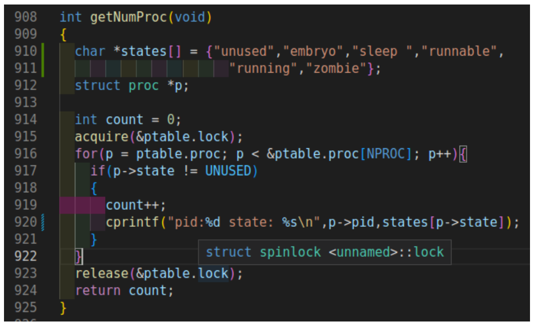
OS assignment 2

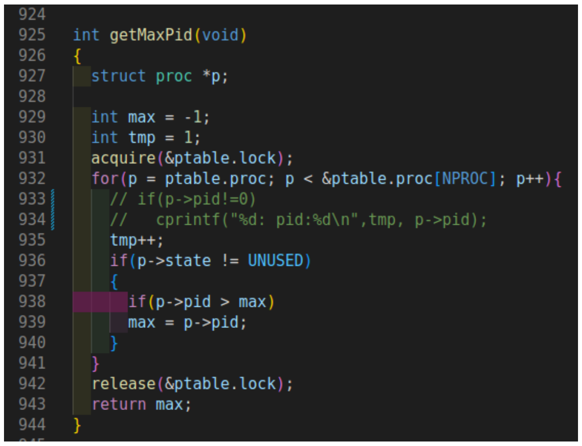
Pranshu Shah-200101085
Vedant Shah-200101115
Dhruv Shah-200101124

Part A

1) For the system calls **getNumProc()** and **getMaxPid()**, we create user programs to access them and create system functions. Firstly we assign integers to functions in **syscall.h.** Then we declare and place our functions in the array in **syscall.c** so that addressing for our function can be done through appropriate indexing. We implement these functions in **sysproc.c.** Logic for the implementation is to traverse the ptable and search for the required result. So that can be done if we have an instance of ptable. That can be accessed from **proc.c**, so we make the corresponding functions there. Also locks must be placed to ensure that results are consistent.(see acquiring and releasing of the ptable locks).



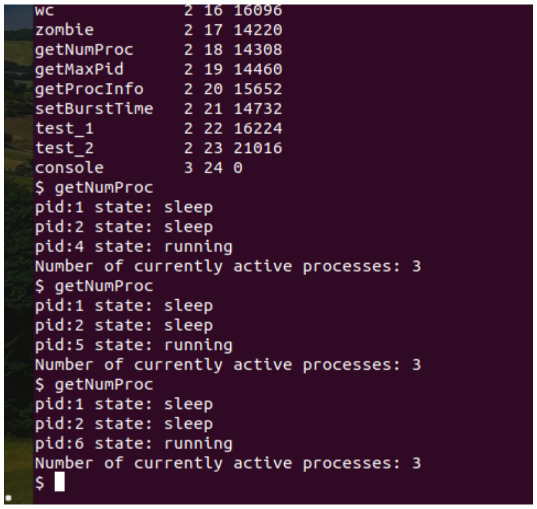
```c
int getNumProc(void)
{
    char *states[] = {"unused","embryo","sleep ","runnable",
                      "running","zombie"};
    struct proc *p;

    int count = 0;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != UNUSED)
        {
            count++;
            cprintf("pid:%d state: %s\n",p->pid,states[p->state]);
        }
    }
    release(&ptable.lock);
    return count;
}
```



```c
int getMaxPid(void)
{
    struct proc *p;

    int max = -1;
    int tmp = 1;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        // if(p->pid!=0)
        //   cprintf("%d: pid:%d\n",tmp, p->pid);
        tmp++;
        if(p->state != UNUSED)
        {
            if(p->pid > max)
            max = p->pid;
        }
    }
    release(&ptable.lock);
    return max;
}
```

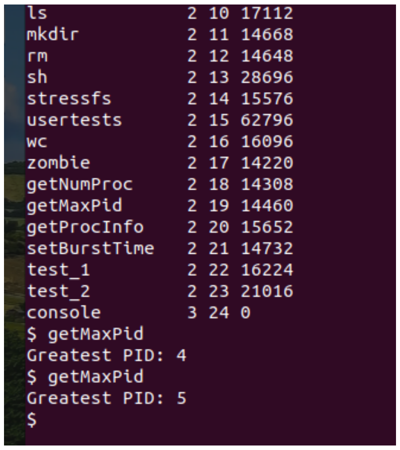
We need to add/declare these functions in the defs.h file to let the kernel access these functions.
The output of the functions is:

a) **GetNumProc()** : It returns the number of active processes in the ptable. Also with them, we have additionally printed the pids and the status of processes.
We can see a pattern in the pids. Pid 1 and 2 are system assigned processes, while after that the process that we run are there. So first ls command was run which had pid 3. Subsequently getNumProc() is run different times and we can see that it is assigned pid serially.



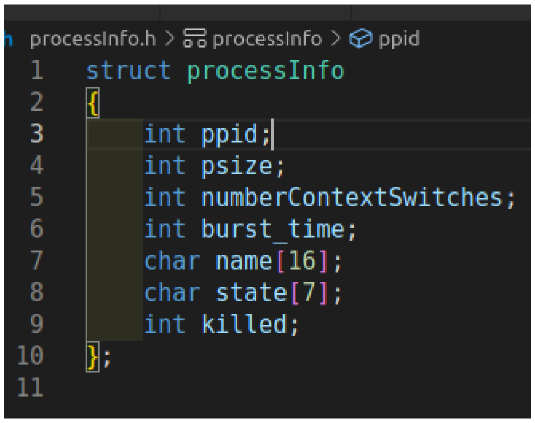
```
wc             2 16 16096
zombie         2 17 14220
getNumProc     2 18 14308
getMaxPid      2 19 14460
getProcInfo    2 20 15652
setBurstTime   2 21 14732
test_1         2 22 16224
test_2         2 23 21016
console        3 24 0
$ getNumProc
pid:1 state: sleep
pid:2 state: sleep
pid:4 state: running
Number of currently active processes: 3
$ getNumProc
pid:1 state: sleep
pid:2 state: sleep
pid:5 state: running
Number of currently active processes: 3
$ getNumProc
pid:1 state: sleep
pid:2 state: sleep
pid:6 state: running
Number of currently active processes: 3
$
```

b) **GetMaxPid()** : This prints the max pid present in the ptable.



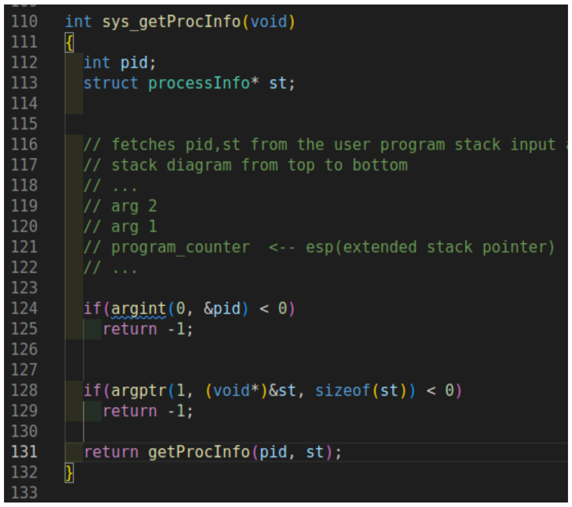
```
ls             2 10 17112
mkdir          2 11 14668
rm             2 12 14648
sh             2 13 28696
stressfs       2 14 15576
usertests      2 15 62796
wc             2 16 16096
zombie         2 17 14220
getNumProc     2 18 14308
getMaxPid      2 19 14460
getProcInfo    2 20 15652
setBurstTime   2 21 14732
test_1         2 22 16224
test_2         2 23 21016
console        3 24 0
$ getMaxPid
Greatest PID: 4
$ getMaxPid
Greatest PID: 5
$
```

2) **getProcInfo()** : This takes in arguments as pids and returns corresponding process info. This is done similarly as the above one(system call implementation). Additionally to supply arguments to system calls argint is used which is already present in xv6. Our processInfo is a struct which contains following information. We iterate through the ptable and find the process and then copy the required contents in processInfo struct.
We can see that 1 and 2 pids are system processes,ie init and shell correspondingly.
Apart from the given properties, we have also printed process state killed status and burst times.



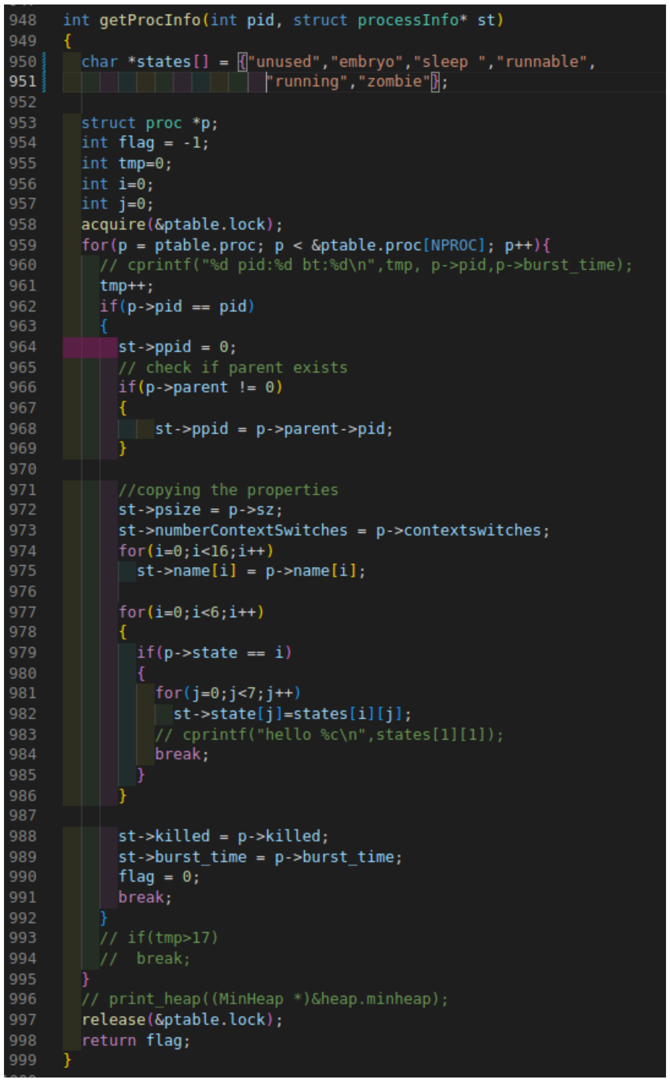
```c
struct processInfo
{
    int ppid;
    int psize;
    int numberContextSwitches;
    int burst_time;
    char name[16];
    char state[7];
    int killed;
};
```

processInfo.h struct



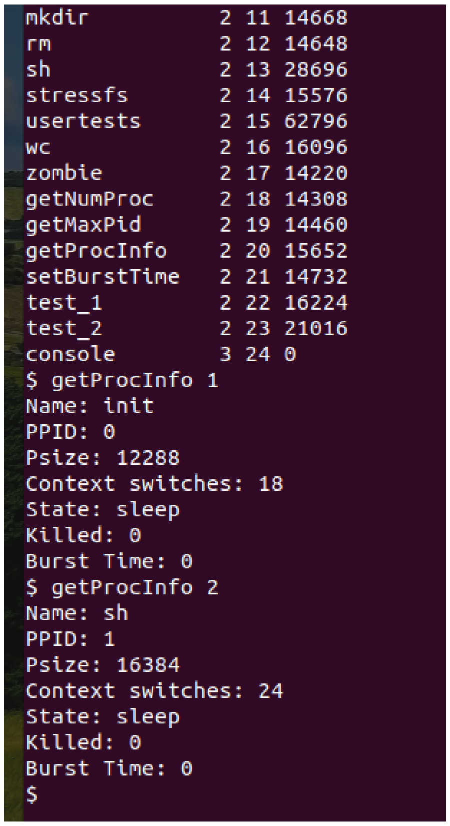
```c
int sys_getProcInfo(void)
{
  int pid;
  struct processInfo* st;


  // fetches pid,st from the user program stack input
  // stack diagram from top to bottom
  // ...
  // arg 2
  // arg 1
  // program_counter  <-- esp(extended stack pointer)
  // ...

  if(argint(0, &pid) < 0)
    return -1;


  if(argptr(1, (void*)&st, sizeof(st)) < 0)
    return -1;

  return getProcInfo(pid, st);
}
```

sysproc.c code



```c
int getProcInfo(int pid, struct processInfo* st)
{
  char *states[] = {"unused","embryo","sleep ","runnable",
                    "running","zombie"};

  struct proc *p;
  int flag = -1;
  int tmp=0;
  int i=0;
  int j=0;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    // cprintf("%d pid:%d bt:%d\n",tmp, p->pid,p->burst_time);
    tmp++;
    if(p->pid == pid)
    {
      st->ppid = 0;
      // check if parent exists
      if(p->parent != 0)
      {
        st->ppid = p->parent->pid;
      }

      //copying the properties
      st->psize = p->sz;
      st->numberContextSwitches = p->contextswitches;
      for(i=0;i<16;i++)
        st->name[i] = p->name[i];

      for(i=0;i<6;i++)
      {
        if(p->state == i)
        {
          for(j=0;j<7;j++)
            st->state[j]=states[i][j];
          // cprintf("hello %c\n",states[1][1]);
          break;
        }
      }

      st->killed = p->killed;
      st->burst_time = p->burst_time;
      flag = 0;
      break;
    }
    // if(tmp>17)
    //  break;
  }
  // print_heap((MinHeap *)&heap.minheap);
  release(&ptable.lock);
  return flag;
}
```

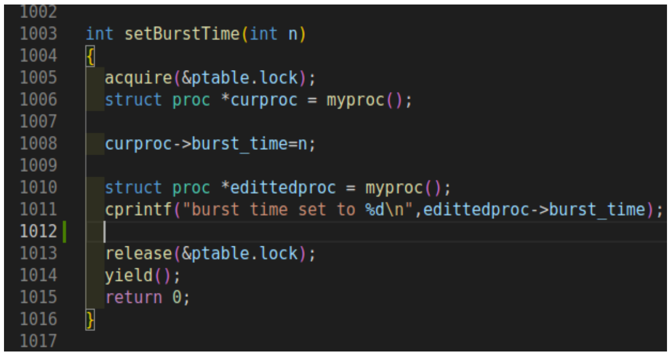
Proc.c code



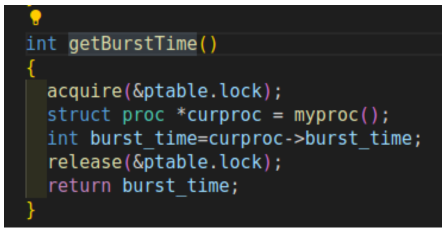
```
mkdir          2 11 14668
rm             2 12 14648
sh             2 13 28696
stressfs       2 14 15576
usertests      2 15 62796
wc             2 16 16096
zombie         2 17 14220
getNumProc     2 18 14308
getMaxPid      2 19 14460
getProcInfo    2 20 15652
setBurstTime   2 21 14732
test_1         2 22 16224
test_2         2 23 21016
console        3 24 0
$ getProcInfo 1
Name: init
PPID: 0
Psize: 12288
Context switches: 18
State: sleep
Killed: 0
Burst Time: 0
$ getProcInfo 2
Name: sh
PPID: 1
Psize: 16384
Context switches: 24
State: sleep
Killed: 0
Burst Time: 0
$
```

output

3) For this part we need to include an additional attribute to the process, burst time. Thus we modify proc.h. We implement 2 system calls **setBurstTime()** and **getBurstTime()**. We also have placed the ptable locks to handle the edge cases.
Initially burst time is set to 0 in allocproc(). We need to access the ptable to modify the processes, so we need the implementation in proc.c files. Sysproc.c redirects us to here. setBurstTime() takes an integer argument using argint().
We use myproc(), which gives us pointer to the current running process. Using it, we modify the burst time.



```
int setBurstTime(int n)
{
  acquire(&ptable.lock);
  struct proc *curproc = myproc();

  curproc->burst_time=n;

  struct proc *edittedproc = myproc();
  cprintf("burst time set to %d\n",edittedproc->burst_time);

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



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

We use **yield()** function to skip scheduling until one round. This is done so that all processes arriving at a single time are analyzed and correctly scheduled.

getBurstTime function is similar, and just returns the burst time value.

In overall the files we changed for the above part are
Syscall.h, syscall.c, sysproc.c, proc.h, proc.c, defs.h, usys.S, user.h, Makefile.

Also new user files were made for user calls to system calls
getNumProc.c, getMaxPid.c, getProcInfo.c, setBurstTime.c

Part B

Firstly change the number of CPUs in params to 1, to observe correct results.
We have a general pattern for implementing both the schedulers.

I have used **minheap** data structure. It's struct is made in proc.h containing a proc*array and an integer size.

There is another struct containing a lock and this minheap. Two instances of it are made in proc.c named heap and compheap.

This minheap orders the elements/processes in its array in such a way that the minimum burst time process is always the first element in the array.

Supporting functions for minheap are implemented in proc.c. They are:

1) insert_minheap: inserting element in minheap
2) heapify: balancing heap after insertion
3) delete_minimum: deleting minimum element from the heap
4) delete_element: deleting element at a particular index
5) IsEmpty: checking whether heap is empty
6) IsFull: checking whether heap is full
7) print_heap: printing all the elements in the heap
8) right_child: index for right child left_child: index for left child
9) parent: index of parent
10) get_min: returns minimum burst time in heap.

"heap" denotes ready queue.
"compheap" represents a hold queue where processes wait for CPUs after completing their time slice.

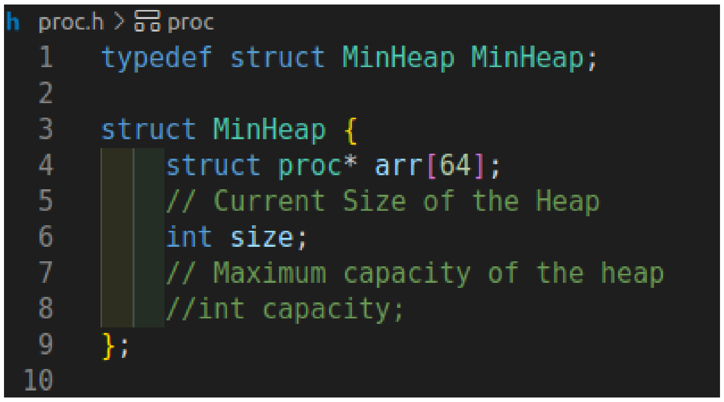
Basic logic for scheduling is:
All the processes which become runnable are put into the ready queue. When their time slice expires they call the yield function which calls sched and transfers control to scheduler. Together with this the process is removed from the ready queue.
The scheduler then schedules another process seeing the ready queue.

The processes which are preempted due to interrupts and are not runnable, such processes are emptied from both the queues,
And ones which are runnable are placed in hold queue, which will be scheduled again in the next round

Files modified: proc.c, trap.c, proc.h

Attributes added in process structure:

1) **Struct MinHeap** is added



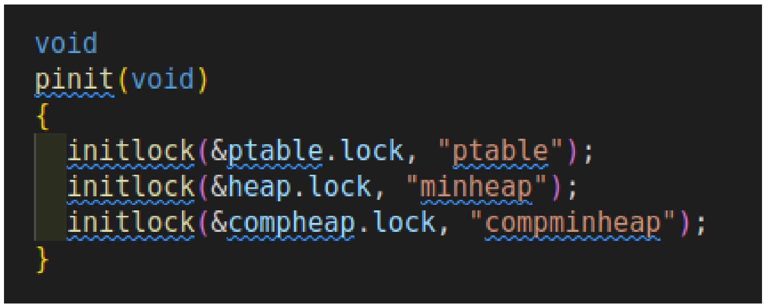
```c
typedef struct MinHeap MinHeap;

struct MinHeap {
    struct proc* arr[64];
    // Current Size of the Heap
    int size;
    // Maximum capacity of the heap
    //int capacity;
};
```

2) **time_quanta**,**time_elapsed** added in struct proc. Time_quanta is quanta of time after which process will be preempted, and time_elapsed is the total cpu time alloted to that proc

Changes in proc.c :-

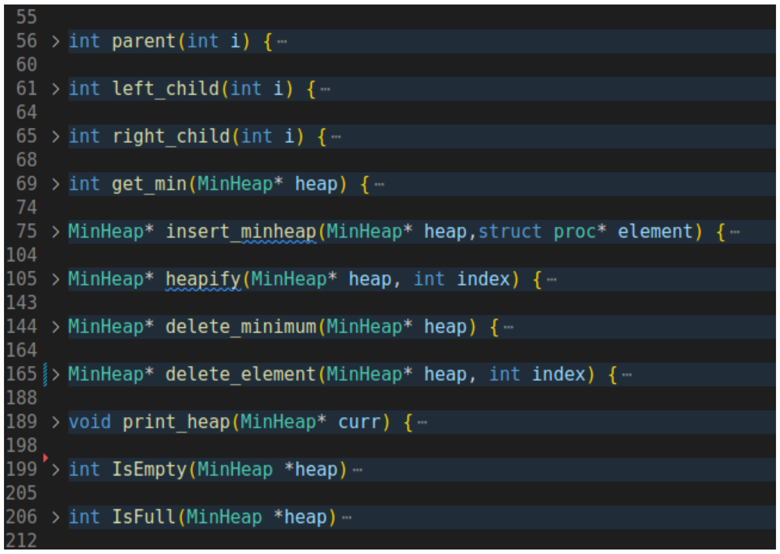
- A quanta named variable is added. It is an integer and decides the time quantum after which the process is preempted.

- **pinit** function is changed and **2 heap locks** are initialized



```
void
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&heap.lock, "minheap");
  initlock(&compheap.lock, "compminheap");
}
```

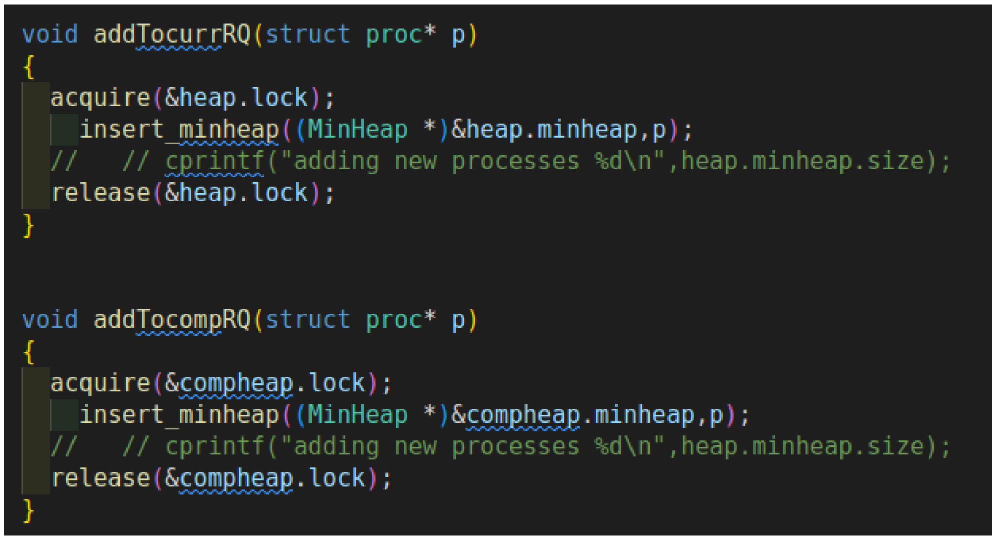
- Various minHeap supporting functions are added here



```
> int parent(int i) {…

> int left_child(int i) {…

> int right_child(int i) {…

> int get_min(MinHeap* heap) {…

> MinHeap* insert_minheap(MinHeap* heap,struct proc* element) {…

> MinHeap* heapify(MinHeap* heap, int index) {…

> MinHeap* delete_minimum(MinHeap* heap) {…

> MinHeap* delete_element(MinHeap* heap, int index) {…

> void print_heap(MinHeap* curr) {…

> int IsEmpty(MinHeap *heap)…

> int IsFull(MinHeap *heap)…
```

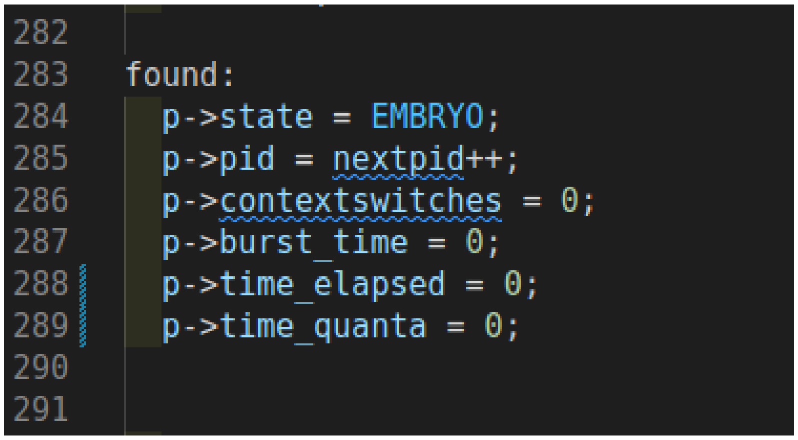
- addTocurrRQ: Calls insert minheap for heap
- addTocompRQ: Calls insert minheap for compheap

Proper locks must be acquired so that anomalies due to insertion and deletion don't arise.



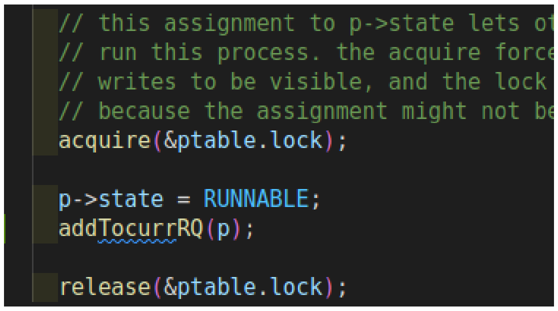
```
void addTocurrRQ(struct proc* p)
{
  acquire(&heap.lock);
    insert_minheap((MinHeap *)&heap.minheap,p);
//    // cprintf("adding new processes %d\n",heap.minheap.size);
  release(&heap.lock);
}


void addTocompRQ(struct proc* p)
{
  acquire(&compheap.lock);
    insert_minheap((MinHeap *)&compheap.minheap,p);
//    // cprintf("adding new processes %d\n",heap.minheap.size);
  release(&compheap.lock);
}
```

- proc variables are initialized in allocproc(), where every new process is alloted pid.



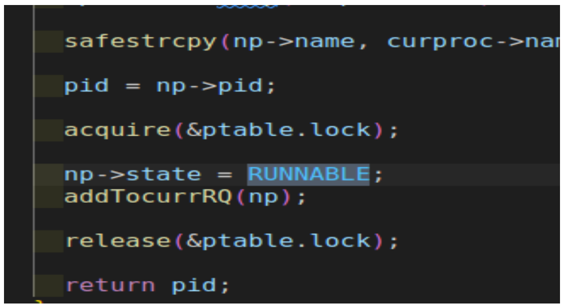
```

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
  p->contextswitches = 0;
  p->burst_time = 0;
  p->time_elapsed = 0;
  p->time_quanta = 0;


```

- Next we check wherever **RUNNABLE** is present, we add it to heap, current ready queue, except yield which passes the process to compheap, as it is called due to expired time. Function where these changes are made are void userinit(void),int fork(void),static void wakeup1(void *chan), int kill(int pid)



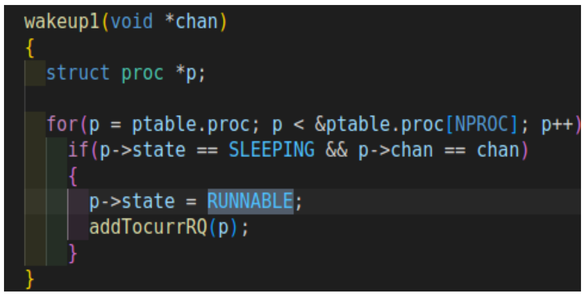
```
// this assignment to p->state lets ot
// run this process. the acquire force
// writes to be visible, and the lock
// because the assignment might not be
acquire(&ptable.lock);

p->state = RUNNABLE;
addTocurrRQ(p);

release(&ptable.lock);
```

userinit()



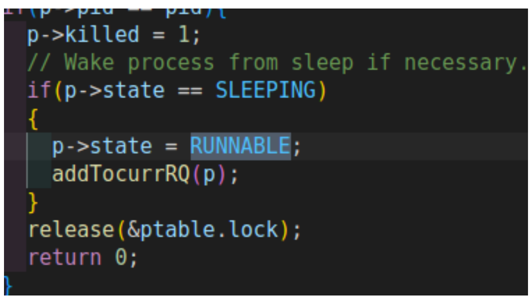
```
safestrcpy(np->name, curproc->nam

pid = np->pid;

acquire(&ptable.lock);

np->state = RUNNABLE;
addTocurrRQ(np);

release(&ptable.lock);

return pid;
```

fork()



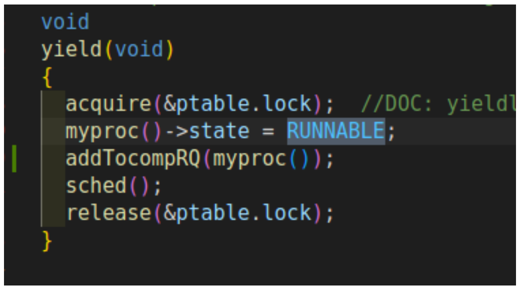
```
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
    {
      p->state = RUNNABLE;
      addTocurrRQ(p);
    }
}
```

wakeup1()



```
if(p->pid == pid){
  p->killed = 1;
  // Wake process from sleep if necessary.
  if(p->state == SLEEPING)
  {
    p->state = RUNNABLE;
    addTocurrRQ(p);
  }
  release(&ptable.lock);
  return 0;
}
```

kill()



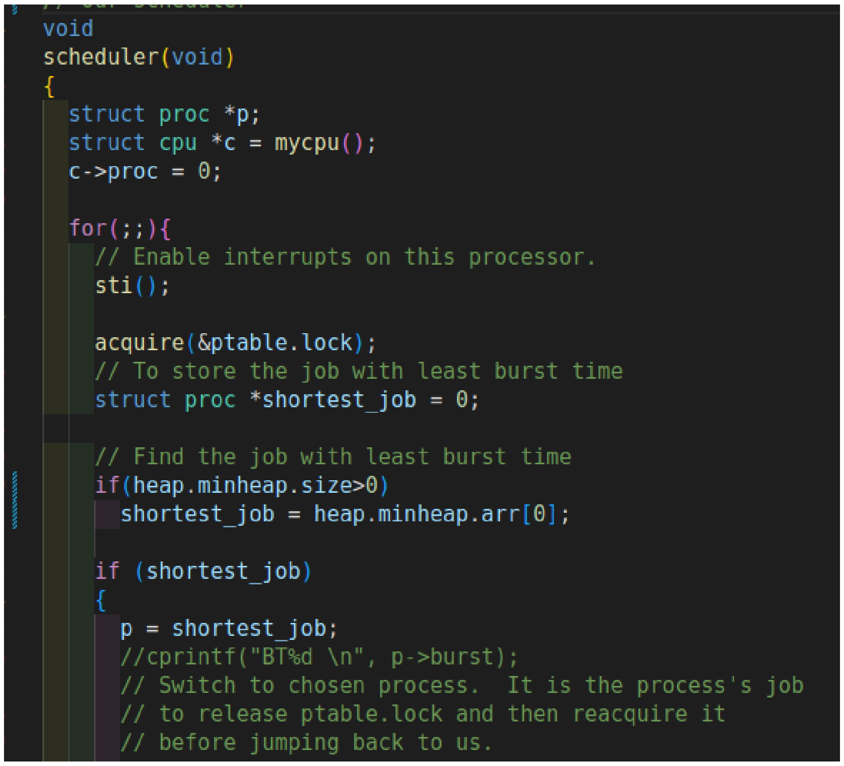
```
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldl
  myproc()->state = RUNNABLE;
  addTocompRQ(myproc());
  sched();
  release(&ptable.lock);
}
```

yield()

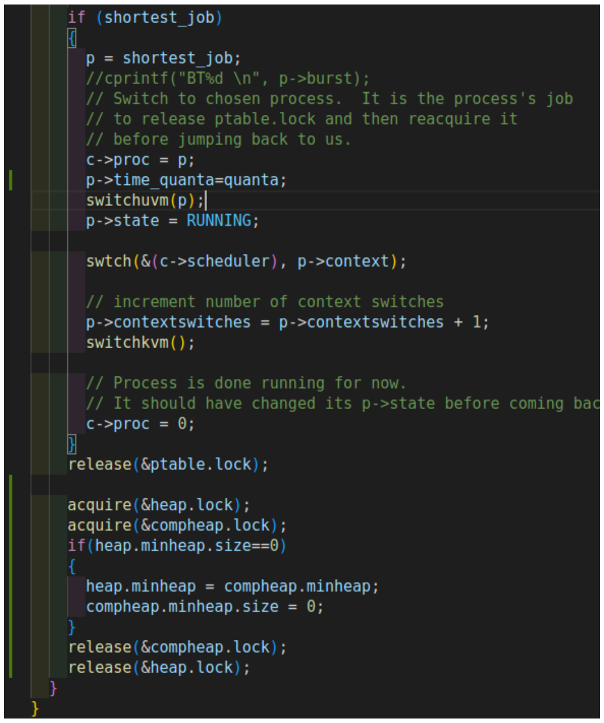
- Next let us look at the scheduler. We get a pointer to our cpu and assign it a null value(0). The outer for loop runs infinitely.
  We then declare a shortest job proc * as null. If our heap isn't empty then a process is assigned to the shortest job.
  It then follows the same step as earlier the scheduler did, allotting cpu this process and calling **switchuvm**,**switchkvm**.
  In between we have assigned it the time quanta to the process(which is determined in setBurstTime(),which we will see later). Also the contextswitches of this process is incremented. At the end it lets us know the number of times the process was switched.(remember the number of times the process exits is the same as the number of times it enters a cpu).
  Finally we release the ptable lock.

If the heap was empty and the shortest job variable remains null then we transfer all the processes in comp heap to heap, and make compheap empty.

The time slice expiry happens due to clock tick interruption.
In the shortest job first scheduling we disable such interrupts(we will see in trap.c) while for hybrid scheduling we enable it.
During SJFS none of the interrupts happen, so none of the process forked go to compheap. CPU bound process finish as they arrive and I/O bound processes enter and exit ready queue(heap as named in code) several times.
While in SJFS + round robin,hybrid scheduling, we enable clock tick interrupts, so the processes then goes to ready and hold queues. There we will observe more contextswitches.



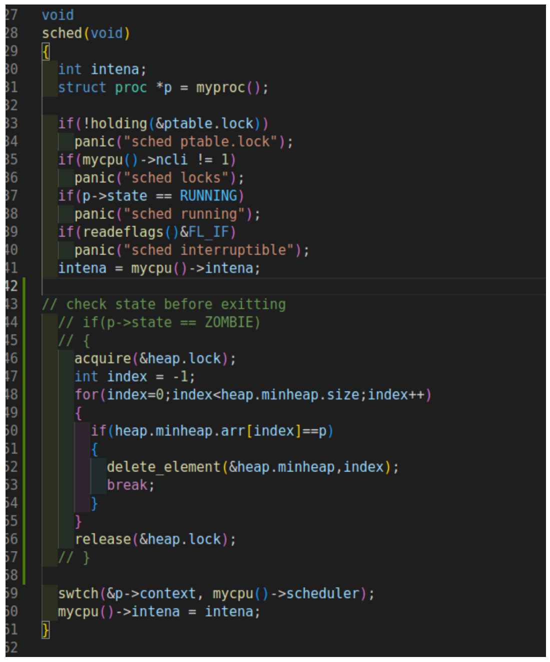
```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

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

    acquire(&ptable.lock);
    // To store the job with least burst time
    struct proc *shortest_job = 0;

    // Find the job with least burst time
    if(heap.minheap.size>0)
      shortest_job = heap.minheap.arr[0];

    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.
```



```
        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;
          p->time_quanta=quanta;
          switchuvm(p);
          p->state = RUNNING;

          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 bac
          c->proc = 0;
        }
        release(&ptable.lock);

        acquire(&heap.lock);
        acquire(&compheap.lock);
        if(heap.minheap.size==0)
        {
          heap.minheap = compheap.minheap;
          compheap.minheap.size = 0;
        }
        release(&compheap.lock);
        release(&heap.lock);
    }
}
```

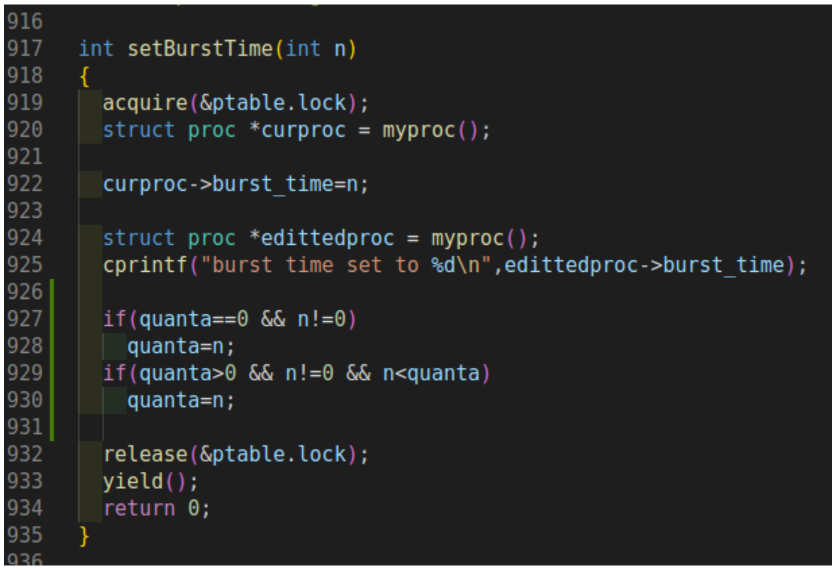
Changes in **sched()**

Sched() is called by other processes to contextswitch from it to scheduler.
Here we remove the process from the ready queue.(We search through the array and then remove that particular element)



```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;

// check state before exitting
  // if(p->state == ZOMBIE)
  // {
    acquire(&heap.lock);
    int index = -1;
    for(index=0;index<heap.minheap.size;index++)
    {
      if(heap.minheap.arr[index]==p)
      {
        delete_element(&heap.minheap,index);
        break;
      }
    }
    release(&heap.lock);
  // }

  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

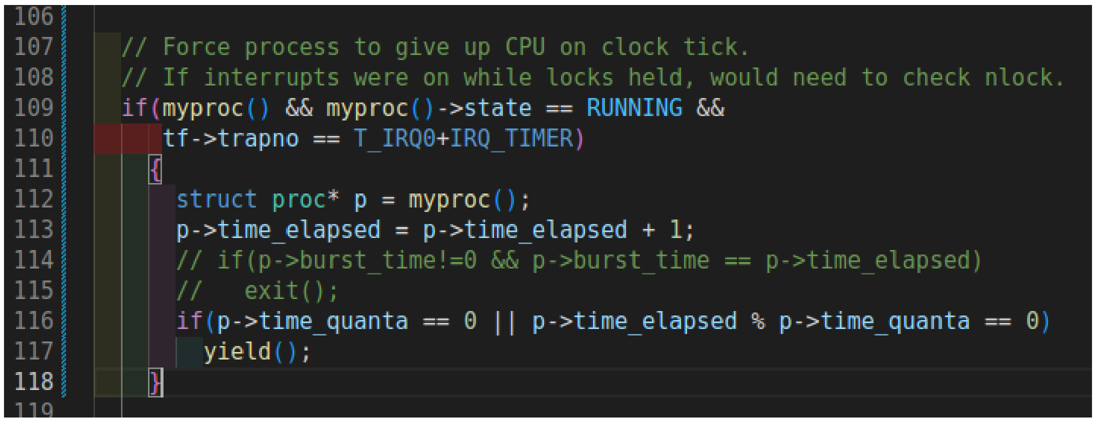
We calculate time quantum while setting burst times. Initially when quanta is 0, a positive value of burst time is assigned. After that minimum among burst time and quanta is assigned as the quanta.
Eg. When a process forks 10 different processes, each process is assigned pid and burst times. Yield is placed here so that a scheduling cycle is skipped for all processes. This is important so that before starting execution of all processes, all burst times are allotted.



```
int setBurstTime(int n)
{
  acquire(&ptable.lock);
  struct proc *curproc = myproc();

  curproc->burst_time=n;

  struct proc *edittedproc = myproc();
  cprintf("burst time set to %d\n",edittedproc->burst_time);

  if(quanta==0 && n!=0)
    quanta=n;
  if(quanta>0 && n!=0 && n<quanta)
    quanta=n;

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

This is trap.c extract.
Line 109-118 is commented to make SJFS scheduling. It disables clock tick interrupts.

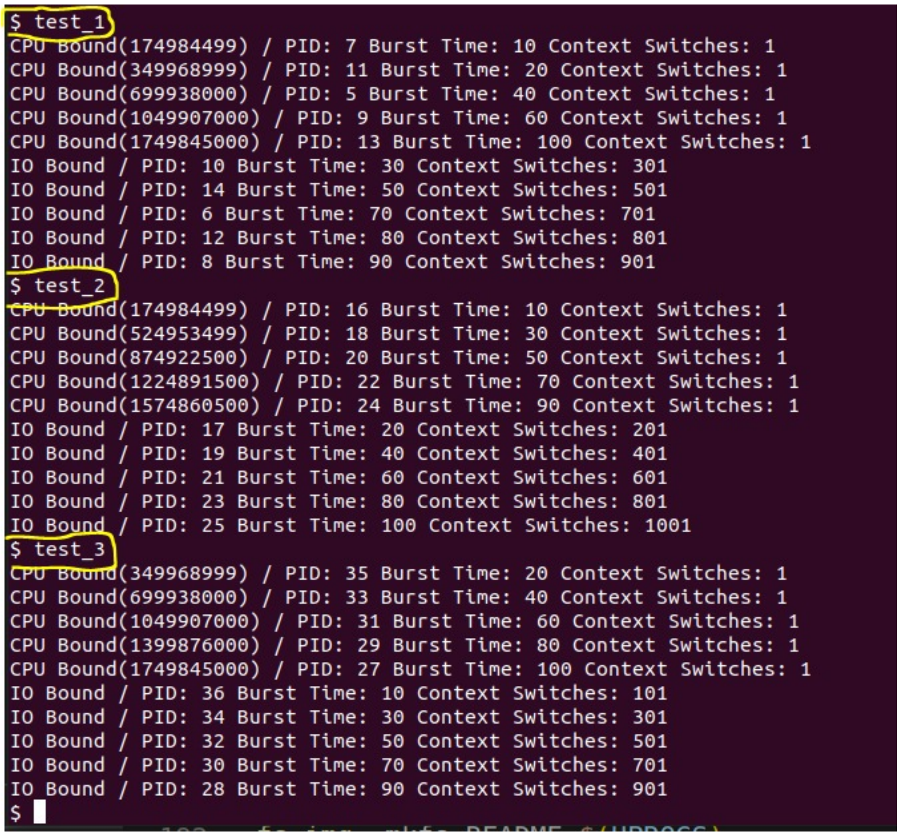
- During hybrid scheduling we check if burst time is not zero and time quanta of cycles are alloted. After that clock tick interruption is flagged which preempts the process.



```
106
107   // Force process to give up CPU on clock tick.
108   // If interrupts were on while locks held, would need to check nlock.
109   if(myproc() && myproc()->state == RUNNING &&
110      tf->trapno == T_IRQ0+IRQ_TIMER)
111     {
112       struct proc* p = myproc();
113       p->time_elapsed = p->time_elapsed + 1;
114       // if(p->burst_time!=0 && p->burst_time == p->time_elapsed)
115       //   exit();
116       if(p->time_quanta == 0 || p->time_elapsed % p->time_quanta == 0)
117         yield();
118     }
```

Test Cases

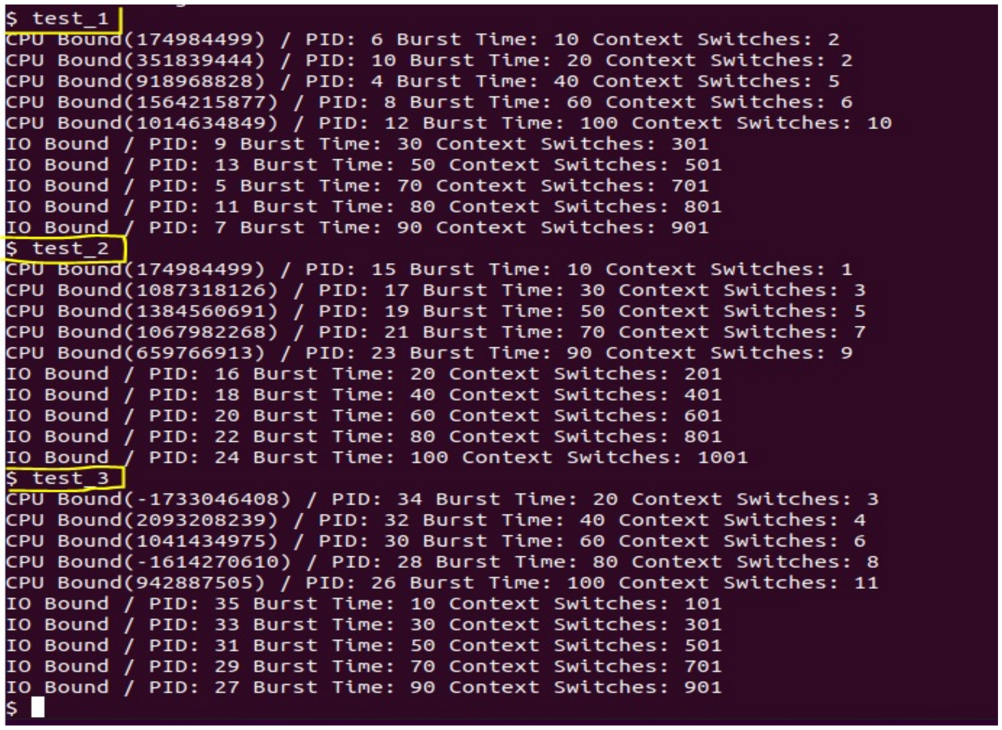
- Shortest job scheduling First



```
$ test_1
CPU Bound(174984499) / PID: 7 Burst Time: 10 Context Switches: 1
CPU Bound(349968999) / PID: 11 Burst Time: 20 Context Switches: 1
CPU Bound(699938000) / PID: 5 Burst Time: 40 Context Switches: 1
CPU Bound(1049907000) / PID: 9 Burst Time: 60 Context Switches: 1
CPU Bound(1749845000) / PID: 13 Burst Time: 100 Context Switches: 1
IO Bound / PID: 10 Burst Time: 30 Context Switches: 301
IO Bound / PID: 14 Burst Time: 50 Context Switches: 501
IO Bound / PID: 6 Burst Time: 70 Context Switches: 701
IO Bound / PID: 12 Burst Time: 80 Context Switches: 801
IO Bound / PID: 8 Burst Time: 90 Context Switches: 901
$ test_2
CPU Bound(174984499) / PID: 16 Burst Time: 10 Context Switches: 1
CPU Bound(524953499) / PID: 18 Burst Time: 30 Context Switches: 1
CPU Bound(874922500) / PID: 20 Burst Time: 50 Context Switches: 1
CPU Bound(1224891500) / PID: 22 Burst Time: 70 Context Switches: 1
CPU Bound(1574860500) / PID: 24 Burst Time: 90 Context Switches: 1
IO Bound / PID: 17 Burst Time: 20 Context Switches: 201
IO Bound / PID: 19 Burst Time: 40 Context Switches: 401
IO Bound / PID: 21 Burst Time: 60 Context Switches: 601
IO Bound / PID: 23 Burst Time: 80 Context Switches: 801
IO Bound / PID: 25 Burst Time: 100 Context Switches: 1001
$ test_3
CPU Bound(349968999) / PID: 35 Burst Time: 20 Context Switches: 1
CPU Bound(699938000) / PID: 33 Burst Time: 40 Context Switches: 1
CPU Bound(1049907000) / PID: 31 Burst Time: 60 Context Switches: 1
CPU Bound(1399876000) / PID: 29 Burst Time: 80 Context Switches: 1
CPU Bound(1749845000) / PID: 27 Burst Time: 100 Context Switches: 1
IO Bound / PID: 36 Burst Time: 10 Context Switches: 101
IO Bound / PID: 34 Burst Time: 30 Context Switches: 301
IO Bound / PID: 32 Burst Time: 50 Context Switches: 501
IO Bound / PID: 30 Burst Time: 70 Context Switches: 701
IO Bound / PID: 28 Burst Time: 90 Context Switches: 901
$
```

As we can see in SJFS scheduling, CPU processes have 1 context switch because CPU bound processes complete and then exit. IO bound processes are implemented using sleep, so context switches happen among them.

- Hybrid scheduling results

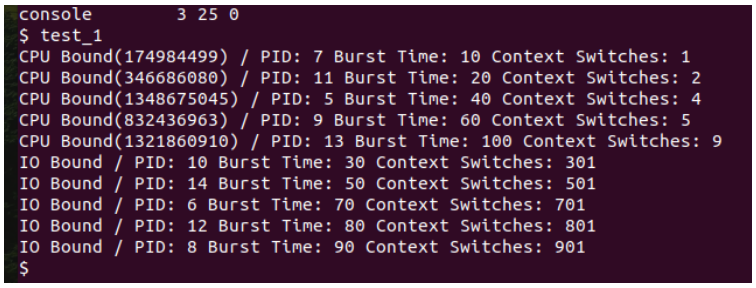


```
$ test_1
CPU Bound(174984499) / PID: 6 Burst Time: 10 Context Switches: 2
CPU Bound(351839444) / PID: 10 Burst Time: 20 Context Switches: 2
CPU Bound(918968828) / PID: 4 Burst Time: 40 Context Switches: 5
CPU Bound(1564215877) / PID: 8 Burst Time: 60 Context Switches: 6
CPU Bound(1014634849) / PID: 12 Burst Time: 100 Context Switches: 10
IO Bound / PID: 9 Burst Time: 30 Context Switches: 301
IO Bound / PID: 13 Burst Time: 50 Context Switches: 501
IO Bound / PID: 5 Burst Time: 70 Context Switches: 701
IO Bound / PID: 11 Burst Time: 80 Context Switches: 801
IO Bound / PID: 7 Burst Time: 90 Context Switches: 901
$ test_2
CPU Bound(174984499) / PID: 15 Burst Time: 10 Context Switches: 1
CPU Bound(1087318126) / PID: 17 Burst Time: 30 Context Switches: 3
CPU Bound(1384560691) / PID: 19 Burst Time: 50 Context Switches: 5
CPU Bound(1067982268) / PID: 21 Burst Time: 70 Context Switches: 7
CPU Bound(659766913) / PID: 23 Burst Time: 90 Context Switches: 9
IO Bound / PID: 16 Burst Time: 20 Context Switches: 201
IO Bound / PID: 18 Burst Time: 40 Context Switches: 401
IO Bound / PID: 20 Burst Time: 60 Context Switches: 601
IO Bound / PID: 22 Burst Time: 80 Context Switches: 801
IO Bound / PID: 24 Burst Time: 100 Context Switches: 1001
$ test_3
CPU Bound(-1733046408) / PID: 34 Burst Time: 20 Context Switches: 3
CPU Bound(2093208239) / PID: 32 Burst Time: 40 Context Switches: 4
CPU Bound(1041434975) / PID: 30 Burst Time: 60 Context Switches: 6
CPU Bound(-1614270610) / PID: 28 Burst Time: 80 Context Switches: 8
CPU Bound(942887505) / PID: 26 Burst Time: 100 Context Switches: 11
IO Bound / PID: 35 Burst Time: 10 Context Switches: 101
IO Bound / PID: 33 Burst Time: 30 Context Switches: 301
IO Bound / PID: 31 Burst Time: 50 Context Switches: 501
IO Bound / PID: 29 Burst Time: 70 Context Switches: 701
IO Bound / PID: 27 Burst Time: 90 Context Switches: 901
$
```

Here there can be multiple context switches in CPU bound processes.
IO bound as earlier have 300 sleep cycles, so 300 context switches.

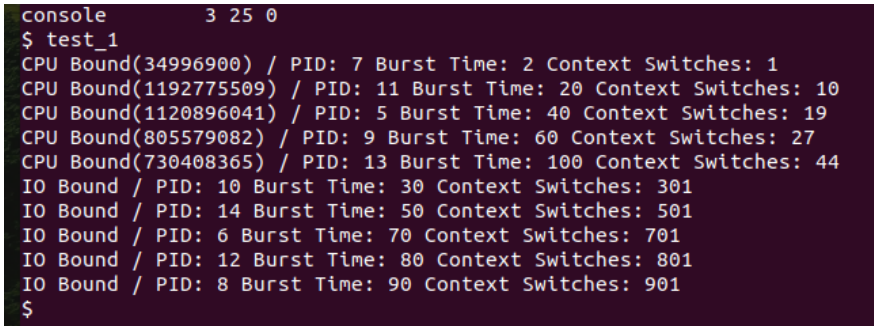
1 context switch is added due to the initial yield added in set burst time.

- Changes due to time quantum.



```
console        3 25 0
$ test_1
CPU Bound(174984499) / PID: 7 Burst Time: 10 Context Switches: 1
CPU Bound(346686080) / PID: 11 Burst Time: 20 Context Switches: 2
CPU Bound(1348675045) / PID: 5 Burst Time: 40 Context Switches: 4
CPU Bound(832436963) / PID: 9 Burst Time: 60 Context Switches: 5
CPU Bound(1321860910) / PID: 13 Burst Time: 100 Context Switches: 9
IO Bound / PID: 10 Burst Time: 30 Context Switches: 301
IO Bound / PID: 14 Burst Time: 50 Context Switches: 501
IO Bound / PID: 6 Burst Time: 70 Context Switches: 701
IO Bound / PID: 12 Burst Time: 80 Context Switches: 801
IO Bound / PID: 8 Burst Time: 90 Context Switches: 901
$
```

In this test, the time quantum was 10 bursts.


```
console        3 25 0
$ test_1
CPU Bound(34996900) / PID: 7 Burst Time: 2 Context Switches: 1
CPU Bound(1192775509) / PID: 11 Burst Time: 20 Context Switches: 10
CPU Bound(1120896041) / PID: 5 Burst Time: 40 Context Switches: 19
CPU Bound(805579082) / PID: 9 Burst Time: 60 Context Switches: 27
CPU Bound(730408365) / PID: 13 Burst Time: 100 Context Switches: 44
IO Bound / PID: 10 Burst Time: 30 Context Switches: 301
IO Bound / PID: 14 Burst Time: 50 Context Switches: 501
IO Bound / PID: 6 Burst Time: 70 Context Switches: 701
IO Bound / PID: 12 Burst Time: 80 Context Switches: 801
IO Bound / PID: 8 Burst Time: 90 Context Switches: 901
$
```

In this test, the time quantum was 2 bursts.
And corresponding changes in context switches can be seen. Tests with lower quanta have higher context switches.