CS344, Assignment 2(Process Management and Scheduling)

Group Number 17

Name	Roll Number
1. Samiksha Sachdeva	180123040
2. Kartikay Goel	180101033
3. Falak Chhikara	180101023
4. Rahul Choudhary	180101060

Part A:

1. **Tasks: To implement getNumProc(), getMaxPid().**
2. **Task: To implement getProcInfo(pid, &processInfo).**
3. **Tasks: To implement set_burst_time(n), get_burst_time().**

We have also added a system call process_state() to print the states of the active processes.
Also Added five programs namely printNumProc, printMaxPid, printProcInfo, testBurstTime, forktest to test the system calls.

To add these system calls, we have made modifications to the following files in xv6 OS.
All system calls that do not have an explicit return value mentioned above return 0 on success and a negative value on failure.

1. **user.h**
   Added the system call definitions in xv6.

```
41  // Function declaration for the system call
42  int getNumProc(void);
43  int getMaxPid(void);
44  int getProcInfo(int ,struct processInfo*);
45  int set_burst_time(int);
46  int get_burst_time(void);
47  int process_state(void);
```

2. **usys.S**
   Added the system calls exported by the kernel.

```
32  SYSCALL(getNumProc)
33  SYSCALL(getMaxPid)
34  SYSCALL(getProcInfo)
35  SYSCALL(set_burst_time)
36  SYSCALL(get_burst_time)
37  SYSCALL(process_state)
```

3. **syscall.h**
   Added mapping from system call name to system call number.

```
23  #define SYS_getNumProc 22
24  #define SYS_getMaxPid 23
25  #define SYS_getProcInfo 24
26  #define SYS_set_burst_time 25
27  #define SYS_get_burst_time 26
28  #define SYS_process_state 27
```

4. **syscall.c**
   Added helper functions to parse system call arguments, and pointers to the actual system call implementations.

```
106 extern int sys_getNumProc(void);
107 extern int sys_getMaxPid(void);
108 extern int sys_getProcInfo(void);
109 extern int sys_set_burst_time(void);
110 extern int sys_get_burst_time(void);
111 extern int sys_process_state(void);
```

```
135     [SYS_getNumProc] sys_getNumProc,
136     [SYS_getMaxPid] sys_getMaxPid,
137     [SYS_getProcInfo] sys_getProcInfo,
138     [SYS_set_burst_time] sys_set_burst_time,
139     [SYS_get_burst_time] sys_get_burst_time,
140     [SYS_process_state] sys_process_state,
```

5. **sysproc.c**
   Implemented the process related system calls.(implementation of the system calls displayed in the left screenshot above.)

```
// get max PID amongst the PIDs of all currently active processes
int sys_getMaxPid(void){
  return getMaxPid();      // call getMaxPid function
}

// get number of active processes in the system
int sys_getNumProc(void){
    return getNumProc();    // call getNumProc function
}

// gives info of the process
int sys_getProcInfo(void){
  int pid;
  struct processInfo *pif;
  int sz1 = sizeof(pid);
  int sz2 = sizeof(pif);
  argptr(0, (void *)&pid, sz1);
  argptr(1, (void *)&pif, sz2);
  return getProcInfo(pid, pif);     // call getProcInfo function
}

// set burst time of a process
int sys_set_burst_time(void)
{
  int n;
  int sz = sizeof(n);
  argptr(0, (void *)&n, sz);
  return set_burst_time(n);       // call set_burst_time function
}

// get burst time of a process
int sys_get_burst_time(void){
  return get_burst_time();        // call get_burst_time function
}

// get the state of the process
int sys_process_state(void){
  return process_state();         // call process_state function
}
```

6. **proc.h**
   It contains the struct proc structure. Added two more parameters in proc struct.

```
52    int count_contextswitch;     // Counter for Context Switches of the process
53    int burst_time;              // Burst Time for process
54  };
```

7. **proc.c(Code is present in the proc.c file. (line 597 to 685).)**
   Implemented the following functions:

a) **int getNumProc(void)**
b) **int getMaxPid(void)**
c) **int getProcInfo(int pid, struct processInfo *pif)**
d) **int set_burst_time(int bt)**
e) **int get_burst_time()**
f) **int process_state()**

Made one initialization in allocProc function(initialized number of context switches to zero)

8. **Added the processInfo.h and defs.h file provided in the patch.**

```
123  int·····getNumProc(void);
124  int·····getMaxPid(void);
125  int·····getProcInfo(int, struct processInfo*);
126  int·····set_burst_time(int);
127  int·····get_burst_time(void);
128  int·····process_state(void);
```

9. **Made necessary changes in Makefile.**
   Added user programs to **UPROGS** and **c files** corresponding to the programs to **EXTRA.**

Output:

Programs to test system calls:

1. printNumProc: System call that gets the total number of processes.(either in embryo, running, runnable, sleeping, or zombie states).
2. printMaxPid: System call that gets the maximum PID among all the processes.
3. printProcInfo: System call that gets process information, ie., the parent PID, the number of times the process was context switched in by the scheduler, and the process size in bytes.
4. forktest: System call that forks the process and checks if the fork call was successful or not.
5. testBurstTime: System call that sets and gets the burst time of the currently scheduled process.

```
$ printNumProc
Number of active processes(in either state): 3
$ printMaxPid
Maximum PID(process ID) is: 4
$ printProcInfo 4
Process Info:
  Parent PID is: 1
 Size of the process: 16384
Number of context switches: 21
$ printProcInfo 6
Process Info:
  Parent PID is: 1
 Size of the process: 16384
Number of context switches: 23
$ forktest
Fork test started
OK tested.
$ testBurstTime
Burst Time before setting burst time: 0
Burst Time after setting burst time: 15
```

Part B:

We have modified the default round robin scheduler to take into account user-defined process burst time and implemented a shortest job first scheduler. This scheduling algorithm chooses the job with the lowest burst time and schedules it for execution after an already scheduled job is completed.

Design and Implementation:

1. **We have designed a scheduler to implement the Shortest Job First scheduling policy.**
2. **Implementation:**

```
// Shortest Job First scheduler implementation
void scheduler(void)
{
  struct proc *p,*lowest_bt,*p1;        //lowest_bt is the process with sho
  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);
    lowest_bt= 0;
    int currentMin = __INT_MAX__; // CPU burst times are between 1 and 20.
    for (p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)
    {
      if (p1->state != RUNNABLE)
      {
        continue;
      }
      if(p1->burst_time < currentMin)
      {
        lowest_bt = p1;
        currentMin = p1->burst_time;
      }
      else if(p1->pid < lowest_bt->pid && p1->burst_time == currentMin)
      {
        lowest_bt=p1;
      }

    }
    p=lowest_bt;
    if (p != 0)
    {
      (p->count_contextswitch)++;
      // 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;

      swtch(&(c->scheduler), p->context);
      switchkvm();
      c->proc = 0;
    }
    release(&ptable.lock);
  }
}
```

The scheduler loops over the process table, and schedules the shortest job for execution. If the process is not runnable, it skips that entry. In any other case, it finds and schedules the shortest job for execution.

We have added two variables **burst_time** and **count_contextswitch** in the **struct proc** in **proc.h**.
We have initialised burst_time of each process to 0 in the allocproc function in **proc.c** and we can change this using the **set_burst_time** function.
We have removed the **yield() function call** in **trap function** in **trap.c file** because we no longer want context switch to happen due to time quantum expiry as was the case with Round Robin Scheduling algorithm.

Some theory behind implementation:

Context switching in xv6:

xv6 performs two kinds of context switches:
1. From a process's kernel thread to the current CPU's scheduler thread.
2. From the scheduler thread to a process's kernel thread.
xv6 never directly switches from one user-space process to another.
This happens by way of a user-kernel transition, a context switch to the scheduler, a context switch to a new process's kernel thread, and a 1 trap return.
Every xv6 process has its own kernel stack and register set.
Each CPU has a separate scheduler thread for use when it is executing the scheduler rather than any process's kernel thread.
Switching from one thread to another involves saving the old thread's CPU registers, and restoring previously-saved registers of the new thread.

Working of swtch function-:

swtch doesn't directly know about threads. it just saves and restores register sets, called contexts. When it is time for the process to give up the CPU, the process's kernel thread will call swtch to save its own context and return to the scheduler context. Each context is represented by a struct context*, a pointer to a structure stored on the kernel stack involved. Swtch takes two arguments: struct context **old and struct context *new. It pushes the current CPU register onto the stack and saves the stack pointer in *old. Then swtch copies new to esp, pops previously saved registers, and returns. Instead of following the scheduler into swtch, let's instead follow our user process back in.
Swtch returns on the scheduler's stack as though the scheduler's swtch had returned. The scheduler continues the for loop, finds a process to run, switches to it, and the cycle repeats.
We just saw that xv6 holds ptable.lock across calls to swtch: the caller of swtch must already hold the lock, and control of the lock passes to the switched-to code.

We implemented a new scheduler function.

Explanation of code-:

Both the CPUs are running this code concurrently and the code is safe as we have used locks whenever we are accessing any kernel data structures.
Three pointers to **struct proc** type named **p, p1** and **lowest_bt** have been created.
p is the pointer to the **desired process** i.e. the process we want to run. We have to assign the **process with the lowest burst time** to p. For **calculating the process with lowest burst time**, we have made a pointer **lowest_bt.**
We have created a pointer **c** to the **struct cpu** type and initialized **c->proc** to **0.**

This is a per-CPU process scheduler. **Scheduler never returns.(Hence, the never ending for loop)**
It loops, doing:
-- choose a process to run.
-- swtch to start running that process.
-- eventually that process transfers control via switchback to the scheduler.

Call to **sti() function enables interrupts** on this processor.

To maintain safety when using kernel data structure, we have acquired a lock using **acquire(&ptable.lock)** since a process that wants to give up the CPU must acquire the process

table lock **ptable.lock**, release any other locks it is holding, update its own state (proc->state), and then call sched.
We have initialized **lowest_bt** to null process.
We are maintaining **current_min** as the minimum burst time encountered till now.

for (p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)
We are running the for loop starting from the first process in the proc array (ptable.proc is the address of the first element of the array). It goes till the 64th process and each time p1 is incremented.
Our aim is to calculate the process with the shortest burst time that is currently in RUNNABLE state. If the process is not in RUNNABLE state we continue the for loop.
If the process is in RUNNABLE state, then we check if the burst time of the process is less than the currentMin, then we update lowest_bt as well as the currentMin, else if the burst_time of the current process equal to currentMin, then we check which process has smaller pid and we accordingly update the lowest process. Finally, we assign the address of the process pointed to by lowest_bt to p.
Now, p contains the process having the lowest burst time.
Since, now we have switched to this process, we increment the count_contextswitch variable.
We assign CPU to this process using c->proc = p.
Then it switches to the process's page table with switchuvm, marks the process as RUNNING, and then calls swtch to start running it.
After the process comes back to scheduler, the **kernel loads its memory(switchkvm();)**
The process finally **releases the lock** acquired.

The system call set_burst_time has a call to yield() so that the scheduler is called when burst time is set.
The above implementation perfectly simulates the Shortest Job First Scheduler handling all the corner cases.

Kernel data structures:
The **scheduler runs over the p-table** in order to loop through the burst times and searches for the shortest job to be scheduled.

Runtime complexity-:
The runtime complexity of our algorithm is **O(n)** since we traverse all the processes in the ptable to calculate the process with the minimum burst time present in the ready queue. Here, n is the number of processes.

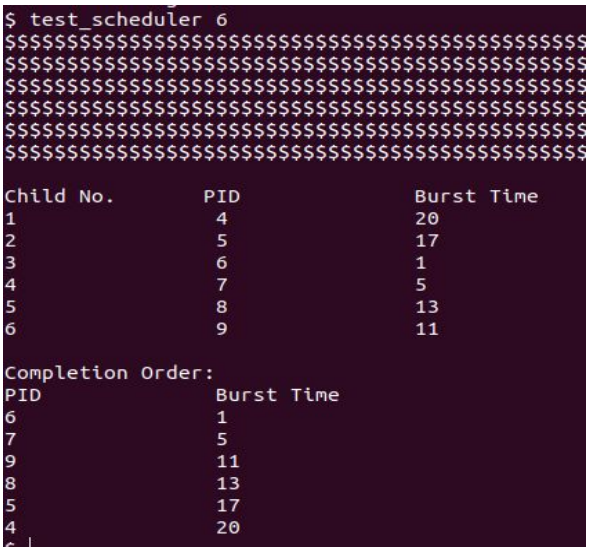
Corner cases:
For the processes having **equal burst times**, we have implemented FCFS scheduling. The process which arrived first(the one with lesser PID) is scheduled first. Also, the code is safe when run over multiple CPU cores as we have used locks when accessing kernel data structures. Every time ptable is accessed, acquire and release have been used to ensure good locking discipline.

Scheduler Testing-:
We have made two programs test_scheduler.c(for test case 3 and 4) and test_sched.c(for test case 1 and 2) to test our shortest job first scheduler.
We have to specify the number of childs to be created while using this system call on the xv6 console.
In the file **test_scheduler.c,** the function **random_computation executes a numerical calculation** and **therefore is a CPU-bound process.**
In the file **test_scheduler.c,** the function **random_io** executes **two I/O bound processes, printf, and sleep.**
In all the test cases below, output is significantly different as compared to the default round robin scheduler.
Note that the burst-time we set is not equal to the actual execution time.

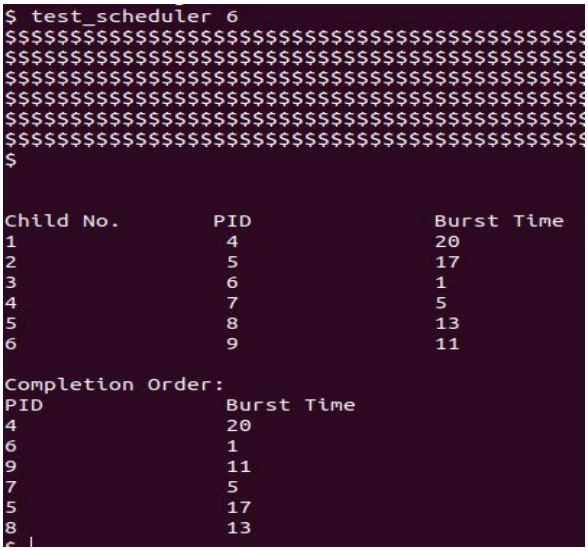
Test Case 1: (Only I/O bound processes with random burst times)

In this test case, we have created 6 processes using fork() system call. All the processes are I/O bound processes where we are printing $ sign on the console and sleep(3). We renamed the name of the file to test_sched.c after taking the screenshots. That is why, here in the screenshots, it is being displayed as test_scheduler. To test the scheduler function, type test_sched <no. of processes> on the xv6 console.

Output for Shortest Job First Scheduler.



Output for Round Robin Scheduler.



Shortest Job First Scheduler-:

In the output for the shortest job first scheduler, the processes are completed in the ascending order of burst times. Since the **random_io** function is the same for all processes, all the processes take the same time for executing the I/O operations. This is the reason that all the processes are executed in the desired ascending order of burst times. If there would have been any CPU bound process, then it may have been executed before the I/O bound process depending on the execution time of the I/O bound process.

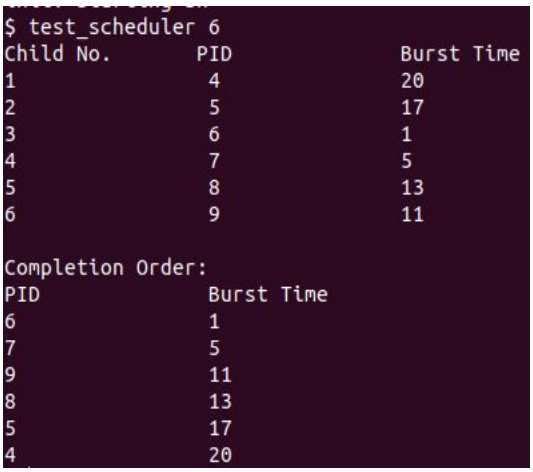
Default Round Robin Scheduler-:

The default round robin scheduler doesn't take burst times into account when scheduling the processes. Therefore, order of completion is random and not dependent on the burst times.

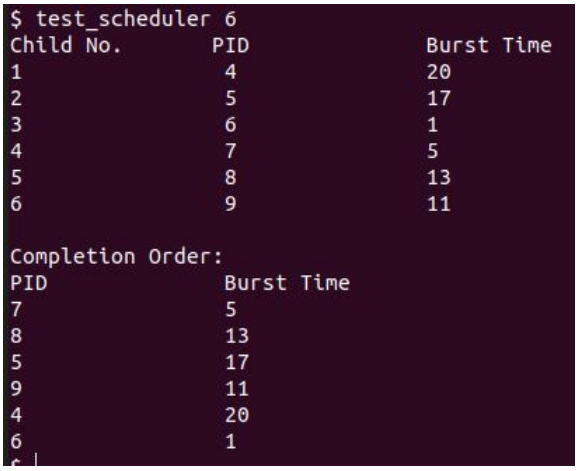
Test Case 2: (Only CPU bound processes with random burst times)

In this test case, we have created 6 processes using fork() system call. All the processes are CPU bound processes where we are performing ALU operations.

Output for Shortest Job First Scheduler.



Output for Round Robin Scheduler.



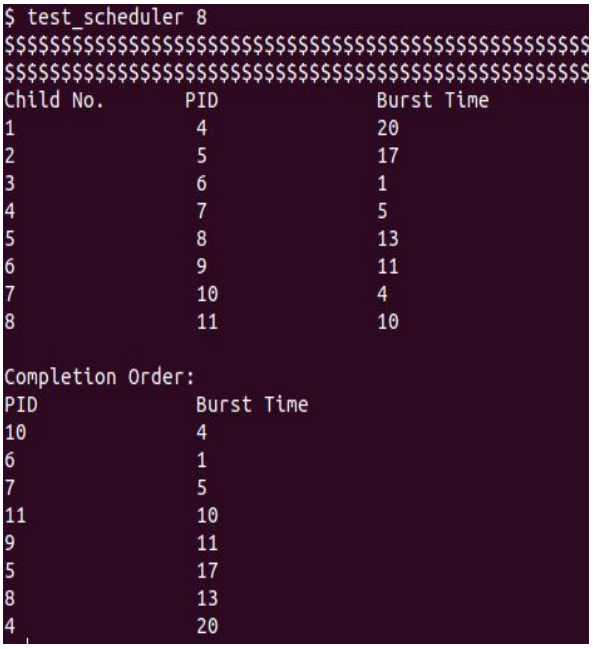
Shortest Job First Scheduler-:
In the output for the shortest job first scheduler, the processes are completed in the ascending order of burst times. Since the **random_compuation** function is the same for all processes, all the processes take the same time for executing the ALU operations. This is the reason that all the processes are executed in the desired ascending order of burst times.

Default Round Robin Scheduler-:
The default round robin scheduler doesn't take burst times into account when scheduling the processes. Therefore, order of completion is random and not dependent on the burst times.

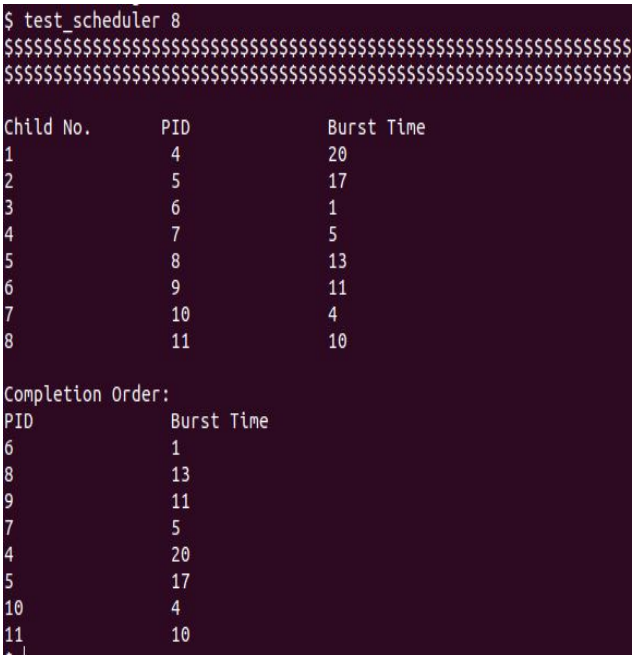
Test Case 3:(Both CPU and I/O bound processes with random burst times)

In this test case, we have created 8 child processes. Child number 3 and 5 are I/O bound and all other processes are CPU bound.
In the I/O operation, we are simply printing $ sign on the terminal and calling sleep(3).

Output for Shortest Job First Scheduler.



Output for Round Robin Scheduler.



Shortest Job First Scheduler-:
In the above test case, child No. 3 has a PID of 6 and burst time 1. Ideally, it should be the first one to get completed but as it is an I/O bound process, it takes some time to do the I/O operation. Meanwhile, the OS schedules Child No. 7 with PID 10 to the CPU and therefore it gets completed before the process having lowest burst time. Same is the case with Child N. 5 with burst time of 11 and it gets completed after child No. 2 which is having burst time of 17. We can easily see that the scheduler works correctly even when the processes block. The main observation here is that the ptable automatically pops the process from the running queue as soon as it requires an I/O operation to be done. It is pushed to the waiting queue and another process is scheduled to the running queue from the ready queue. This corner case is being handled by the ptable itself.

Default Round Robin Scheduler-:
The default round robin scheduler doesn't take burst times into account when scheduling the processes. Therefore, order of completion is random and not dependent on the burst times.

To test the scheduler function, type test_scheduler <no. of processes> on the xv6 console.

Test Case 4:(Both CPU and I/O bound processes with random burst times)

In this test case, we have created 16 child processes. Child number 3, 5, 10, 15 are I/O bound and all other processes are CPU bound.

Shortest Job First Scheduler-:

Child No. 3,5,10,15 with PIDs of 12,14,19,24 respectively are I/O bound processes. The order of completion if there were no I/O operations would have been:
12,21,19,20,16,13,18,23,24,17,15,14,25,11,22,10

But since PID 12,14,19,24 are I/O bound, the order of completion is-:
21,12,20,19,16,13,18,23,17,24,15,25,14,11,22,10
In this case also, it is evident that the scheduler works correctly even when the processes are blocked due to I/O operation.
The main observation here is that the ptable automatically pops the process from the running queue as soon as it requires an I/O operation to be done. It is pushed to the waiting queue and another process is scheduled to the running queue from the ready queue. This corner case is being handled by the ptable itself.

Default Round Robin Scheduler-:

The default round robin scheduler doesn’t take burst times into account when scheduling the processes. Therefore, order of completion is random and not dependent on the burst times.

To test the scheduler function, type test_scheduler <no. of processes> on the xv6 console.

Output for Shortest Job First Scheduler.

```
$ test_scheduler 16
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Child No.        PID              Burst Time
1                 10               20
2                 11               17
3                 12               1
4                 13               5
5                 14               13
6                 15               11
7                 16               4
8                 17               10
9                 18               6
10                19               3
11                20               3
12                21               2
13                22               18
14                23               7
15                24               8
16                25               13

Completion Order:
PID               Burst Time
21                2
12                1
20                3
19                3
16                4
13                5
18                6
23                7
17                10
24                8
15                11
25                13
14                13
11                17
22                18
10                20
$ |
```

Output for Round Robin Scheduler.

```
$ test_scheduler 16
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
$$$$$$$$$$$$$$$$$$$$$$$

$$$$$
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Child No.       PID             Burst Time
1               13              20
2               14              17
3               15              1
4               16              5
5               17              13
6               18              11
7               19              4
8               20              10
9               21              6
10              22              3
11              23              3
12              24              2
13              25              18
14              26              7
15              27              8
16              28              13

Completion Order:
PID             Burst Time
15              1
16              5
17              13
21              6
18              11
22              3
23              3
13              20
24              2
27              8
20              10
25              18
19              4
26              7
28              13
14              17
```

Running the patch-:

To run the patch file use the following command-:

patch -ruN -d <original_xv_folder> < g17.patch

It will ask some questions. Always type **n** answer to every question asked. The original folder will contain the modified changes.

BONUS Question (Included a separate patch in the zip file with name g17hybrid.patch)

Task-: To implement a hybrid of round robin and shortest job first algorithm.

Algorithm Details --

Following a hybrid scheduling algorithm. The smallest process is picked and executed for 2 quantums and its burst_time is decremented by the number of cycles in 1 quantum after which it is preempted and a new process with the smallest burst is scheduled by the scheduler. This is done repeatedly till all the processes have terminated (exhausted their burst time, or completed it's burst time in the processor).

Implementation Details --

We added another parameter to **struct proc**, named **cpucounter**. **This denotes the number of CPU cycles the process has been in the running state since it was initialized.**

We have implemented two functions in proc.c in order to implement the hybrid scheduler.

1. **dec_burstTime(): To decrease the burst time of the running process after every time-slice.**
2. **inc_cpucounter(): For every clock cycle that the process is in the running state, this function increments the cpucounter parameter by 1 clock cycle.**

```
void dec_burstTime(void)
{
  acquire(&ptable.lock);
  if(mycpu()->proc->burst_time < 0)
  {
    mycpu()->proc->burst_time = -1;
  }
  else
  {
    mycpu()->proc->burst_time -= QUANTA;
  }
  release(&ptable.lock);
}

int inc_cpucounter(void)
{
  int res;
  acquire(&ptable.lock);
  res = mycpu()->proc->cpucounter+1;
  mycpu()->proc->cpucounter = mycpu()->proc->cpucounter+1;
  release(&ptable.lock);
  return res;
}
```

Also, we defined the scheduler function, which is a hybrid of Shortest Job First and default round-robin. The algorithm chooses the process with the least burst time and executes the process for a fixed time quanta(in our case 2) till there is an interrupt called by trap.c when the time quanta expires. We have initialised cpucounter of every process to zero.

sysproc.c

```
int sys_inc_cpucounter(void)
{
  return inc_cpucounter();
}

int sys_dec_burstTime(void)
{
  dec_burstTime();
  return 29;
}
```

In **trap.c** we have changed the condition to include a check for when **cpucounter** becomes equal to **QUANTA** in which case we decrement the burst, reset **cpucounter** and call yield() to call scheduler again. In the default code, we check whether **inc_cpucounter()** has become equal to the time QUANTA. When it becomes equal to the time quanta, we call the **dec_burstTime** function to decrease the burst time and then the **yield()** function that schedules another process.

```
if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER && inc_cpucounter() == QUANTA)
{
  dec_burstTime();
  // cprintf("Interrupt %d with burst %d \n", myproc()->pid, myproc()->burst_time);
  myproc()->cpucounter=0;
  yield();
}
```

In the **param.h** file, we have declared the **QUANTA** as **2,** which is the minimum burst time in our test function. **This is the time quantum for the round-robin preemptive scheduling.**

```
#define QUANTA       2  // Time Quantum hybrid (SJF+RR). 2 is the min burst time.
```

Output-:
Note that the burst-time we set is not equal to the actual execution time.

In the test case shown, we have created 10 child processes. We have set the time quantum to 2. So, in this case first of all the scheduler picks the process with lowest burst time. So, the process with PID of 6 and burst time of 2 is scheduled and runs for 2 time quantums after which it finishes. Then, the process with PID 13 is scheduled and runs for 2 time quantums after which the time quantum expires and then the next process having PID 10 is scheduled on the processor. It also runs for 2 time quantums and is then preempted by the scheduler. In similar fashion, all the processes are scheduled and executed and finally we get the order of completion to be the ascending order of burst times.

```
$ test_scheduler 10
Child No.       PID             Burst Time
1               4               20
2               5               17
3               6               2
4               7               5
5               8               13
6               9               11
7               10              4
8               11              10
9               12              6
10              13              3

Completion Order:
PID             Burst Time
6               2
13              3
10              4
7               5
12              6
11              10
9               11
8               13
5               17
4               20
```

Running the patch-:

To run the patch file use the following command-:

patch -ruN -d <original_xv_folder> < g17hybrid.patch

It will ask some questions. Always type **n** answer to every question asked. The original folder will contain the modified changes.

SUBMISSION-----

1. **g17 folder contains all the files that were changed to implement the normal Shortest Job First Scheduler.**
2. **g17hybrid contains all the files that were changed to implement the hybrid scheduler.**
3. **g17.patch is the patch for the normal scheduler.**
4. **g17hybrid.patch is the patch for the hybrid scheduler.**

To test the scheduler function, type test_scheduler <no. of processes> on the xv6 console.