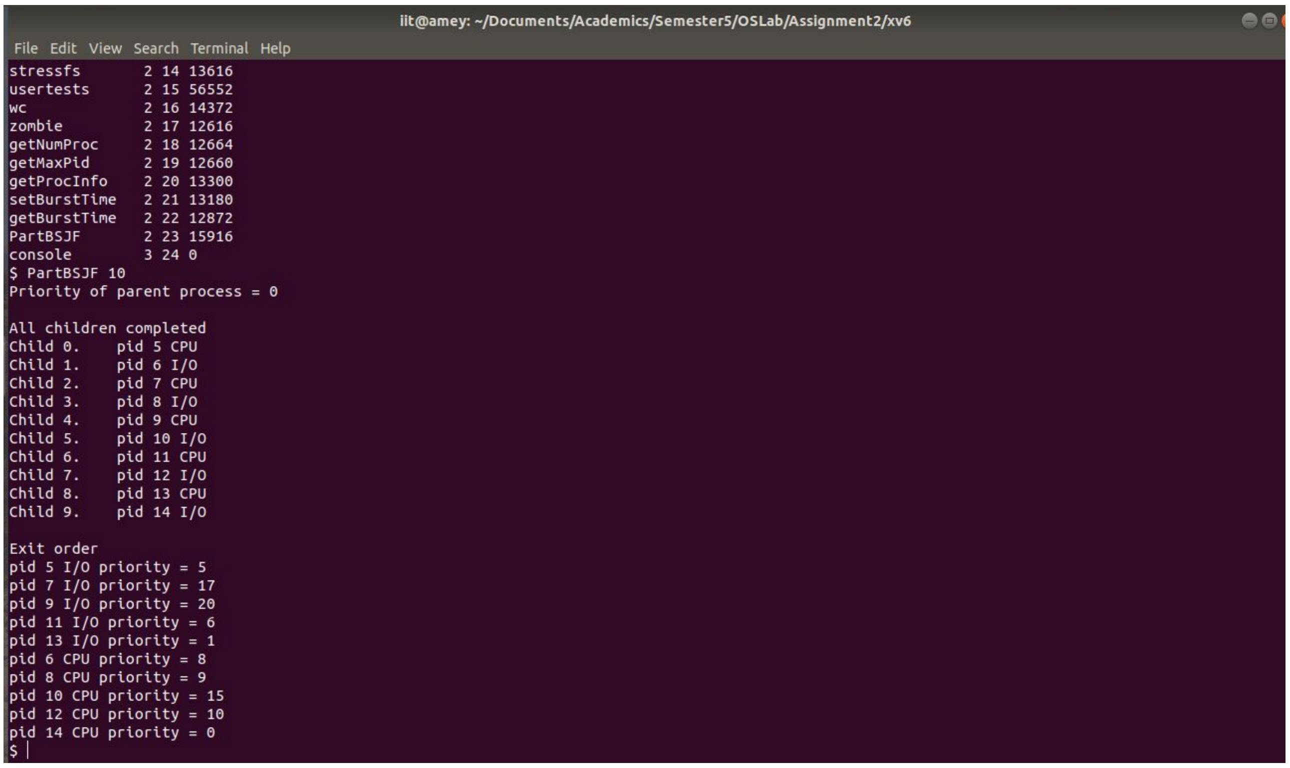
Proper locking discipline is followed and hence the code is suitable enough to run on a multiple-CPU system

The below image depicts the scheduling in **SJF implementation**
Give the number of child processes as an argument with the system call as shown below:
Since it is an SJF based scheduler, and the forked child processes alternate between being I/O bound and CPU bound, the output observed has completion of CPU bound processes before I/O bound processes. Within the completed processes of the same category, they are completed based on their burst_times, the one with the shortest is completed first.



```
qemu-system-i386 -nographic -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
$ PartBSJF 8
Priority of parent process = 0

All children completed
Child 0.    pid 4 CPU
Child 1.    pid 5 I/O
Child 2.    pid 6 CPU
Child 3.    pid 7 I/O
Child 4.    pid 8 CPU
Child 5.    pid 9 I/O
Child 6.    pid 10 CPU
Child 7.    pid 11 I/O

Exit order
pid 6 CPU priority = 8
pid 8 CPU priority = 9
pid 4 CPU priority = 12
pid 10 CPU priority = 15
pid 5 I/O priority = 5
pid 11 I/O priority = 6
pid 7 I/O priority = 17
pid 9 I/O priority = 20
$ |
```

The below image shows the **Round Robin scheduler implementation**



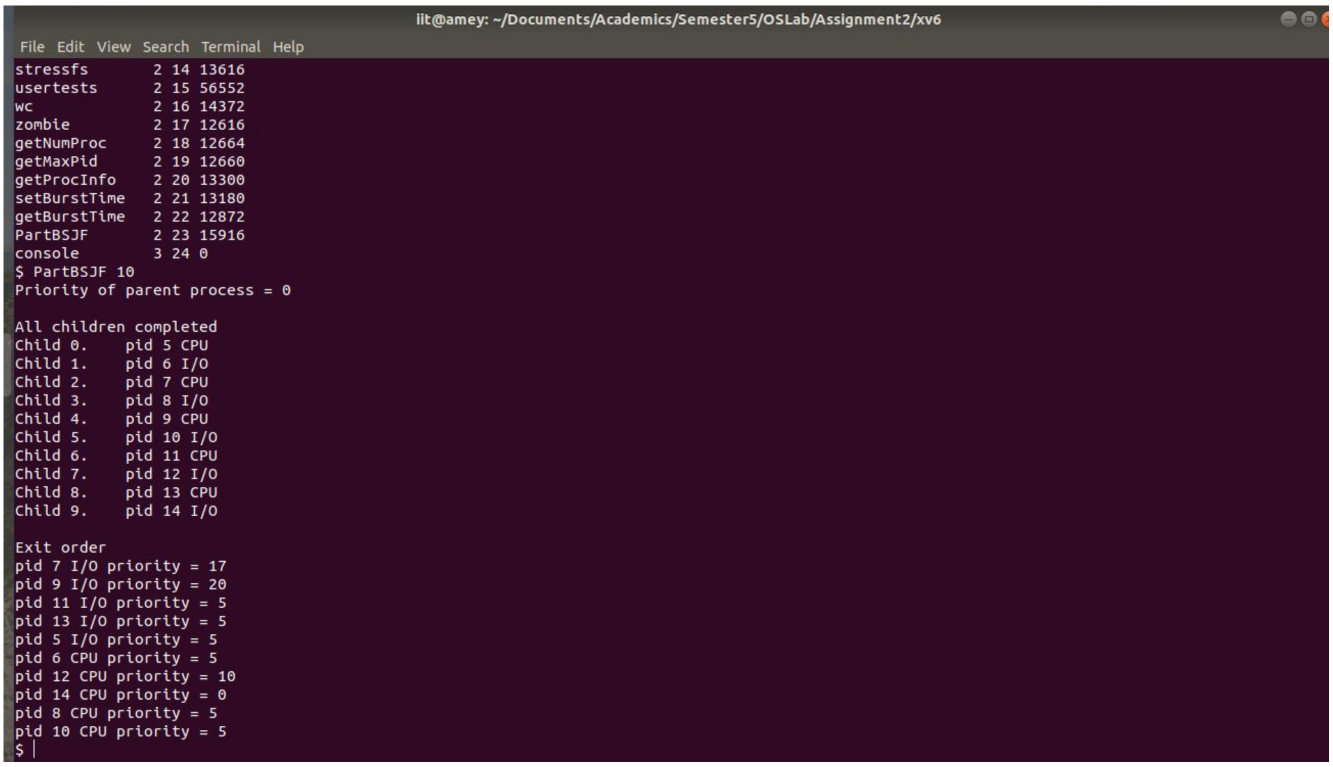
```
iit@amey: ~/Documents/Academics/Semester5/OSLab/Assignment2/xv6
stressfs       2 14 13616
usertests      2 15 56552
wc             2 16 14372
zombie         2 17 12616
getNumProc     2 18 12664
getMaxPid      2 19 12660
getProcInfo    2 20 13300
setBurstTime   2 21 13180
getBurstTime   2 22 12872
PartBSJF       2 23 15916
console        3 24 0
$ PartBSJF 10
Priority of parent process = 0

All children completed
Child 0.    pid 5 CPU
Child 1.    pid 6 I/O
Child 2.    pid 7 CPU
Child 3.    pid 8 I/O
Child 4.    pid 9 CPU
Child 5.    pid 10 I/O
Child 6.    pid 11 CPU
Child 7.    pid 12 I/O
Child 8.    pid 13 CPU
Child 9.    pid 14 I/O

Exit order
pid 5 I/O priority = 5
pid 7 I/O priority = 17
pid 9 I/O priority = 20
pid 11 I/O priority = 6
pid 13 I/O priority = 1
pid 6 CPU priority = 8
pid 8 CPU priority = 9
pid 10 CPU priority = 15
pid 12 CPU priority = 10
pid 14 CPU priority = 0
$ |
```

We have a mixture both of CPU Bound and I/O Bound processes so when we have a Round Robin implementation, the I/O Bound[I/O bursts are much more compared to the CPU bursts] processes are scheduled before the CPU Bound processes as this is a preemptive scheduling Algorithm and I/O Bound processes will have more context switches and will be preempted more frequently. This reduces the overall waiting times and response time as well.
Each process gets put back at the end of the queue no matter how much or how little of the quantum was used. I/O bound processes tend to run for a short period of time and then block which means they might have to wait in the queue for a long time.

If any process blocks then the next one is scheduled and the current is placed at the end of the queue
In round-robin scheduling, among the processes of the same category, those with the minimum PID's are completed first, as the ready queue is by default set on the FIFO principle.

Corner cases :

When multiple processes have the same **Burst Time**

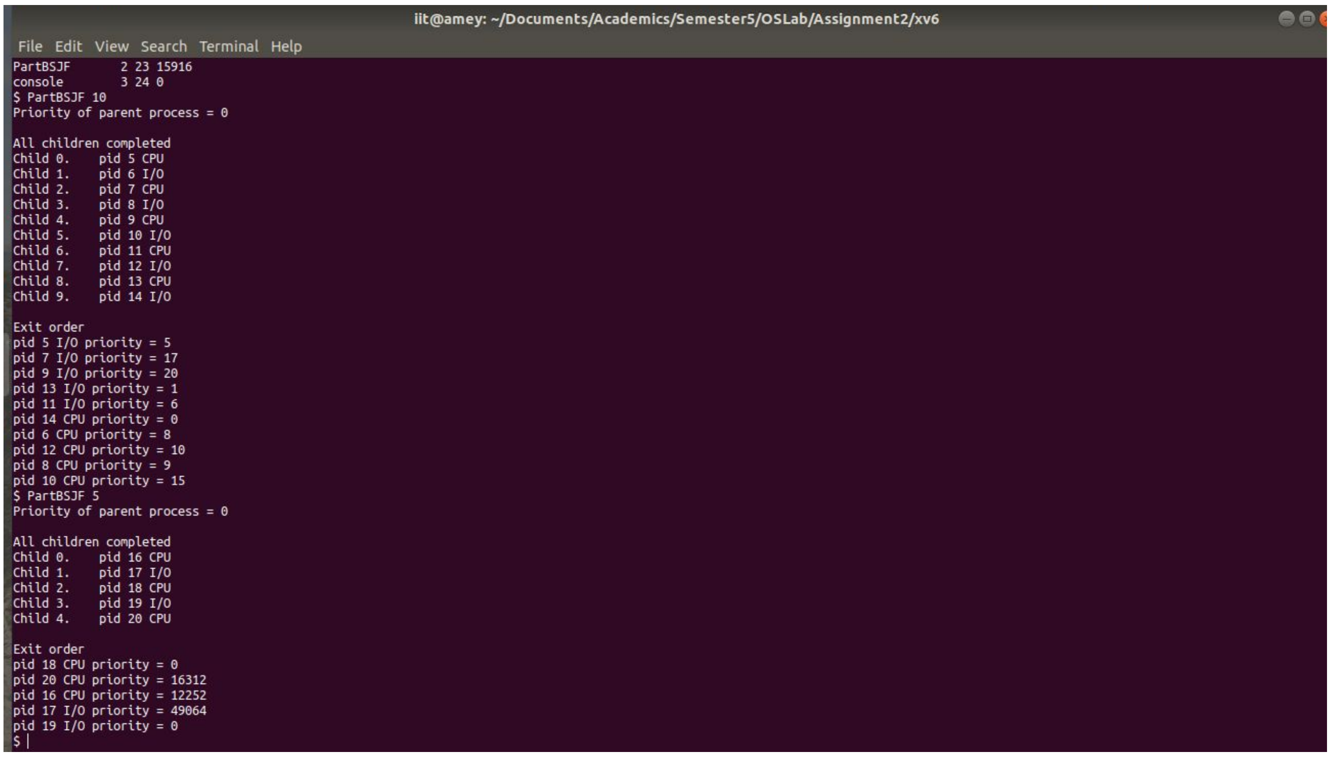


```
stressfs       2 14 13616
usertests      2 15 56552
wc             2 16 14372
zombie         2 17 12616
getNumProc     2 18 12664
getMaxPid      2 19 12660
getProcInfo    2 20 13300
setBurstTime   2 21 13180
getBurstTime   2 22 12872
PartBSJF       2 23 15916
console        3 24 0
$ PartBSJF 10
Priority of parent process = 0

All children completed
Child 0.    pid 5 CPU
Child 1.    pid 6 I/O
Child 2.    pid 7 CPU
Child 3.    pid 8 I/O
Child 4.    pid 9 CPU
Child 5.    pid 10 I/O
Child 6.    pid 11 CPU
Child 7.    pid 12 I/O
Child 8.    pid 13 CPU
Child 9.    pid 14 I/O

Exit order
pid 7 I/O priority = 17
pid 9 I/O priority = 20
pid 11 I/O priority = 5
pid 13 I/O priority = 5
pid 5 I/O priority = 5
pid 6 CPU priority = 5
pid 12 CPU priority = 10
pid 14 CPU priority = 0
pid 8 CPU priority = 5
pid 10 CPU priority = 5
$ |
```

We have observed that whenever two processes have the same Burst Time the process which has arrived first(not applicable here as all are assumed to arrive at the time 0) or the one which has a lesser process ID is scheduled first in SJF. This is taken care of by our implementation.

Multiple CPUs:

This image portrays how the scheduling works in cases of **multiple CPUs**



```
PartBSJF       2 23 15916
console        3 24 0
$ PartBSJF 10
Priority of parent process = 0

All children completed
Child 0.    pid 5 CPU
Child 1.    pid 6 I/O
Child 2.    pid 7 CPU
Child 3.    pid 8 I/O
Child 4.    pid 9 CPU
Child 5.    pid 10 I/O
Child 6.    pid 11 CPU
Child 7.    pid 12 I/O
Child 8.    pid 13 CPU
Child 9.    pid 14 I/O

Exit order
pid 5 I/O priority = 5
pid 7 I/O priority = 17
pid 9 I/O priority = 20
pid 13 I/O priority = 1
pid 11 I/O priority = 6
pid 14 CPU priority = 0
pid 6 CPU priority = 8
pid 12 CPU priority = 10
pid 8 CPU priority = 9
pid 10 CPU priority = 15
$ PartBSJF 5
Priority of parent process = 0

All children completed
Child 0.    pid 16 CPU
Child 1.    pid 17 I/O
Child 2.    pid 18 CPU
Child 3.    pid 19 I/O
Child 4.    pid 20 CPU

Exit order
pid 18 CPU priority = 0
pid 20 CPU priority = 16312
pid 16 CPU priority = 12252
pid 17 I/O priority = 49064
pid 19 I/O priority = 0
$ |
```

Whenever there are multiple processors then processes are scheduled as desired by the Algorithm i.e Shortest first but they are running on different CPUs so their completion/exit times are not necessarily the same as they are running “independently” and the scheduling algorithm doesn’t have much ‘role’ as there is no scarcity of resources(CPU)

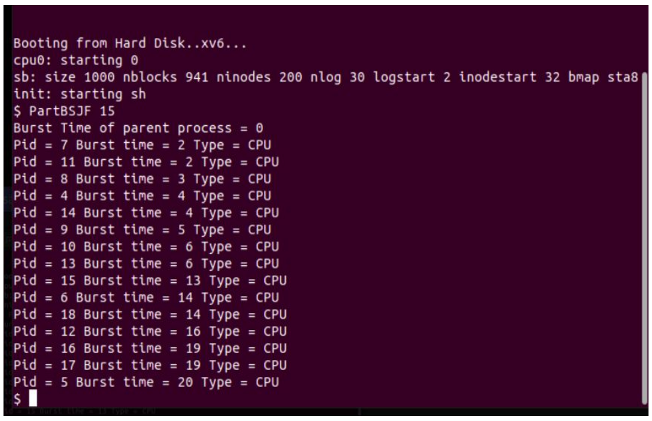
2. Hybrid of Shortest Job First(SJF) and Robin Round Algorithm

For hybrid we will have to change three files namely **proc.c, defs.h and trap.c**.
In defs.h we have to define the variable extern int time_quantum. This variable need to be initialized in the proc.c. We have implemented the algorithm in the scheduler function in **proc.c** .

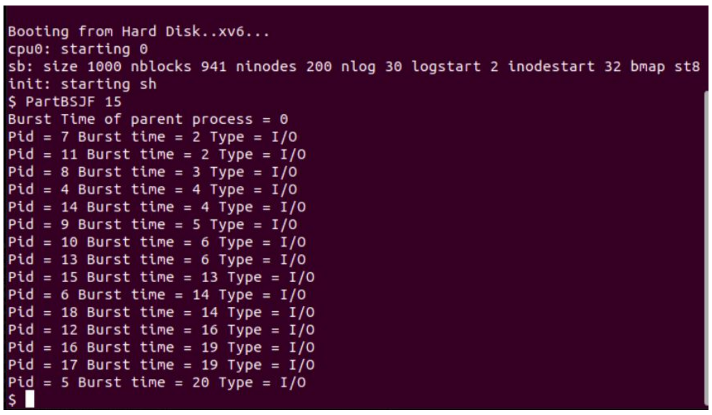
Design and implementation
We have implemented two queues. Whenever we find a RUNNABLE function we insert it into the queue RQ1 and call the sort function. The second queue RQ2 is used to store the process whose round is completed and and are waiting for their next round. Shortest process is extracted from RQ1(at the beginning of the queue) by min_process function and is sent for scheduling. If the burst time is greater than the time quantum the burst time is reduced by time quantum and then pushed in RQ2 else the process is completed. When the RQ1 becomes empty, all the processes are shifted from RQ2 to RQ1.
Time complexity - O(n^2)   (insertion sort).

Run_time attribute is added in the struct of process. In the trap.c file, we then change the interrupt handler function for implementing the round robin with the SJF.

Following is the output :

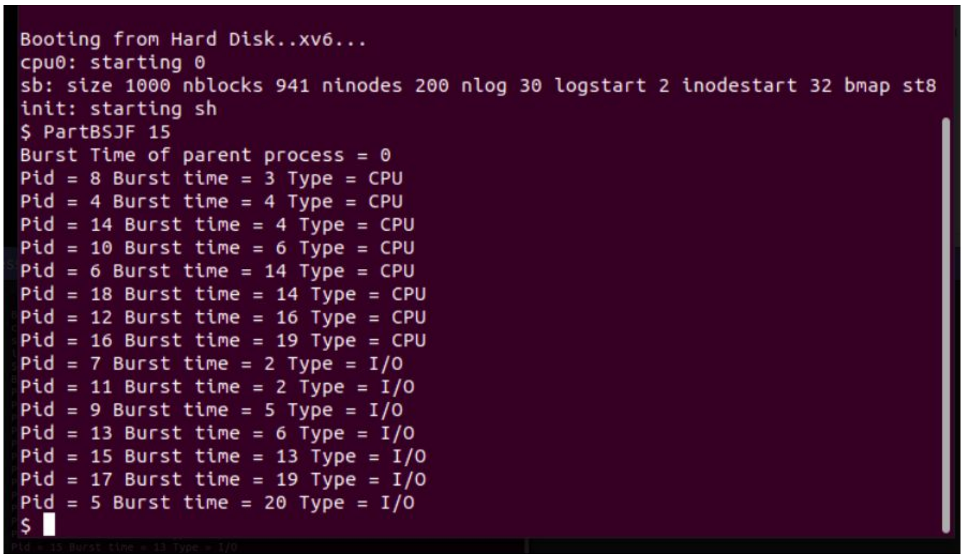


```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ PartBSJF 15
Burst Time of parent process = 0
Pid = 7 Burst time = 2 Type = CPU
Pid = 11 Burst time = 2 Type = CPU
Pid = 8 Burst time = 3 Type = CPU
Pid = 4 Burst time = 4 Type = CPU
Pid = 14 Burst time = 4 Type = CPU
Pid = 9 Burst time = 5 Type = CPU
Pid = 10 Burst time = 6 Type = CPU
Pid = 13 Burst time = 6 Type = CPU
Pid = 15 Burst time = 13 Type = CPU
Pid = 6 Burst time = 14 Type = CPU
Pid = 18 Burst time = 14 Type = CPU
Pid = 12 Burst time = 16 Type = CPU
Pid = 16 Burst time = 19 Type = CPU
Pid = 17 Burst time = 19 Type = CPU
Pid = 5 Burst time = 20 Type = CPU
$ 
```



```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap st8
init: starting sh
$ PartBSJF 15
Burst Time of parent process = 0
Pid = 7 Burst time = 2 Type = I/O
Pid = 11 Burst time = 2 Type = I/O
Pid = 8 Burst time = 3 Type = I/O
Pid = 4 Burst time = 4 Type = I/O
Pid = 14 Burst time = 4 Type = I/O
Pid = 9 Burst time = 5 Type = I/O
Pid = 10 Burst time = 6 Type = I/O
Pid = 13 Burst time = 6 Type = I/O
Pid = 15 Burst time = 13 Type = I/O
Pid = 6 Burst time = 14 Type = I/O
Pid = 18 Burst time = 14 Type = I/O
Pid = 12 Burst time = 16 Type = I/O
Pid = 16 Burst time = 19 Type = I/O
Pid = 17 Burst time = 19 Type = I/O
Pid = 5 Burst time = 20 Type = I/O
$ 
```

The output includes both the CPU bound and IO bound processes. Both IO and CPU bound processes are completed in increasing order of their burst time.(priority = burst time).



```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap st8
init: starting sh
$ PartBSJF 15
Burst Time of parent process = 0
Pid = 8 Burst time = 3 Type = CPU
Pid = 4 Burst time = 4 Type = CPU
Pid = 14 Burst time = 4 Type = CPU
Pid = 10 Burst time = 6 Type = CPU
Pid = 6 Burst time = 14 Type = CPU
Pid = 18 Burst time = 14 Type = CPU
Pid = 12 Burst time = 16 Type = CPU
Pid = 16 Burst time = 19 Type = CPU
Pid = 7 Burst time = 2 Type = I/O
Pid = 11 Burst time = 2 Type = I/O
Pid = 9 Burst time = 5 Type = I/O
Pid = 13 Burst time = 6 Type = I/O
Pid = 15 Burst time = 13 Type = I/O
Pid = 17 Burst time = 19 Type = I/O
Pid = 5 Burst time = 20 Type = I/O
$
```

The following files were modified/added [included in the zip folder] :

1) Trap.c
2) Usys.S
3) User.h
4) Proc.c
5) Proc.h
6) getNumProc.c
7) getMaxPid.c
8) getBurstTime.c
9) setBurstTime.c
10) ProcInfo.c
11) PartBSJF.c
12) Syscall.h
13) Syscall.c
14) Makefile
15) Defs.h
16) Sysproc.c
17) Param.h