# CS 344 ASSIGNMENT - 2

**GROUP MEMBERS :**

Arya Avinash Phadke 200101020
Mansi 200101064
Pranshu Pandya 200101073

---

## Part A

We made different functions in proc.c to implement each functionality as mentioned in the assignment.

1. **GetNumProc** - This has been implemented in the function getNumProc(). We are basically looping through the entries of ptable and the processes which are in unused state are being counted.

```
int getNumProc()
{
  acquire(&ptable.lock);
  int cnt = 0;
  for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state != UNUSED)
    {

      cnt++;
    }
    // else
    //   break;
    // printf("%p",p);
  }
  // printf("%d",cnt);
  release(&ptable.lock);
  return cnt;
}
```

2. **GetMaxPid** - This has been implemented in the function getMaxPid(). We are basically looping through the entries of ptable and finding the max Pid of the processes which are currently running (meaning not in unused state).

```c
int getMaxPid()
{
  acquire(&ptable.lock);
  int mxpid = -1;
  for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state != UNUSED)
    {

      mxpid = (mxpid > p->pid ? mxpid : p->pid);
    }
    // printf(1,"%d",p->pid);
    // printf("%p",p);
  }
  release(&ptable.lock);
  return mxpid;
}
```

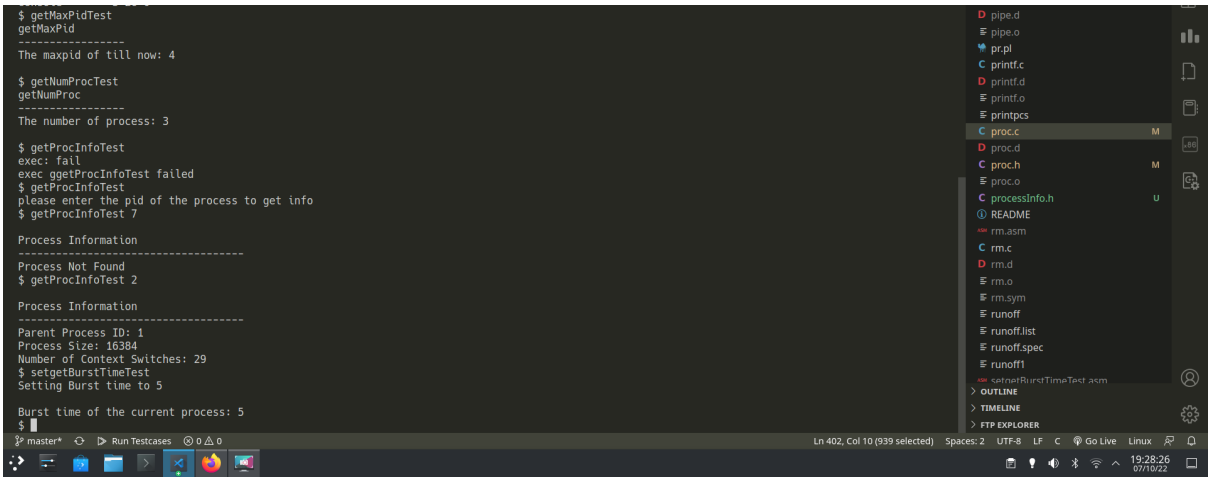
3. **GetProcInfo** - This has been implemented in the function getProcInfoStruct(). Pid is given as an argument, so we look through the ptable to find the correct process and retrieve the information stored in the procInfo struct associated with that process. This procInfo struct includes the pid, the size and the number of context switches of the process.

```c
int getProcInfoStruct(int pid, struct processInfo *processInfo)
{
  acquire(&ptable.lock);
  int chala = 0;
  for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {

    if (p->pid == pid)
    {

      processInfo->ppid = pid;
      processInfo->psize = p->sz;
      processInfo->numberContextSwitches = p->numContextSwitches;
      // processInfo->burstTime=0;
      chala = 1;

      break;
    }
  }

  if (!chala)
  {
    release(&ptable.lock);
    return -1;
  }
  release(&ptable.lock);
  return 0;
}
```

4. **Set_burst_time** - This has been implemented in the function set_burst_time(). It simply takes the burst time as input and sets it as the burst time of the current process.
5. **Get_burst_time** - This has been implemented in the function get_burst_time(). This retrieves the burst time of the current process.

```c
int set_burst_time(int n)
{
  acquire(&ptable.lock);

  struct proc *currp = myproc();

  if (n < 1)
  {
    release(&ptable.lock);
    return -1;
  }
  currp->burstTime = n;

  release(&ptable.lock);
  yield();
  return 0;
}

int get_burst_time()
{
  acquire(&ptable.lock);

  struct proc *currp = myproc();
  release(&ptable.lock);
  return currp->burstTime;
}
```

## OUTPUT OF PART A WITH HANDLING OF FAILURE CASES




The above screenshot shows the output of all the functions implemented in part A.
As shown above the following commands are executed:

1. **getNumProcTest** - prints out number of processes currently in use
2. **getProcInfoTest** - this requires the pid of the required process as input. Since it is not given the first time it shows an error and the relevant message. After that it prints the information as stored in the procInfo struct.
3. **setgetBurstTime** - the burst time is also given in this command, it sets the burst time of the currently running process and outputs that as well.

---

# Part B

## Shortest Job First

Here, we had to replace the default round robin scheduler with the shortest job first scheduler. For this, we had to

1. make changes in the scheduler function to schedule the process with the shortest burst time first.
2. stop the preemption in the trap.c file as there will be no context switches in the case of the shortest job first.
3. set the number of cpu to 1(in param.h file)  as usage of multiple cpus would have exempted us from showing the shortest job first scheduling, as other cpus would have interfered by selecting a new process with sjf to run, thus running multiple processes in parallel.
4. For testing, as we want to put the child process in the ready queue(RUNNABLE state), we put yield() at the end of the set_burst_time() function so that just after the child process is created, we put it in the ready queue.

## Changes in the files:

### proc.c

**scheduler():**
Here, in order to implement the shortest job first scheduler, we first acquire the lock before accessing the ptable so that no new processes get added to the ptable while we are finding the process with the shortest burst time.

Then, we loop through all the processes in the ptable which are in the ready queue(i.e. in the RUNNABLE state). Then, we compare the burst time of the selected process with the burst time of the **shortest_job** process (initialised to the 1st RUNNABLE process in the ptable). This way, we get the shortest_job process after traversing the ptable once, thus ending up with a time complexity of **O(n)**.

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

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);

  struct proc *shortest_job = 0;

  // Find the job with least burst time
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state != RUNNABLE)
      continue;

    if (!shortest_job)
    {
      shortest_job = p;
    }
    else
    {
      if (p->burstTime < shortest_job->burstTime)
      {
        shortest_job = p;
      }
    }
  }
```

Then, for the case where we find a process with the smallest burst time (i.e. when the ready queue is not empty), we hand over the process to the cpu via context switching.
As context switching can only happen for kernel processes, we first need to switch the process to the kernel process. For this, we use the **switchuvm(p)** function which switches the user process p to a kernel process. Then, we set the state of this process to RUNNING as it is going to be executed, and increment the number of **context switches**, as it is going to preempt the process currently running by the cpu. Then, we context switch this process with any other process which the cpu might be running via the **switch(&(c->scheduler), p->context)** function. Then, after this context switching, we switch this kernel process back to a user process via the **switchkvm()** function. The process runs on the cpu and then, cpu->proc is set to 0 showing that the cpu is free now. Finally, we release the lock on the ptable for new processes to enter in case there are any.

```
if (shortest_job)
{
  p = shortest_job;
  // cprintf("BT%d \n", p->burst);
  //  Switch to chosen process.  It is the process's job
  //  to release ptable.lock and then reacquire it
  //  before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
  p->numContextSwitches += 1;

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

  // increment number of context switches
  // p->contextswitches = p->contextswitches + 1;
  switchkvm();

  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
}
release(&ptable.lock);
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

## trap.c

Here, in the case of shortest job first scheduling, the process with the shortest burst time completes first and there cannot be any context switches or preemption when the process is running. Therefore, we need not context switch after a specified time quantum(default in the case of the original round robin scheduler). Therefore, we comment out the following lines

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

// As there will be no context switches in case of sjf scheduling.

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

These were responsible for pre-empting the current process in the cpu with the next scheduled process after a timer interrupt.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

## param.h

Here, as mentioned above, we need to set the number of cpus to 1

```
1   #define NPROC        64  // maximum number of processes
2   #define KSTACKSIZE 4096  // size of per-process kernel stack
3   #define NCPU          1  // maximum number of CPUs
4   #define NOFILE       16  // open files per process
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

## Testing

For testing we have written a user-space test program test_scheduler1. It contains 2 test cases to check the working of the scheduler. This test_schedluer1 accepts n(number of processes) as the argument. We have written **3 test cases** as follows:

### Test 1

We fork the parent process to create n number of children in a for loop. Then, we set the burst times of these processes as the burst times in the t array. Note, that in the set_burst_time function itself we yield the process i.e. out it in the ready queue. Then, we delay the child process for a time which is proportional to the assigned burst time of the process. This is done to prevent the child process from completing execution before all the child processes hereby created enter the ready queue.

Here, we make half of the processes **CPU bound** and the other half as IO bound. The only difference in these processes is that the **IO bound** processes must take longer than the CPU bound processes, therefore, we put the IO bound processes to sleep for 1ms.

```
// CPU bound process
if (i % 2 == 0)
{
    int *data = (int *)malloc(sizeof(int) * 10240);
    if (data <= 0)
        printf(1, "Memory not allocated due to some error\n");

    for (int i4 = 0; i4 < t[i] * 1000; i4++)
    {
        // for (int k = 0; k < 570; k++)
        for (int j = 0; j < 10240; j++)
            data[j]++;
        // printf(1, "\n-->%d %d\n", i,t[i]);
    }
    printf(1, "CPU Bound  ");
}
```

```
// IO bound process
else
{
    // mimicking IO wait
    for (int i2 = 0; i2 < t[i] * 10; i2++)
    {
        sleep(1);
    }
    printf(1, "IO Bound   ");
}
```

Then, when all these processes enter the ready queue, the sjf scheduler schedules them and the processes run according to the sjf algorithm. This can be clearly seen from the output:

```
---------------------------Test 1 ---------------------------------------
Process Type      Burst Time      Context Switches       PID
CPU Bound              10                1                6
CPU Bound              20                1                10
CPU Bound              40                1                4
CPU Bound              60                1                8
CPU Bound              100               1                12
IO Bound               30                301              9
IO Bound               50                501              13
IO Bound               70                701              5
IO Bound               80                801              11
IO Bound               90                901              7
```

Comparing this from the default round robin scheduler, below,

```
---------------------------Test 1 ---------------------------------------
Process Type      Burst Time      Context Switches       PID
CPU Bound              40                2                4
CPU Bound              10                2                6
CPU Bound              60                2                8
CPU Bound              20                2                10
CPU Bound              100               2                12
IO Bound               30                302              9
IO Bound               50                502              13
IO Bound               70                702              5
IO Bound               80                802              11
IO Bound               90                902              7
```

we can clearly see that in the case of sjf, the processes complete in the ascending order of burst time.

Here, we can notice that the number of context switches for CPU bound processes is 1 which shows the non - preemptive behaviour of sjf algorithm. Here, the 1st context switch occurs when the process is brought to the cpu for running. After that, there are no further context switches as the next process will be taken for running only when the current one completes execution.

Here, the IO processes complete execution after the CPU bound processes. This is because the IO processes sleep before execution and therefore, they come out of the ready queue and therefore, aren't scheduled along with the CPU processes. Then, by the time they come out of the sleep mode, all the CPU bound processes have completed their execution and thus only the IO bound processes are scheduled according to the hybrid scheduling algorithm.

### Test 2

In this test, we had put the processes in the ready queue in the descending order of the burst times. But, from the output of the sjf scheduler, we can clearly see that the processes complete in the ascending order of burst times, thus abiding by the sjf algorithm.

output for sjf scheduling:

```
-------------------------Test 2 ------------------------------------
Process Type      Burst Time      Context Switches      PID
CPU Bound             22                1         22
CPU Bound             24                1         20
CPU Bound             26                1         18
CPU Bound             28                1         16
CPU Bound             30                1         14
IO Bound              21                211         23
IO Bound              23                231         21
IO Bound              25                251         19
IO Bound              27                271         17
IO Bound              29                291         15
```

output for default round robin scheduling:

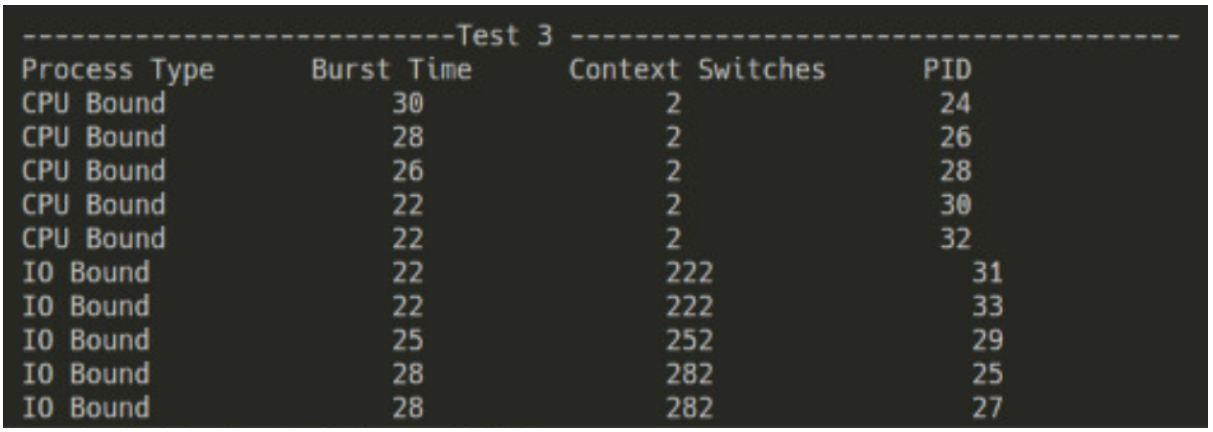
```
-------------------------Test 2 ------------------------------------
Process Type      Burst Time      Context Switches      PID
CPU Bound             30                2         14
CPU Bound             28                2         16
CPU Bound             26                2         18
CPU Bound             24                2         20
CPU Bound             22                2         22
IO Bound              21                212         23
IO Bound              23                232         21
IO Bound              25                252         19
IO Bound              27                272         17
IO Bound              29                292         15
```

## Test 3

In this test case we show that the scheduler does strict sjf scheduling. To show this, we made this test case such that 2 child processes have the same burst time. In this case we can clearly see from the output that the process which came first in the ready queue, i.e. the process with lower pid completes first.
output for sjf scheduling:

```
-------------------------Test 3 ------------------------------------
Process Type      Burst Time      Context Switches      PID
CPU Bound             22                1                30
CPU Bound             22                1                32
CPU Bound             26                1                28
CPU Bound             28                1                26
CPU Bound             30                1                24
IO Bound              22                221                31
IO Bound              22                221                33
IO Bound              25                251                29
IO Bound              28                281                25
IO Bound              28                281                27
```

default output for round robin:



```
-------------------------Test 3 ---------------------------------------
Process Type      Burst Time      Context Switches      PID
CPU Bound              30                2                24
CPU Bound              28                2                26
CPU Bound              26                2                28
CPU Bound              22                2                30
CPU Bound              22                2                32
IO Bound               22                222                31
IO Bound               22                222                33
IO Bound               25                252                29
IO Bound               28                282                25
IO Bound               28                282                27
```

==========

## Hybrid Scheduler

Here, we had to implement a round robin scheduler with the shortest job first where the time quantum for which the processes will be running is the burst time of the process with the smallest burst time in the very first round and hence remains constant for the rest of the rounds. In each round we schedule the processes according to the shortest job first scheduling.

As the sjf scheduling was **non - preemptive**, we had disabled the timer interrupts. Here, in the case of round robin, we require interrupts and therefore, we had to uncomment the previously commented lines in trap.c file and modify it so that the time quantum is set as the minimum burst time of all the processes initially.

## Changes in the files:

### proc.c

**scheduler()**
Here, to implement the hybrid scheduler, we first acquire a lock for the ptable and then, get all the RUNNABLE processes in the ready queue.

```
413     // Set up Ready Queue
414     struct proc *RQ[NPROC];
415
416     int k = 0;
417
418     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
419     {
420       if (p->state == RUNNABLE)
421       {
422         RQ[k++] = p;
423       }
424     }
```

Then, we sort the ready queue in the ascending order of the burst times of the processes using merge sort. For this, we have created a function for implementing merge sort on the array of the RUNNABLE processes in the ready queue.

```
385   void mergeSort(struct proc *RQ[], int l, int r)
386   {
387     if (l < r)
388     {
389       // Same as (l+r)/2, but avoids overflow for
390       // large l and h
391       int m = l + (r - l) / 2;
392
393       // Sort first and second halves
394       mergeSort(RQ, l, m);
395       mergeSort(RQ, m + 1, r);
396
397       merge(RQ, l, m, r);
398     }
399   }
```

Then, we run the processes in the ready queue one by one in the sorted order, in the similar manner as that in the case of sjf scheduling by switching the user process to kernel process for context switching and then switching it back to the user process.

```
429       // Find the job with least burstTime time
430       for (int i = 0; i < k; i++)
431       {
432         p = RQ[i];
433         if (p->state == RUNNABLE)
434         {
435           // cprintf("BT %d \n", i);
436           // Switch to chosen process.  It is the process's job
437           // to release ptable.lock and then reacquire it
438           // before jumping back to us.
439           c->proc = p;
440           switchuvm(p);
441           p->state = RUNNING;
442
443           swtch(&(c->scheduler), p->context);
444
445           // increment number of context switches
446           p->numContextSwitches = p->numContextSwitches + 1;
447           switchkvm();
448
449           // Process is done running for now.
450           // It should have changed its p->state before coming back.
451           c->proc = 0;
452         }
453       }
454       release(&ptable.lock);
```

Here, as we have to set the time quanta as the minimum burst time of the processes, we set the time quanta in the set_burst_time as the minimum of the burst time of all the processes whenever we are setting the burst time of a process. Here the implementation tweak is that as we do not change the value of the burst time of the processes and therefore, the time quanta will always be the minimum burst time.

```
851   int set_burst_time(int n)
852   {
853     acquire(&ptable.lock);
854
855     struct proc *currp = myproc();
856
857     if (n < 1)
858     {
859       release(&ptable.lock);
860       return -1;
861     }
862     currp->burstTime = n;
863     if (n < TimeQuanta)
864       TimeQuanta = n;
865
866     release(&ptable.lock);
867     yield();
868     return 0;
869   }
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

## trap.c

In the trap.c file, for the processes in the RUNNING state, we add 1 to time_slice for that process for each timer interrupt. We yield the process only when the time_slice equals the time quanta. Here, this time_slice is a new field added in the proc structure and is initialised to 0. It is again set to 0 once the process runs for the required time quanta.

```
135     if (myproc() && myproc()->state == RUNNING &&
136         tf->trapno == T_IRQ0 + IRQ_TIMER)
137     {
138       struct proc *p = myproc();
139
140       // acquire(&ptable.lock);
141       p->time_slice += 1;
142
143       int n = p->time_slice;
144       // release(&ptable.lock);
145
146       if (n % TimeQuanta == 0)
147       {
148         yield(); // one more Time Quanta is complete
149       }
150       // return 0;
151     }
```

The **yield()** function first locks the ptable and then makes the current process RUNNABLE, then it calls the **sched()** function which is responsible for switching the context of the process to that of the scheduler. After this, it releases the lock.

```
// Give up the CPU for one scheduling round.
void yield(void)
{
  acquire(&ptable.lock); // DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

## Testing

We are using the same test_scheduler as that for the sjf scheduling. The output clearly shows the proper execution. Here, the IO processes complete execution after the CPU

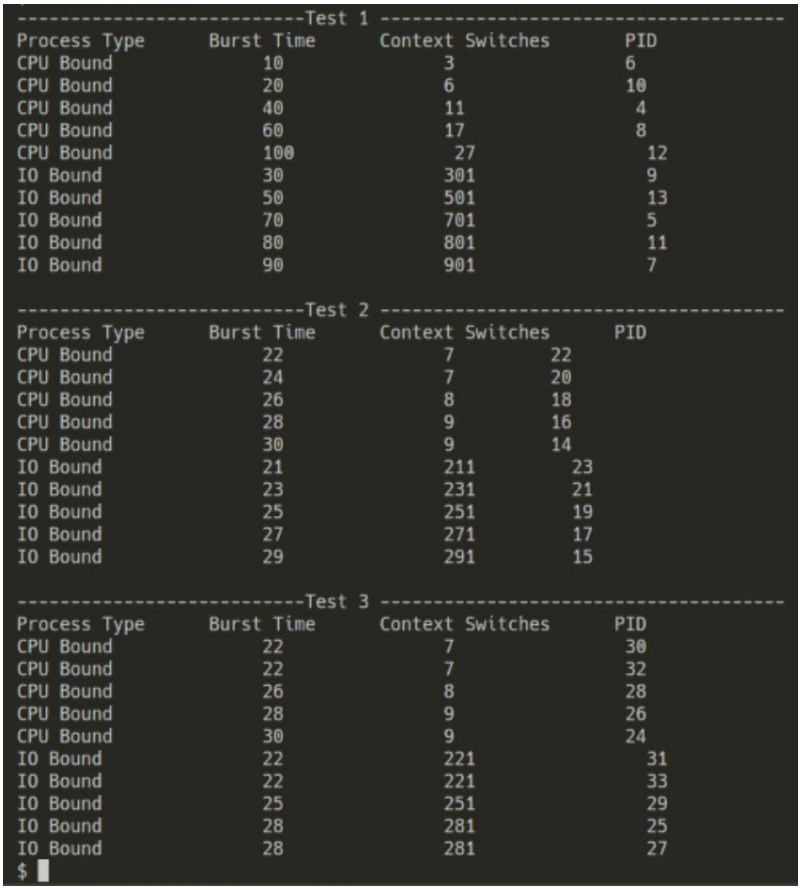
bound processes. This is because the IO processes sleep before execution and therefore, they come out of the ready queue and therefore, aren't scheduled along with the CPU processes. Then, by the time they come out of the sleep mode, all the CPU bound processes have completed their execution and thus only the IO bound processes are scheduled according to the hybrid scheduling algorithm.

In the first test case, the time quanta is 10 time units as it is the minimum burst time among all the processes. The sequence in which the processes complete execution is the same as that of the sjf scheduling. The only difference here is that the processes get to run for some time and therefore does not let the larger processes to starve for longer. Also, it decreases the average waiting time and average turn-around time.

Comparing the outputs of this hybrid scheduler with that of the default round robin scheduler, we can see that in the later, the processes complete in any order, whereas in the case of hybrid scheduler, the processes complete in the order same as that of sjf.

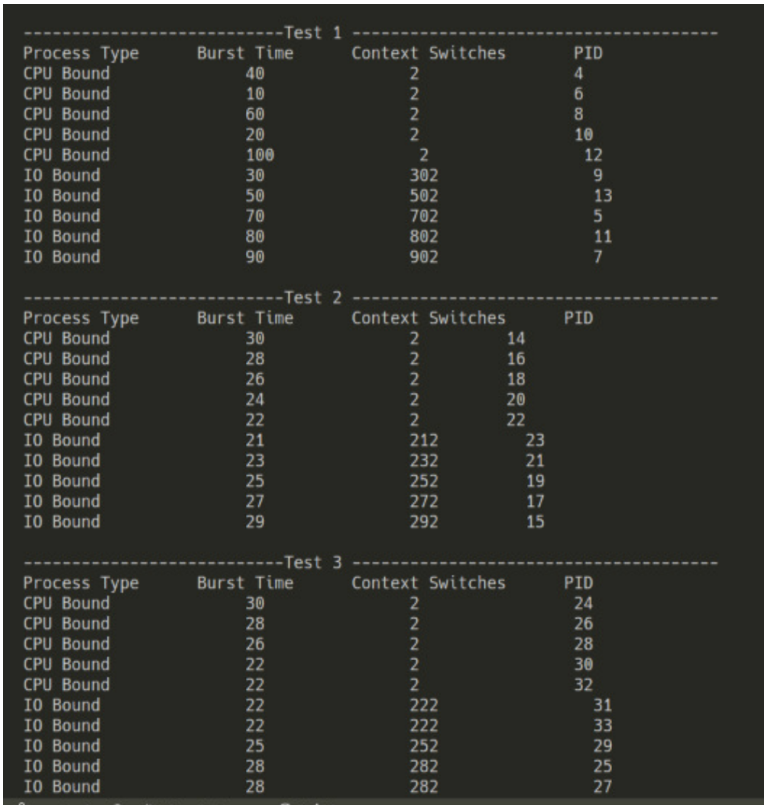
Note, here the context switches for the CPU bound as well as IO bound processes are higher (>1) which shows that round robin was functional.

Output for hybrid round robin scheduler:



```
-------------------------Test 1 -------------------------------------
Process Type      Burst Time      Context Switches       PID
CPU Bound              10               3                6
CPU Bound              20               6                10
CPU Bound              40               11                4
CPU Bound              60               17                8
CPU Bound              100               27                12
IO Bound               30               301               9
IO Bound               50               501               13
IO Bound               70               701               5
IO Bound               80               801               11
IO Bound               90               901               7

-------------------------Test 2 -------------------------------------
Process Type      Burst Time      Context Switches      PID
CPU Bound              22               7         22
CPU Bound              24               7         20
CPU Bound              26               8         18
CPU Bound              28               9         16
CPU Bound              30               9         14
IO Bound               21               211         23
IO Bound               23               231         21
IO Bound               25               251         19
IO Bound               27               271         17
IO Bound               29               291         15

-------------------------Test 3 -------------------------------------
Process Type      Burst Time      Context Switches      PID
CPU Bound              22               7                30
CPU Bound              22               7                32
CPU Bound              26               8                28
CPU Bound              28               9                26
CPU Bound              30               9                24
IO Bound               22               221               31
IO Bound               22               221               33
IO Bound               25               251               29
IO Bound               28               281               25
IO Bound               28               281               27
$
```

Output for default round robin scheduler:



```
-------------------------Test 1 -------------------------------------
Process Type      Burst Time      Context Switches       PID
CPU Bound              40               2               4
CPU Bound              10               2               6
CPU Bound              60               2               8
CPU Bound              20               2               10
CPU Bound              100               2               12
IO Bound               30               302               9
IO Bound               50               502               13
IO Bound               70               702               5
IO Bound               80               802               11
IO Bound               90               902               7

-------------------------Test 2 -------------------------------------
Process Type      Burst Time      Context Switches      PID
CPU Bound              30               2         14
CPU Bound              28               2         16
CPU Bound              26               2         18
CPU Bound              24               2         20
CPU Bound              22               2         22
IO Bound               21               212         23
IO Bound               23               232         21
IO Bound               25               252         19
IO Bound               27               272         17
IO Bound               29               292         15

-------------------------Test 3 -------------------------------------
Process Type      Burst Time      Context Switches      PID
CPU Bound              30               2               24
CPU Bound              28               2               26
CPU Bound              26               2               28
CPU Bound              22               2               30
CPU Bound              22               2               32
IO Bound               22               222               31
IO Bound               22               222               33
IO Bound               25               252               29
IO Bound               28               282               25
IO Bound               28               282               27
```

---