ASSIGNMENT 2

OS344 - Operating Systems Laboratory

G4 Group Members:

1. B Venkatesh 180101013
2. Harsh Motwani 180101028
3. Nishchay Manwani 180101051
4. Vaibhav Kumar Singh 180101086

Part A

(1) 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 also create user programs named **`getNumProcTest`** and **`getMaxPIDTest`** to use the above system calls. Two functions namely, **`getNumProcAssist`** and **`getMaxPIDAssist`** are implemented in `proc.c` which help us in achieving the desired functionalities. They are attached below.

```c
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;
}
```

```c
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;
}
```

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:

```
theharshshow@theharshshow-VirtualBox: ~/Desktop/CS344/x...
SeaBIOS (version 1.13.0-1ubuntu1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap st8
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16336
echo           2 4 15188
forktest       2 5 9496
grep           2 6 18556
init           2 7 15776
kill           2 8 15220
ln             2 9 15072
ls             2 10 17700
mkdir          2 11 15316
rm             2 12 15292
sh             2 13 27928
stressfs       2 14 16208
usertests      2 15 67312
wc             2 16 17072
zombie         2 17 14884
getNumProcTest 2 18 14956
getMaxPIDTest  2 19 14956
console        3 20 0
$ getMaxPIDTest
The maximum PID of all active processes in the system is 4
$ getNumProcTest
The total number of active processes in the system is 3
$ 
```

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, getMaxPIDTest is run. The next available process ID is 4 which is allotted to getMaxPIDTest. 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 getMaxPIDTest with PID 4. Hence, the output is 4. Similarly, when getNumProcTest is run, 3 processes are running (2 system processes and 1 getNumProcTest). Hence the output is 3.

(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 unlike the ones we previously made above. Also, we need to devise a way to store the number of context switches for every process.

We solve the problem of passing parameters to syscall using **argptr** which is a predefined system call which serves our purpose.

As for the context switch part, we will solve the problem by modifying struct proc to include one additional member named **nocs** which will store the number of context switches. Now we need to initialise noc for every process. We do this by setting **p->nocs** to 0 in allocproc() since before any process is run, it is allocated (assigned a place in ptable) using allocproc. The next thing we do is add the statement **(p->nocs)++** 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 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:

```c
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;

}
```

```c
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){
        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;

}
```

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 getProcInfoTest is given below:

```
venky@venky-VirtualBox: ~/xv6-public
SeaBIOS (version 1.13.0-1ubuntu1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk...
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ getProcInfoTest 1
PPID: No Parent Process
Size: 12288
Number of Context Switches: 25
$ getProcInfoTest 2
PPID: 1
Size: 16384
Number of Context Switches: 33
$ getProcInfoTest 3
No process has that PID.
$
```

getProcInfoTest takes process ID as a command line parameter which it then passes to getProcInfo which then passes it to getProcInfoAssist. GetProcInfoAssist 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) For this part, an additional attribute namely **burst_time** has to be added to **proc** structure. We also have to implement 2 system calls namely, **set_burst_time** and **get_burst_time** which will set and get the burst time of current process to a given value respectively. 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().
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 are given below.

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

  return p->burst_time;

}
```

```c
int
set_burst_timeAssist(int burst_time)
{
  struct proc *p = myproc();

  p->burst_time = burst_time;
  return 0;
}
```

To test the above processes, we create a user program named **burst_time_test**. It is shown below.

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

int
main(void){
    printf(1,"This is a sample process to test set_burst_time and get_burst_time system calls.\n");
    set_burst_time(3);
    printf(1, "The burst time is: %d\n", get_burst_time());
    exit();
}
```

The output when the above user program is run is shown below.

```
venky@venky-VirtualBox: ~/xv6-public
SeaBIOS (version 1.13.0-1ubuntu1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk...
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ burstTimeTest
This is a sample process to test set_burst_time and get_burst_time system calls.
The burst time is: 3
$ 
```

As we can see, the burst time is set to 3. It changes from the default burst time (0) to 3 indicating that the above functions are working correctly.

Part B (not the Bonus attempt) ***Bonus in next section**

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:

- 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:

```

  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  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:

```
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.):

```c
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 locks are acquired and released in the priority queue functions)**:

```c
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:

```
acquire(&ptable.lock);

short check = (np->state!=RUNNABLE);
np->state = RUNNABLE;

//Insert Process Into Queue
if(check)
  insertIntoHeap(np);

release(&ptable.lock);
```

- Insert a **yield** call into **set_burst_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;
}
```

Runtime 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
        ---      ----    ----------      ----------------

        21       CPU        2                  2
        11       CPU        3                  2
        7        CPU        6                  2
        13       CPU        6                  2
        17       CPU        9                  2
        19       CPU        14                 2
        15       CPU        14                 2
        23       CPU        17                 2
        9        CPU        20                 2
        5        CPU        20                 2
        6        I/O        1                  102
        8        I/O        2                  102
        10       I/O        6                  102
        20       I/O        9                  102
        4        I/O        13                 102
        22       I/O        15                 102
        12       I/O        17                 102
        18       I/O        18                 102
        16       I/O        19                 102
        14       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:**

```
$ cpuProcTester 4
        PID      Type     Burst Time      Context Switches
        ___      ____     __________      ________________

        4        CPU        13                 17
        5        CPU        20                 18
        6        CPU        1                  18
        7        CPU        6                  19
$
```

```
$ ioProcTester 4
        PID      Type     Burst Time      Context Switches
        ___      ____     __________      ________________

        12       I/O        13                 22
        13       I/O        20                 22
        14       I/O        1                  22
        15       I/O        6                  22
$
```

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

```
$ cpuProcTester 4
        PID      Type     Burst Time      Context Switches
        ___      ____     __________      ________________

        6        CPU        1                  2
        7        CPU        6                  2
        4        CPU        13                 2
        5        CPU        20                 2
$
```

```
$ ioProcTester 4
        PID      Type     Burst Time      Context Switches
        ___      ____     __________      ________________

        6        I/O        1                  22
        7        I/O        6                  22
        4        I/O        13                 22
        5        I/O        20                 22
$
```

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 **random.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.

PTO for Bonus! Bonus starts on next page.

Part B (Bonus Part)

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 **pqueue2** and the corresponding functions for this priority queue. Functions like insertIntoHeap, fix, extractMax, etc were created corresponding to **pqueue.** The corresponding functions for **pqueue2** are **insertIntoHeap2, extractMax2, fix2 etc.** The functions are the same as the original ones except they modify pqueue2 instead of pqueue so you can refer to the previous section to get details about the functions.

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 - rt** 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 **rt** 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 - 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 **pqueue2.**

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
{

  (myproc()->rt)++;
  if(myproc()->burst_time != 0){

    if(myproc()->burst_time == myproc()->rt)
      exit();
  }
  if((myproc()->rt)%quant == 0)
      new_yield();

}
```

```
void new_yield(void){

  acquire(&ptable.lock);

  myproc()->state = RUNNABLE;

  insertIntoHeap2(myproc());

  sched();
  release(&ptable.lock);

}
```

```
void
scheduler(void)
{
  defaultParent.pid = -2;
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for(;;){
    sti();
    acquire(&ptable.lock);
    if(isEmpty()){
      if(isEmpty2())
        goto label;
      while(!isEmpty2()){
        if((p = extractMin2()) == 0){release(&ptable.lock);break;}
        insertIntoHeap(p);
      }
    }
    label:
    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);
  }
}
```

Next, we modified the scheduler. Basically, in the new scheduler while deciding on which process to run next, we check if the original priority queue, **pqueue** is empty or not. If not, we **extractMin** from **pqueue** run the extracted process. If **pqueue** is empty, we remove all processes from **pqueue2** and insert them into **pqueue** (The processes that were preempted because of time quantum) and then we normally pick the front element from **pqueue** using extractMin.

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.

The output obtained from the above tests are shown below.

```
$ cpuProcTester 20
        PID       Type     Burst Time       Context Switches
        ---       ----     ----------       ----------------

        10        CPU         252                  9
        7         CPU         264                  9
        13        CPU         270                  9
        20        CPU         411                  9
        17        CPU         443                  9
        4         CPU         635                  9
        15        CPU         675                  9
        19        CPU         677                  9
        22        CPU         738                  9
        12        CPU         822                  9
        23        CPU         846                  9
        18        CPU         889                  9
        16        CPU         932                  9
        14        CPU         962                  9
        9         CPU         985                  9
        5         CPU         999                  9
$
```

```
$ ioProcTester 20
        PID       Type     Burst Time       Context Switches
        ---       ----     ----------       ----------------

        6         I/O         21                  12
        21        I/O         67                  12
        8         I/O         71                  12
        11        I/O         126                 12
        10        I/O         252                 12
        7         I/O         264                 12
        13        I/O         270                 12
        20        I/O         411                 12
        17        I/O         443                 12
        4         I/O         635                 12
        15        I/O         675                 12
        19        I/O         677                 12
        22        I/O         738                 12
        12        I/O         822                 12
        23        I/O         846                 12
        18        I/O         889                 12
        16        I/O         932                 12
        14        I/O         962                 12
        9         I/O         985                 12
        5         I/O         999                 12
$
```

```
$ test_scheduler 30
        PID       Type     Burst Time       Context Switches
        ---       ----     ----------       ----------------

        6         I/O         21                  12
        8         I/O         71                  12
        30        I/O         145                 12
        24        I/O         179                 12
        10        I/O         252                 12
        20        I/O         411                 12
        32        I/O         423                 12
        4         I/O         635                 12
        22        I/O         738                 12
        26        I/O         805                 12
        12        I/O         822                 12
        18        I/O         889                 12
        28        I/O         914                 12
        16        I/O         932                 12
        14        I/O         962                 12
        7         CPU         264                 9
        13        CPU         270                 9
        27        CPU         418                 9
        17        CPU         443                 9
        31        CPU         457                 9
        33        CPU         571                 9
        15        CPU         675                 9
        19        CPU         677                 9
        23        CPU         846                 9
        9         CPU         985                 9
        25        CPU         624                 10
        5         CPU         999                 10
$
```

Results

Here's the exciting part. Results were as we expected. There were three key 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 **rt** 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 preemption 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 preempted 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.
- **Note:** 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 preempted. This leads to out of order execution of some CPU bound processes.
- Also, if you want to compare it with the round robin scheduler, refer to the pictures in the previous section (**non BONUS section**) and compare them with the ones in this section.