

GROUP 9 MNC

CHANDRASHEKHAR DHARMARAJAN	220123076
SRINJOY SOM	220123074
SHUBHAM KUMAR JHA	220123081
DHRUVA KUTHARI	220123015

Patch files

Part - A

i)

To solve the first question we have added two additional members in the struct proc i.e tickets and ticks. Processes are supposed to have 10 tickets by defaults. So in the allocproc function we are setting it to 10 whenever created. For allotting a number of tickets for processes we have used the settickets function in sysproc.c.

```
static struct proc*
allocproc(void)
{
    struct proc *p;
    char *sp;

    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found;

    release(&ptable.lock);
    return 0;

found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->tickets = 10;
    p->ticks = 0;
    release(&ptable.lock);
}
```

We define the system call settickets(int tickets) , which assigns the specified number of tickets to a given process . The code for the system call is as follows :

```
int
sys_settickets(void)
{
    int n;
    struct proc *p = myproc(); // Get the current process using myproc()

    if (argint(0, &n) < 0) // Retrieve the integer argument
        return -1;

    if (n < 1 || n > 100000) // Ensure the number of tickets is within range
        return -1;

    p->tickets = n; // Set the number of tickets for the current process

    //cprintf("Reached Here") ;
    return 0;
}
```

We also followed the regular steps of assigning the system call number and making the required changes in the file user.h , usys.S and syscall.c to make the system call function properly .

ii)

Now in second question of part A -

We have to get information of all the processes in a data structure defined as

```
struct processes_info {
    int num_processes;
    int pids[NPROC];
    int ticks[NPROC]; // ticks = number of times process has
                      // been scheduled
    int tickets[NPROC]; // tickets = number of tickets set by
                      // settickets()
}
```

```
};
```

Now to put the values to these files we have to iterate over the whole ptable and find all the processes that are not flagged as unused . Set number of such processes as num-process of process_info .Also for all the values of $i < \text{NPROC}$ where a not unused process is there we fill the value of their pid in pids array .In ticks array fill in the no of ticks value of all the processes(not unused) and same for tickets as well.

We first declare the struct proc_info in a file called procinfo.h .

```
C processInfo.h X
C processInfo.h
1 struct processInfo
2 {
3     int ppid;
4     int psize;
5     int numberContextSwitches;
6 };
7
8
9 #define NPROC 64 // Ensure this matches the value used in proc.c
10
11 struct processes_info {
12     int num_processes;
13     int pids[NPROC];
14     int ticks[NPROC]; // Number of times process has been scheduled
15     int tickets[NPROC]; // Number of tickets set by settickets()
16 };
```

Then we included this header file in proc.h and wrote a function called fetch_processinfo in proc.c , which gets the required information .

```

int
fetch_processes_info(struct processes_info *p)
{
    struct proc *proc_entry;
    int num_processes = 0;

    // Check if the user pointer is valid
    if (p == 0 || !p->pids || !p->ticks || !p->tickets) {
        return -1;
    }

    acquire(&ptable.lock); // Lock the process table

    // Iterate over the process table
    for (proc_entry = ptable.proc; proc_entry < &ptable.proc[NPROC]; proc_entry++) {
        if (proc_entry->state == UNUSED)
            continue; // Skip UNUSED processes

        if (num_processes >= NPROC) // Prevent overflow
            break;

        p->pids[num_processes] = proc_entry->pid;
        p->ticks[num_processes] = proc_entry->ticks;
        p->tickets[num_processes] = proc_entry->tickets;
        num_processes++;
    }

    release(&ptable.lock); // Unlock the process table

    // Set the number of non-UNUSED processes found
    p->num_processes = num_processes;

    return 0; // Success
}

```

And finally we use this function in sysproc.c to write the actual system call . Note that this procedure had to be followed because the process table can only be acquired inside proc.c , so we had to declare and define the function there .

```

int
sys_getprocessesinfo(void)
{
    struct processes_info *p;

    // Retrieve the user argument: a pointer to a struct processes_info
    if (argptr(0, (void*)&p, sizeof(*p)) < 0) {
        return -1;
    }

    // Call the helper function to fetch process information
    return fetch_processes_info(p);
}

```

Also in order keep the number of ticks updated (note that the number of ticks here mean the number of times the process is scheduled) we update the scheduler function to update the ticks for a process when it is scheduled . We will attach the code snippet for the scheduler function in the upcoming question .

iii)

We have changed the default scheduler of xv6 into a lottery scheduler using the tickets assigned to the processes. To design the lottery scheduler we traverse the array of all the processes to count total tickets assigned to the used processes then we choose a random number from 1 to the calculated number of tickets. To generate a random number we used `next_random` function which in turn use park miller method which has been mentioned in the problem statement . Then we again go through the `ptable` and check in which process's range the number falls into and then select the process and change its state to running.

The following is the code for the pseudo random generator which has been included in the `proc.c` file .

```

static unsigned random_seed = 1;

#define RANDOM_MAX ((1u << 31u) - 1u)
unsigned lcg_parkmiller(unsigned *state)
{
    const unsigned N = 0x7fffffff;
    const unsigned G = 48271u;

    /*
       Indirectly compute state*G%N.

       Let:
       div = state/(N/G)
       rem = state%(N/G)

       Then:
       rem + div*(N/G) == state
       rem*G + div*(N/G)*G == state*G

       Now:
       div*(N/G)*G == div*(N - N%G) == -div*(N%G) (mod N)

       Therefore:
       rem*G - div*(N%G) == state*G (mod N)

       Add N if necessary so that the result is between 1 and N-1.
    */
    unsigned div = *state / (N / G); /* max : 2,147,483,646 / 44,488 = 48,271 */
    unsigned rem = *state % (N / G); /* max : 2,147,483,646 % 44,488 = 44,487 */

    unsigned a = rem * G; /* max : 44,487 * 48,271 = 2,147,431,977 */
    unsigned b = div * (N % G); /* max : 48,271 * 3,399 = 164,073,129 */

    return *state = (a > b) ? (a - b) : (a + (N - b));
}

unsigned next_random() {
    return lcg_parkmiller(&random_seed);
}

```

Now we give the updated scheduler function which works on the principle of lottery scheduling(implementation discussed above)

```

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

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

        // Loop over process table looking for process to run.
        acquire(&ptable.lock);
        int total_tickets = 0 ;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state == RUNNABLE){
                total_tickets += p->tickets ;
            }
        }

        if(total_tickets > 0){
            unsigned rand_num = next_random() % total_tickets + 1;

            int curr_total = 0 ;
            for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                if(p->state == RUNNABLE){
                    curr_total += p->tickets ;
                    if(curr_total >= rand_num){
                        p->ticks ++ ;
                        c->proc = p ;
                        switchvm(p) ;
                        p->state = RUNNING ;
                        swtch(&(c->scheduler), p->context);
                        switchkvm();

                        // Process is done running for now.
                        // It should have changed its p->state before coming back.
                        c->proc = 0;
                        break ;
                    }
                }
            }
        }

        release(&ptable.lock);
    }
}

```

We test the changes made in the OS code by given test programmes.

1)processlist.c : This user program was there to test the functioning of the getprocessinfo system call . We got the following output .

```
$ processlist
3 running processes
PID      TICKETS TICKS
1         10     24
2         10     18
3         10      7
```

2) timewithtickets.c : This user program tested the distribution of ticks for a process , based on the allotted tickets .

```
$ timewithtickets 5 10 20 30
TICKETS TICKS
10       0
20       1
30       4
$ timewithtickets 50 100 1000 100000
TICKETS TICKS
100      0
1000     3
100000   47
$ timewithtickets 50 20 30 60
TICKETS TICKS
20       9
30       9
60      32
$ timewithtickets 10 10 10 10
TICKETS TICKS
10       4
10       1
10       5
```

This output snippet shows the distribution for various test cases that we tried out . We can see that the larger ticket number have more tickets as expected .

3)lotterytest.c : This user program tests the overall functioning of our lottery scheduler in various cases .

```
$ lotterytest
one process: passed 3 of 3
two processes, unequal ratio: passed 7 of 7
two processes, unequal ratio, small ticket count: passed 7 of 7
two processes, equal: passed 7 of 7
two processes, equal, small ticket count: passed 7 of 7
three processes, unequal: passed 9 of 9
three processes, unequal, small ticket count: passed 9 of 9
three processes, but one io-wait: passed 9 of 9
three processes, but one exits: passed 9 of 9
seven processes: passed 17 of 17
two processes, not all yielding: passed 7 of 7
overall: passed 91 of 91
```

We can see that all the test cases have been passed successfully.
To run the code

Make sure to have a linux system

make qemu

If it runs successfully then run the test programs.

Else

If it shows mkfs permission denied error 127 -

Then run

ls -l ./mkfs

If it does not have execute permission
then run

chmod +x ./mkfs

and then

make mkfs

Part - B

(1) To create the system calls **getNumProc()** and **getMaxPID()**, a procedure similar to the one used to create a system call in Assignment-0B is used. We also create user programs named **getNumProcTest** and **getMaxPIDTest** to use the above system calls. Two functions namely, **getNumProcHelper** and **getMaxPIDHelper** are implemented in `proc.c` which help us in achieving the desired functionalities. They are attached below.

```
int getNumProcHelper(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;
}

int getMaxPIDHelper(void){
    int max=0;
    struct proc *p;

    acquire(&ptable.lock);

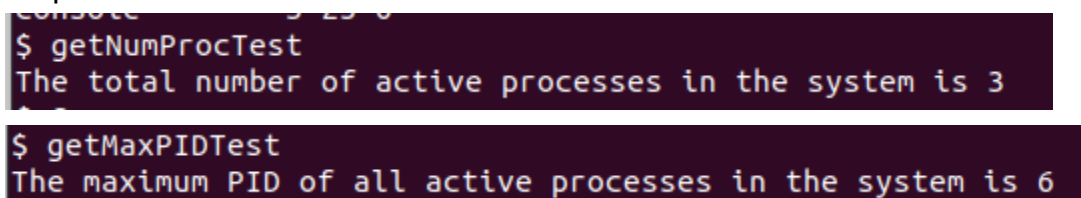
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->state != UNUSED){
            if(p->pid > max)
                max = p->pid;
        }
    }

    release(&ptable.lock);

    return max;
}
```

The functions access `ptable` by acquiring its lock and then loop through it to carry out their respective tasks.

Output:



```
console 3 23 0
$ getNumProcTest
The total number of active processes in the system is 3
$ getMaxPIDTest
The maximum PID of all active processes in the system is 6
```

(2) The initial steps for implementing the **getProcInfo** system call are similar to those of previously created system calls. However, in this case, we also need to **pass certain arguments to the system call**, unlike the earlier ones. Additionally, we must come up with a method to track and store the number of context switches for each process. We solve the problem of passing parameters to syscall using `argptr` which is a predefined system call which serves our purpose.

To handle context switching, we will modify the struct `proc` by adding a new member called `nocs`, which will keep track of the number of context switches. We initialize `nocs` for each process by setting `p->nocs` to 0 in `allocproc()`, since processes are allocated space in the process table (`ptable`) using `allocproc()` before they start running. Next, we

insert `(p->nocs)++` into `scheduler()`, ensuring that each time a process is scheduled, the number of context switches is incremented appropriately.

We also create a dummy process called `defaultParent` and set it as the parent for every new process using `p->parent = &defaultParent` in `allocproc()`. After the process is allocated, `fork()` replaces `defaultParent` with the real parent. We assign `defaultParent` a PID of `-2` in `scheduler()`. With `defaultParent`, we can quickly determine whether a process has a parent; if it does, we retrieve the parent's PID using `p->parent->pid`.

```
int sys_getProcInfo(void){
    int pid;

    struct processInfo *procInfo;
    argptr(0,(void *)&pid, sizeof(pid));
    argptr(1,(void *)&procInfo, sizeof(procInfo));

    struct processInfo tempInfo = getProcInfoHelper(pid);

    if(tempInfo.ppid == -1){
        return -1;
    }

    procInfo->ppid = tempInfo.ppid;
    procInfo->psize = tempInfo.psize;
    procInfo->numberContextSwitches = tempInfo.numberContextSwitches;

    return 0;
}

struct processInfo getProcInfoHelper(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;
}
```

The output obtained on running `getProcInfoTest` is given below:

```
$ getProcInfoTest 2
PPID: 1
Size: 20480
Number of Context Switches: 55
```

(3) For this part, we need to add a new attribute, `burstTime`, to the `proc` structure. Additionally, we will implement two system calls: `setBurstTime` and `getBurstTime`. These will allow us to set and retrieve the burst time of the current process, respectively. Although we have already added the `burstTime` attribute, we need to initialize it with a default value. We set `p->burstTime` to `0` (with `0` being the default burst time) in `allocproc()`.

The next challenge is how to access the current process without information like the process ID. To solve this, we can use the predefined `myproc()` function in `xv6`, which returns a pointer to the `proc` structure of the current process. Using this function, we can easily access and modify the burst time for the current process.

```

int setBurstTimeHelper(int burst_time){
    struct proc *p = myproc();
    p->burst_time = burst_time;
    yield();
    return 0;
}

int getBurstTimeHelper(){
    struct proc *p = myproc();
    return p->burst_time;
}

```

(Output obtained on running user program)

```

$ burstTimeTest
This is a sample process to test setBurstTime and getBurstTime system calls.
The burst time is: 3

```

Part - C

Implementation of 'Shortest Job First' (SJF scheduler) is done by making the following two changes in the 'Round Robin' scheduler :

1. 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 UePRYe the following lines from WUaS.c to fix this issue:

```

103
104 // Force process to give up CPU on clock tick.
105 // If interrupts were on while locks held, would need to check nlock.
106 if(myproc() && myproc()->state == RUNNING &&
107    tf->trapno == T_IRQ0+IRQ_TIMER)
108     yield();
109

```

2. 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 extractMin. We had to make the following changes in proc.c:
 - a. Declare priority queue and implement spinlock.
 - b. Implement the following functions in the priority queue:
 - i. insertIntoHeap
 - ii. isEmpty (Checks if the priority queue is empty or not)
 - iii. isFull (Checks if the priority queue is full or not)
 - iv. extractMin (Removes the process at the front of the queue)

- v. `changeKey` (Changes the burst time of a process with a given PID in the priority queue and updates the queue accordingly)
- vi. `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. Change the scheduler so that it uses the priority queue to schedule the next process.
- d. Insert processes into the priority queue as and when their state becomes `RUNNABLE`. This happens in five functions - `yield`, `kill`, `fork`, `userinit` and `wakeup1`. A check variable 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.
- e. Insert a **`yield`** call into `setBurstTime`. This is because when the 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 setBurstTimeHelper(int burst_time){

    struct proc *p = myproc();
    p->burst_time = burst_time;
    yield();

    return 0;
}
```

Runtime complexity:

The runtime complexity of the scheduler is $O(\log n)$ because `extractMin` has a $O(\log n)$ time complexity and that is the dominating part of the scheduling process. The rest of the statements run in $O(1)$ time.

Handling Corner Cases

1. If the queue is empty, `extractMin` returns 0 after which the scheduler doesn't schedule any process. (See scheduler function)
2. 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.
3. 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.

4. 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.
5. 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 the scheduler:

In order to test our scheduler we fork multiple processes and give 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). **Loops are ignored by the compiler if the information computed in the loop isn't used later.** Hence, we use the information computed in the loop later.

I/O bound processes are 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.

testScheduler 6				
PID	Type	Burst Time	Context Switches	
27	CPU	6	2	
29	CPU	20	2	
25	CPU	20	2	
26	I/O	1	102	
28	I/O	2	102	
24	I/O	13	102	

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 setBurstTime 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)

PART - C (BONUS)

We can implement a **hybrid scheduler** which **combines SJF and Round Robin** by making changes in **trap.c**, **proc.c**, **defs.h** files.

We begin by implementing the **logic for setting up a time quantum**. To achieve this, we declare an **external integer variable** in **defs.h**, named **quant**, which can then be **initialized in proc.c**. By **default**, this quant variable is **set to 1000**, as burst times typically range between 1 and 1000. The assignment requires that the time quantum be set to the burst time of the process with the shortest burst time in the priority queue. However, since the default burst time is zero, we can't predict how long a process will run if its burst time is zero. In such cases, we assign a default burst time of zero. The **quant variable** is then **updated in the setBurstTime function**. If the burst time being set is smaller than the current quant value, quant is updated to match the new burst time.

Next, we create a second priority queue called pqueue2 along with the associated functions for managing this new queue. Functions such as insertIntoHeap, fix, and extractMax were originally created for pqueue, and now we create similar functions for pqueue2 like insertIntoHeap2, extractMax2, and fix2. These functions operate in the same way as the original ones, with the only difference being that they modify pqueue2 instead of pqueue.

Next, we modify the clock tick interrupt handler in the trap.c file. With each clock tick, the running time (rt), a newly added parameter in struct proc (initialized to zero in allocproc()), is incremented for the current process (myproc()). By default, processes have a burst time of zero. If a process's burst time hasn't been manually set, we don't want to terminate the process as soon as its first clock tick occurs, as this could disrupt the OS's functioning. Therefore, before checking if the running time equals the process's burst time, we first ensure that the burst time isn't zero. We only compare rt with burstTime when burstTime is non-zero. If they are equal, the process is terminated using exit(). Otherwise, we perform another check—if the running time is divisible by the

time quantum (quant). If it is, we preempt the current process and insert it 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);
}
```

Next, we update the scheduler. In the new version, when deciding which process to run, we first check if the original priority queue, pqueue, is empty. If it's not empty, we extract the process with the minimum priority using extractMin from pqueue and run that process. If pqueue is empty, we transfer all processes from pqueue2 (the ones that were preempted due to reaching the time quantum) back into pqueue. Once this is done, we proceed as usual by selecting the next process from pqueue with extractMin to run it.

```
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

        // int flag = 0;

        // acquire(&pqueue.lock);
        // for(int i=1;i<=pqueue.sz;i++){
        //     if(pqueue.proc[i]->burst_time != 0)flag=1;
        // }
        // release(&pqueue.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;
switchvm(p);

p->state = RUNNING;
(p->nocs)++;

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

switchkvm();

c->proc = 0;

// //THE OLD ROUND ROBIN SCHEDULER
// // Loop over process table looking for process to run.
// for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
// // while(!isEmpty()){
// // // p=extractMin();
// // // struct proc *temp = p;
// // // if(isEmpty())
// // // break;
// // // p=extractMin();
// // if(p->state != RUNNABLE)
// // continue;
// // // Switch to chosen process. It is the process's job
// // // to release ptable.lock and then reacquire it
// // // before jumping back to us.
// // c->proc = p;
// // switchvm(p);
// // p->state = RUNNING;
// // (p->nocs)++;
// // swtch(&(c->scheduler), p->context);
// // switchkvm();
// // // Process is done running for now.
// // // It should have changed its p->state before coming back.
// // c->proc = 0;
// // p=temp;
// // }
// }
release(&ptable.lock);
}

```

The final step is testing, which is carried out using three user programs: testScheduler, cpuProcTester, and ioProcTester. Here's what was done in each of these code files:

- In testScheduler.c, half of the forked processes are I/O-bound, while the other half are CPU-bound. Each forked process is assigned a random burst time between 1 and 1000. The CPU-bound processes run loops for 10^9 iterations, while the I/O-bound processes simulate I/O by calling sleep(1) ten times.
- In cpuProcTester.c, all forked processes are CPU-bound. The burst times are randomly assigned between 1 and 1000, and each process runs a loop of 10^9 iterations (which takes around 200 clock ticks).
- In ioProcTester.c, every forked process is I/O-bound. To simulate I/O, the processes call sleep(1) ten times. As with the other tests, burst times are randomly assigned between 1 and 1000.

testScheduler 6			
PID	Type	Burst Time	Context Switches
---	---	-----	-----
8	CPU	264	2
10	CPU	985	2
6	CPU	999	2
7	I/O	21	12
9	I/O	71	12
5	I/O	635	12

cpuProcTester 6			
PID	Type	Burst Time	Context Switches
---	---	-----	-----
13	CPU	999	11
17	CPU	985	12
15	CPU	264	13
12	CPU	635	13

ioProcTester 6			
PID	Type	Burst Time	Context Switches
---	---	-----	-----
21	I/O	21	202
23	I/O	71	202
22	I/O	264	202
19	I/O	635	202
24	I/O	985	202
20	I/O	999	202

Results:

Three key observations:

1. **Incomplete CPU-bound Processes:** Not all CPU-bound processes completed their execution. This occurred because some processes had burst times shorter than their actual runtime, leading to termination before they could print any output. This confirms that when a process's `rt` value (running time) equals its `burstTime`, the process is correctly exited.
2. **Increased Context Switches for CPU-bound Processes:** The number of context switches significantly increased with the new hybrid scheduling for CPU-bound processes. This is due to the CPU-bound processes being preempted every quant clock ticks.

3. **I/O-bound Processes Unaffected by Preemption:** The I/O-bound processes were not impacted by preemption and behaved similarly to how they would under Shortest Job First (SJF) scheduling. This is because they spend most of their time sleeping, with very low actual execution time. Since quant is typically a 2-3 digit number (burst times range randomly between 1 and 1000), the likelihood of I/O-bound processes experiencing quant clock ticks is low. As a result, they were not preempted frequently and simply alternated between sleeping and running, maintaining the same number of context switches as in SJF scheduling.

Out-of-Order Execution in Mixed Workloads: When both CPU-bound and I/O-bound processes were running, some CPU-bound processes were preempted more often. This happened because I/O-bound processes returned from the SLEEPING state and forced the currently running CPU-bound processes to be preempted. As a result, some CPU-bound processes experienced out-of-order execution.