CS344 (OS LAB) ASSIGNMENT 2

NAME	RITIK MANDLOI	ARYAN CHAUHAN	KOTKAR ANKET SANJAY	RAHUL MALA
ROLL NO.	180101066	180101012	180101037	180101062

PART A)

1)

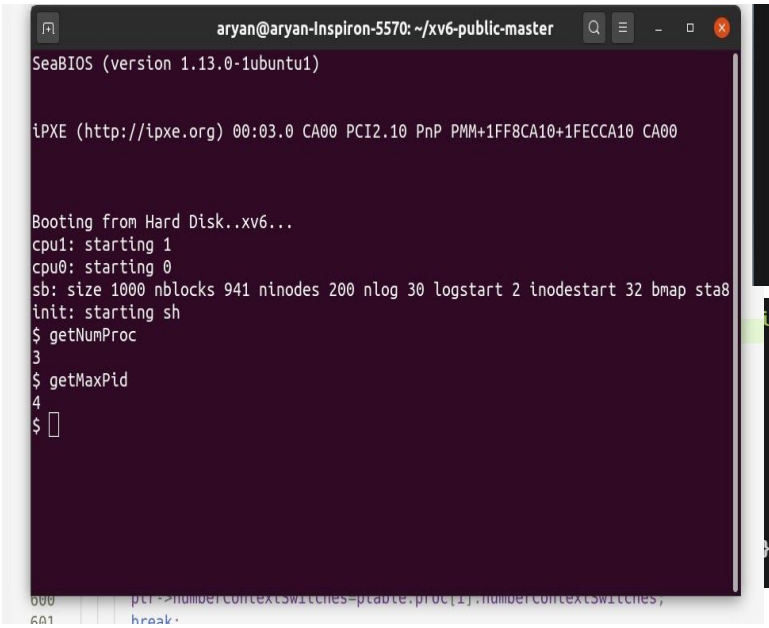
i) getNumProc()

Returns the total number of active processes in the system (either in embryo, running, runnable, sleeping, or zombie states). **We loop through all the processes in the process table (ptable) and increment the counter if any of the above states is encountered.** To test run the **getNumProc** command**.**

ii) getMaxPid()

Returns the maximum PID amongst the PIDs of all currently active (i.e., occupying a slot in the process table) processes in the system. **We loop through all the processes in the process table (ptable) and maintain the maximum of the PIDs.** To test run the **getMaxPid** command**.**

In both the functions we need to acquire the lock before accessing ptable and release the lock later.




```
int getNumProc(void){
  int numactive=0;
  acquire(&ptable.lock);
  for(int i=0;i<NPROC;i++){
    if(ptable.proc[i].state!=UNUSED)numactive++;
  }
  release(&ptable.lock);
  return numactive;
}
```

```
int getMaxPid(void){
  int maxid=-1;
  acquire(&ptable.lock);
  for(int i=0;i<NPROC;i++){
      if(ptable.proc[i].pid>maxid)
        maxid=ptable.proc[i].pid;
  }
  release(&ptable.lock);
  return maxid;
}
```

2) **getProcInfo(pid, &processInfo)**

This function returns the values of some fields(**Parent PID, No. Of Context Switches, process size**) of a process with a given PID from the process table and also fills the required fields in the processInfo structure. If the process doesn't exist in the table, it returns -1.

Here also, we need to acquire the lock before accessing ptable and release the lock when we are done.

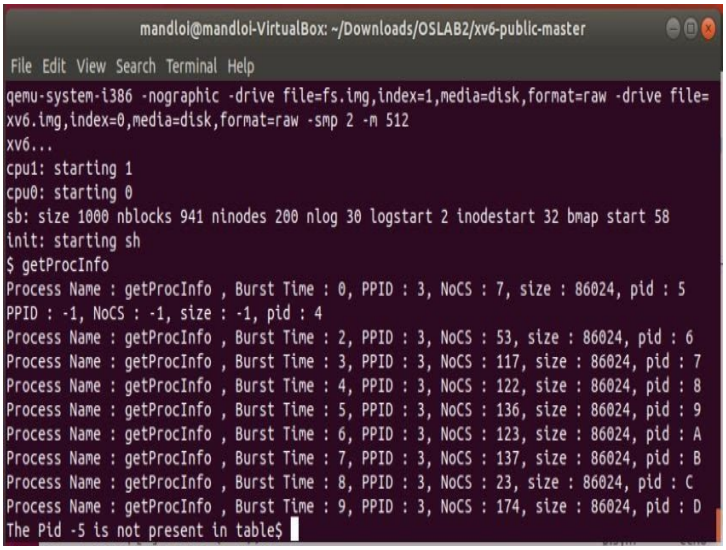
We added two new fields in the proc structure in proc.h

1) **burstTime**
2) **numberContextSwitches**

We initialized both of these fields in the **allocproc()** function in proc.c .

numberContextSwitches is initialized to 0 and gets incremented every time the process state changes to RUNNING in the **scheduler()** function in proc.c before exiting.

To test run the **getProcInfo** command.



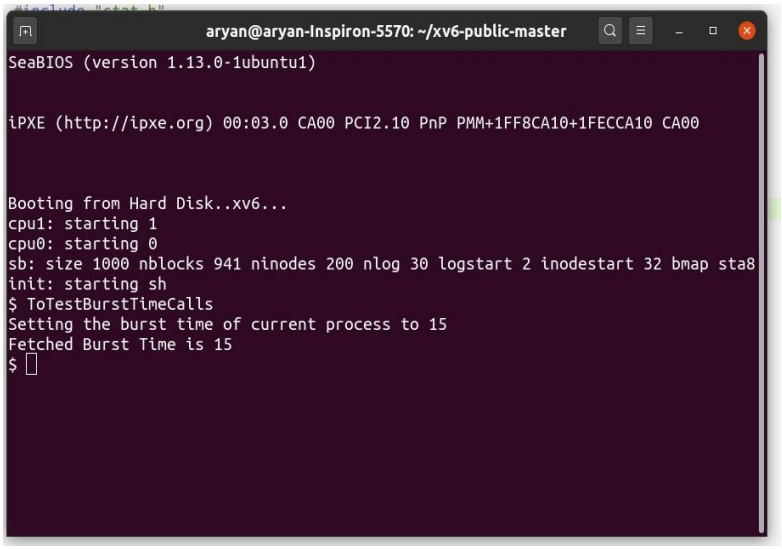

```c
int getProcInfo(int pid,struct processInfo* ptr){
  ptr->numberContextSwitches=-1;
  ptr->ppid=-1;
  ptr->psize=-1;
  if(pid<0 || pid>=NPROC)return -1;
  acquire(&ptable.lock);
  for(int i=0;i<NPROC;i++){
    if(ptable.proc[i].state!=UNUSED && ptable.proc[i].pid==pid){
      ptr->ppid=ptable.proc[i].parent->pid;
      ptr->psize=ptable.proc[i].sz;
      ptr->numberContextSwitches=ptable.proc[i].numberContextSwitches;
      break;
    }
  }
  release(&ptable.lock);
  if(ptr->numberContextSwitches==-1)return -1;
  return 0;
}
```

3)

i) **set_burst_time(n)**
A function to set the burst time of the process to a specified value n.

ii) **get_burst_time()**
A function to read the burst time of the process.

Utility functions to set and get the burst time of a process. To test, run the **ToTestBurstTimeCalls** command. Here we have used the xv6's **myproc()** function which gives a pointer to the current process for setting and getting the burst time.




```c
int set_burst_time(int n){
  myproc()->burstTime=n;
  return 0;
}

int get_burst_time(){
  cprintf("PID OF GET FUNC %d\n",myproc()->pid);
  return myproc()->burstTime;
}
```

PART B)

In the scheduler() function we simply pick the process which has the **least burst time** of all the processes thereby implementing the **strictly shortest job first scheme without any preemption**. To test run the **test_scheduler and second_tc** commands. Also to set the number of cpus to 1 we set **NCPU** in **param.h** to **1**(originally it was **8**).

IMPLEMENTATION DETAILS:

To find a process we have simply iterated over the **ptable.proc aka process table alongside keeping track of runnable process with minimum burst time.** So the running time complexity of the algorithm is **O(n)** where **n** is the number of processes. The scheduler is **non-preemptive**(This change is made by altering **trap.c** file).The cpu processing delays and io delays have been added into the test_scheduler(contains **decreasing** burst times) and second_tc files(contains **increasing** burst times) for implementation details read the **test_scheduler.c** file.

The **observations** made in both the test cases are that the **processes complete** in **ascending** order of the **burst time** i.e. the implementation of the shortest job first scheme is correct.

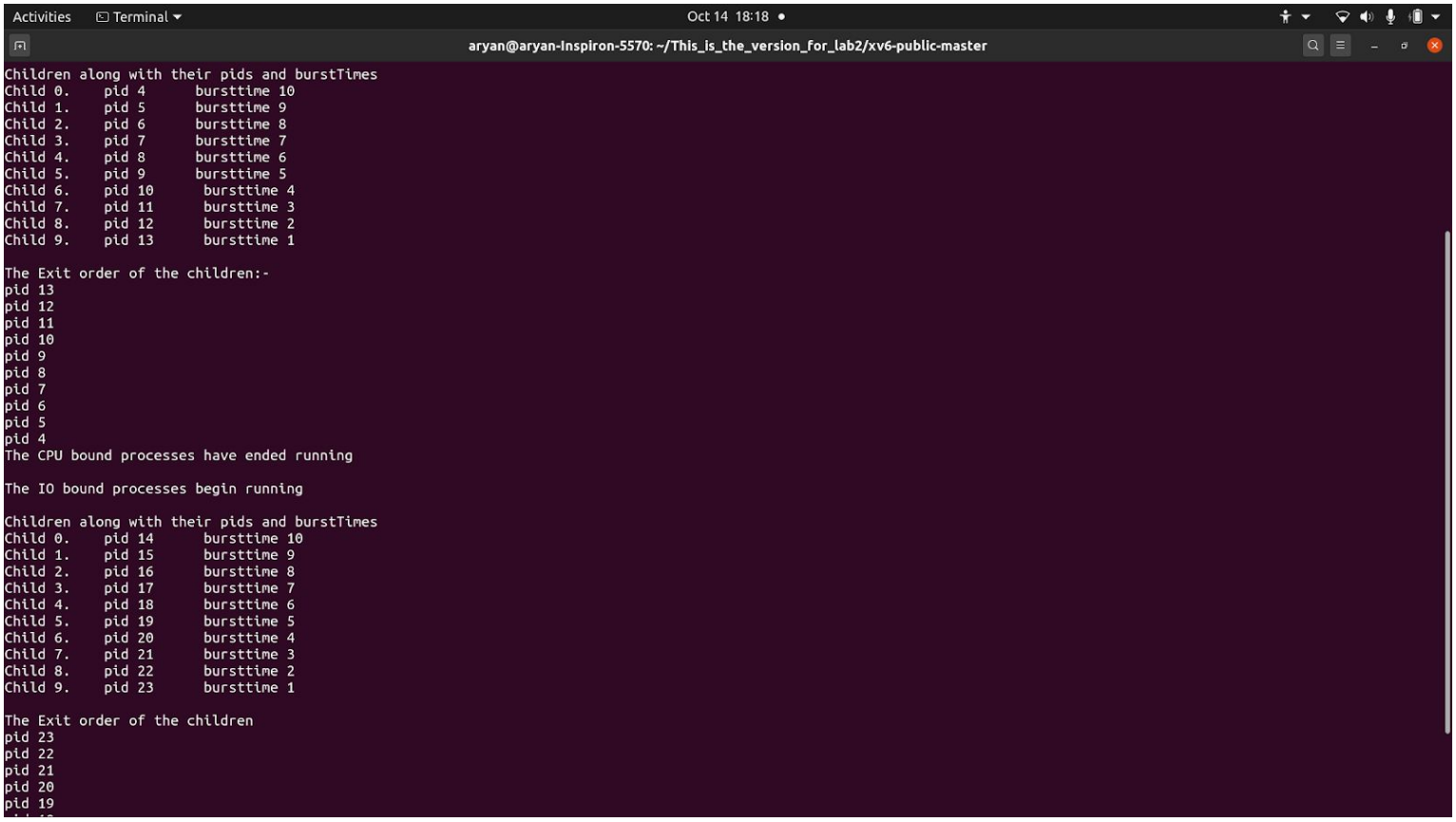
The **default** burst time of a process is set to **0**.

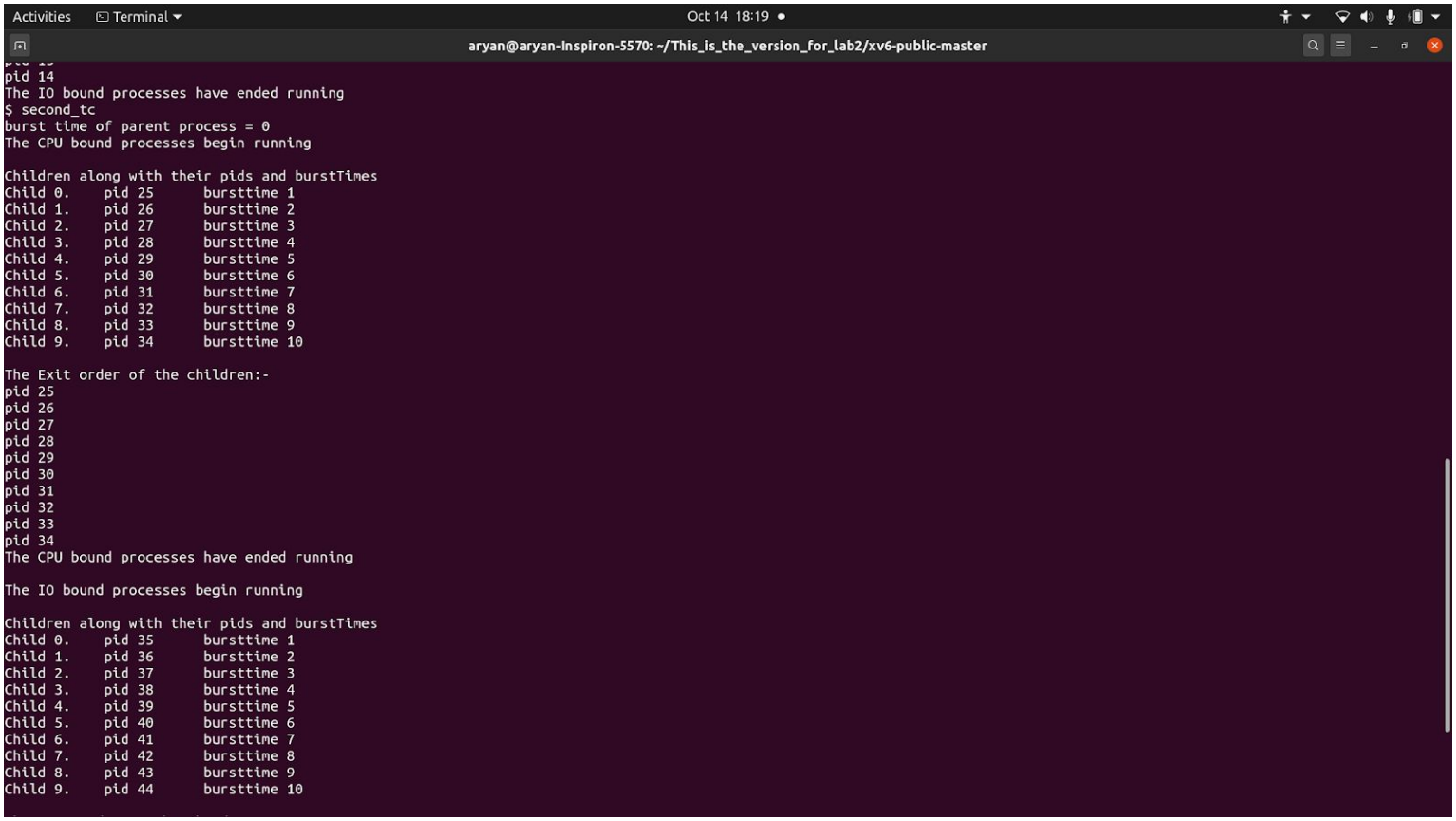
For **equal burst times** the scheduler picks the **first (by index in ptable.proc)** process which is in runnable state.

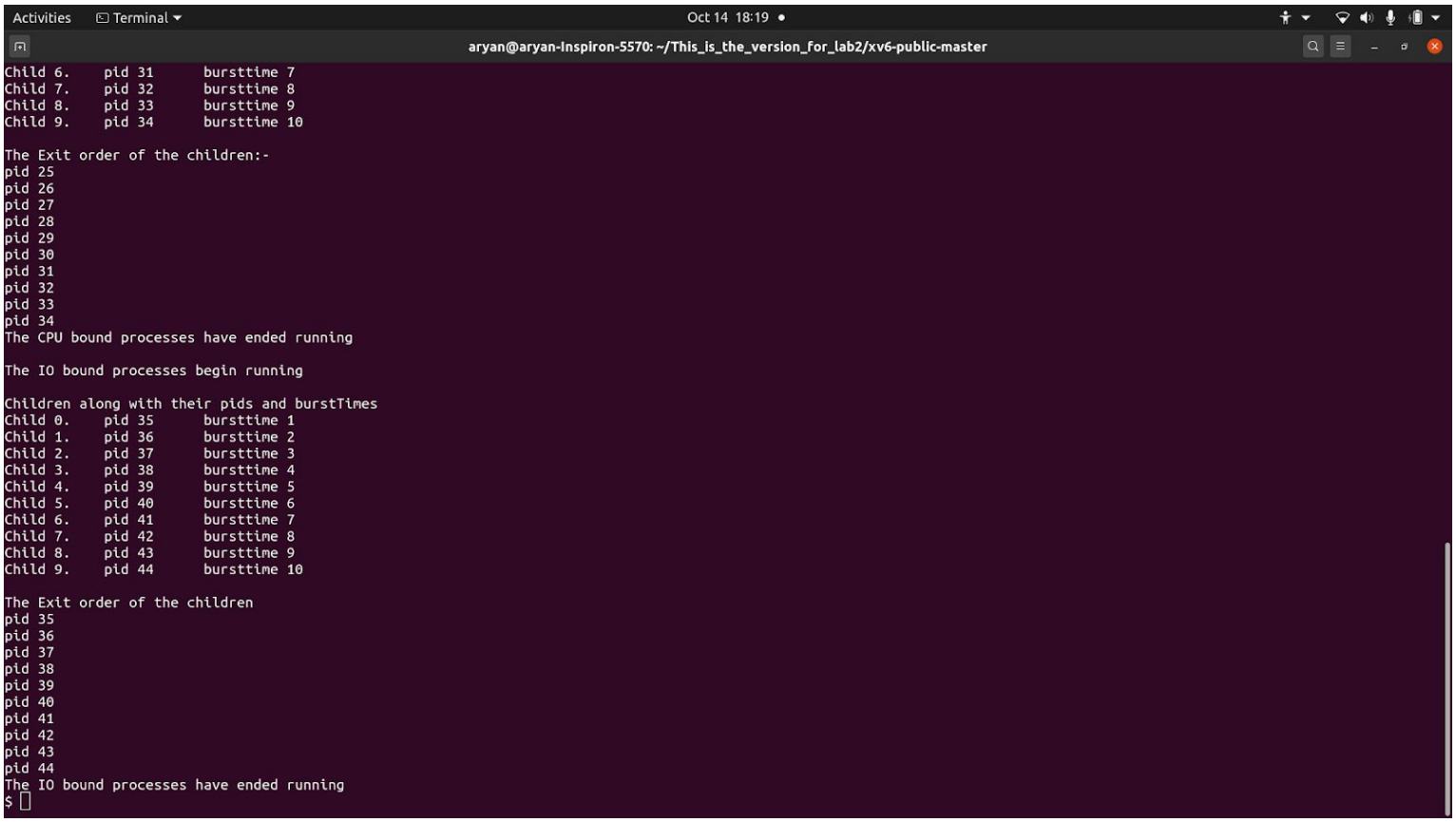
scheduler() function code :

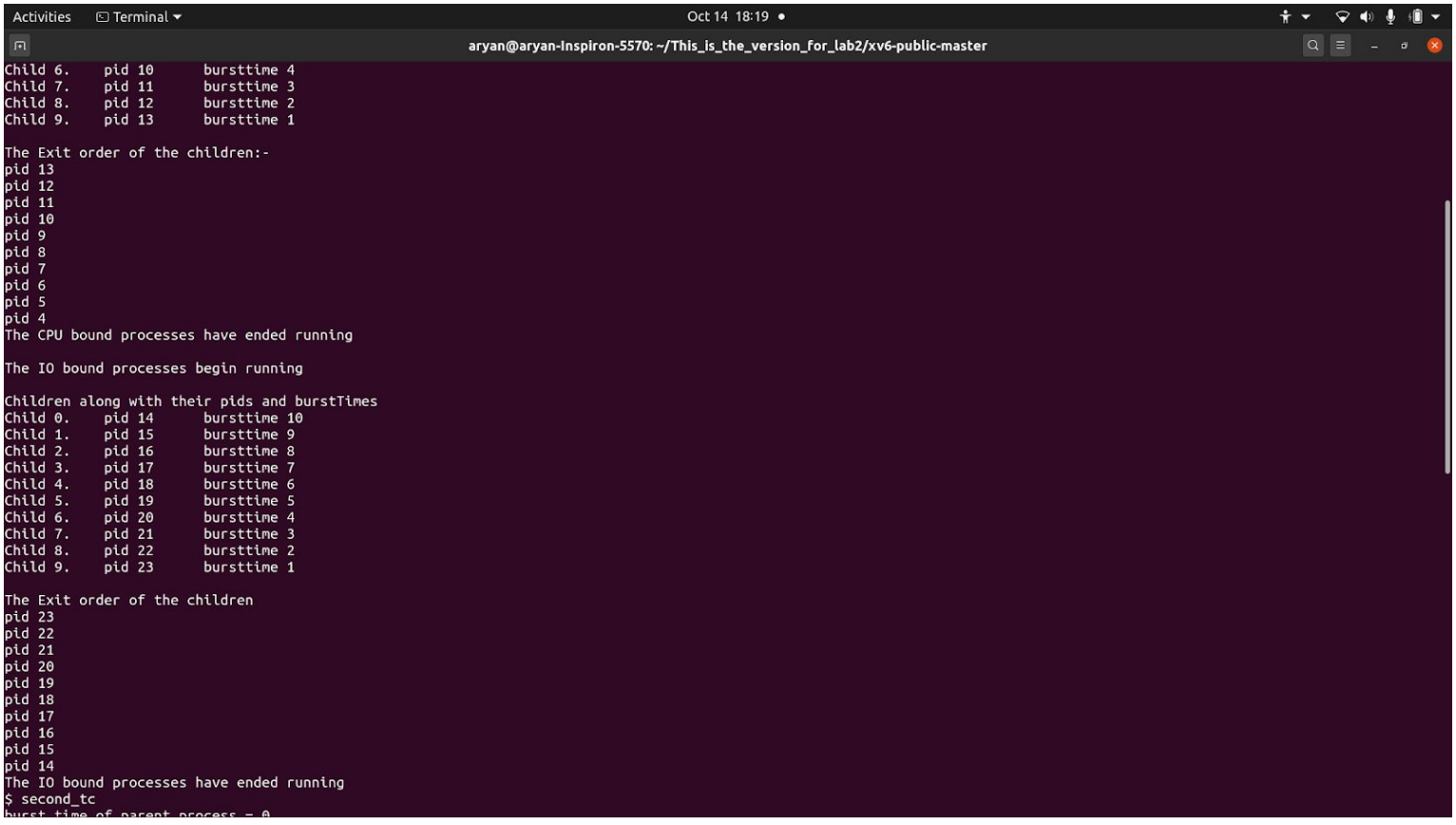
```
void scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    struct proc* selected_process=&ptable.proc[0];
    int currbursttime=-1;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      if(currbursttime==-1){
        currbursttime=p->burstTime;
        selected_process=p;
      }
      else if(currbursttime>p->burstTime){
        currbursttime=p->burstTime;
        selected_process=p;
      }
    }
    if(currbursttime!=-1){
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      // cprintf("PID %d   BURTSTIME %d\n",selected_process->pid,selected_process->burstTime);
      p=selected_process;
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      p->numberContextSwitches++;
      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);
  }
}
```

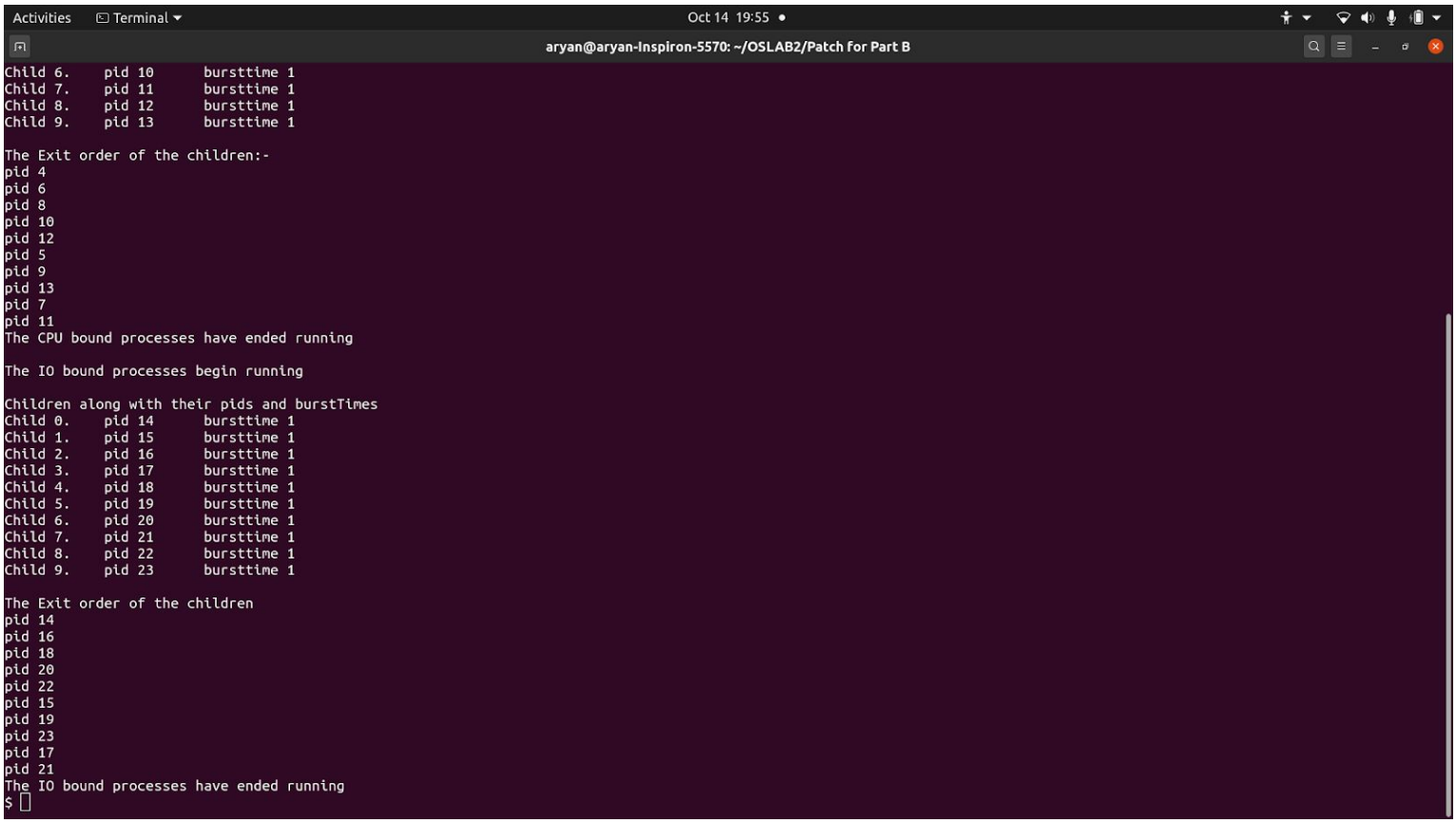
SCHEDULER TESTING






Activities
Terminal
Oct 14 18:19
aryan@aryan-Inspiron-5570: ~/This_is_the_version_for_lab2/xv6-public-master
Child 6. pid 31 bursttime 7
Child 7. pid 32 bursttime 8
Child 8. pid 33 bursttime 9
Child 9. pid 34 bursttime 10
The Exit order of the children:-
pid 25
pid 26
pid 27
pid 28
pid 29
pid 30
pid 31
pid 32
pid 33
pid 34
The CPU bound processes have ended running
The IO bound processes begin running
Children along with their pids and burstTimes
Child 0. pid 35 bursttime 1
Child 1. pid 36 bursttime 2
Child 2. pid 37 bursttime 3
Child 3. pid 38 bursttime 4
Child 4. pid 39 bursttime 5
Child 5. pid 40 bursttime 6
Child 6. pid 41 bursttime 7
Child 7. pid 42 bursttime 8
Child 8. pid 43 bursttime 9
Child 9. pid 44 bursttime 10
The Exit order of the children
pid 35
pid 36
pid 37
pid 38
pid 39
pid 40
pid 41
pid 42
pid 43
pid 44
The IO bound processes have ended running
$


Activities
Terminal
Oct 14 18:19
aryan@aryan-Inspiron-5570: ~/This_is_the_version_for_lab2/xv6-public-master
Child 6. pid 10 bursttime 4
Child 7. pid 11 bursttime 3
Child 8. pid 12 bursttime 2
Child 9. pid 13 bursttime 1
The Exit order of the children:-
pid 13
pid 12
pid 11
pid 10
pid 9
pid 8
pid 7
pid 6
pid 5
pid 4
The CPU bound processes have ended running
The IO bound processes begin running
Children along with their pids and burstTimes
Child 0. pid 14 bursttime 10
Child 1. pid 15 bursttime 9
Child 2. pid 16 bursttime 8
Child 3. pid 17 bursttime 7
Child 4. pid 18 bursttime 6
Child 5. pid 19 bursttime 5
Child 6. pid 20 bursttime 4
Child 7. pid 21 bursttime 3
Child 8. pid 22 bursttime 2
Child 9. pid 23 bursttime 1
The Exit order of the children
pid 23
pid 22
pid 21
pid 20
pid 19
pid 18
pid 17
pid 16
pid 15
pid 14
The IO bound processes have ended running
$ second_tc
burst time of parent process = 0


Activities
Terminal
Oct 14 19:55
aryan@aryan-Inspiron-5570: ~/OSLAB2/Patch for Part B
Child 6.   pid 10   bursttime 1
Child 7.   pid 11   bursttime 1
Child 8.   pid 12   bursttime 1
Child 9.   pid 13   bursttime 1
The Exit order of the children:-
pid 4
pid 6
pid 8
pid 10
pid 12
pid 5
pid 9
pid 13
pid 7
pid 11
The CPU bound processes have ended running
The IO bound processes begin running
Children along with their pids and burstTimes
Child 0.   pid 14   bursttime 1
Child 1.   pid 15   bursttime 1
Child 2.   pid 16   bursttime 1
Child 3.   pid 17   bursttime 1
Child 4.   pid 18   bursttime 1
Child 5.   pid 19   bursttime 1
Child 6.   pid 20   bursttime 1
Child 7.   pid 21   bursttime 1
Child 8.   pid 22   bursttime 1
Child 9.   pid 23   bursttime 1
The Exit order of the children
pid 14
pid 16
pid 18
pid 20
pid 22
pid 15
pid 19
pid 23
pid 17
pid 21
The IO bound processes have ended running
$