CS 344 Operating Systems Laboratory Assignment 2

Group Number 3

TEAM MEMBERS:

NAME	ROLL NUMBER
RITWIK GANGULY	180101067
PULKIT CHANGOIWALA	180101093
SAMAY VARSHNEY	180101097
UDANDARAO SAI SANDEEP	180123063

PART A:

In order to define our all system calls given in Part A in xv6, we changed some files mentioned below.

1) **syscall.h:**
   We added the below new system calls in syscall.h:
   #define SYS_getNumProc 22
   #define SYS_getMaxPid 23
   #define SYS_getProcInfo 24
   #define SYS_set_burst_time 25
   #define SYS_get_burst_time 26
   21 positions were already occupied by the inbuilt system calls in syscall.h.

2) **syscall.c:**
   We added the below pointers to in order to add our custom system call.
   [SYS_getNumProc]   sys_getNumProc,
   [SYS_getMaxPid]   sys_getMaxPid,
   [SYS_getProcInfo]   sys_getProcInfo,
   [SYS_set_burst_time]   sys_set_burst_time,
   [SYS_get_burst_time]   sys_get_burst_time,

   Then below function prototypes are added in syscall.c file to be called by system call numbers:
   extern int sys_getNumProc(void);
   extern int sys_getMaxPid(void);
   extern int sys_getProcInfo(void);
   extern int sys_set_burst_time(void);
   extern int sys_get_burst_time(void);

3) **sysproc.c:**
   Below system call functions are implemented in sysproc.c. All the below function definitions are given along with the function body in sysproc.c with explanations.
   int sys_getNumProc(void) {}
   int sys_getMaxPid(void) {}
   int sys_getProcInfo(void) {}
   int sys_set_burst_time(void) {}
   int sys_get_burst_time(void) {}

4) **usys.s:**
   For creating an interface for our user programs to access system call from user.h we added the following lines in usys.S.
   SYSCALL(getNumProc)
   SYSCALL(getMaxPid)
   SYSCALL(getProcInfo)
   SYSCALL(set_burst_time)
   SYSCALL(get_burst_time)

5) **user.h:**
   We added the below function declarations that the respective user programs will be calling in user.h for invoking system calls.
   int getNumProc(void);
   int getMaxPid(void);
   int getProcInfo(int pid, struct processInfo*);
   int set_burst_time(int);
   int get_burst_time(void);

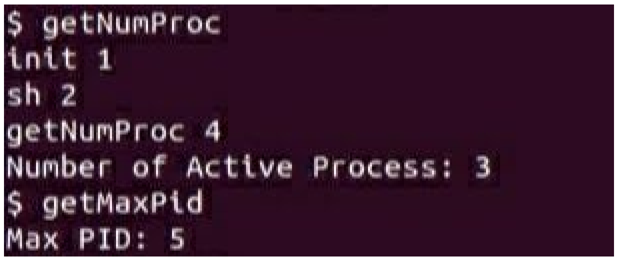
6) **proc.h:**
   Per process state is updated to include 2 more fields:

1) int numOfSwitches;	to count number of context switches
2) int burstTime;	to save burst time for the given process in seconds

7) **defs.h:**
   Below function definitions are declared in defs.h so that these become available to the kernel in the proc.c file.
   int getNumProc(void);
   int getMaxPid(void);
   int getProcInfo(int , struct processInfo*);
   int set_burst_time(int);
   int get_burst_time(void);

8) **proc.c:**
   The above functions are defined in proc.c along with their function body so that the kernel can access and execute the code in prog.c.

We saved C programs corresponding to all system calls (with the names of their respective system calls) inside the source code directory of xv6 operating system. We edit the MakeFile and added the necessary changes under UPROGS and EXTRA flags in MakeFile.

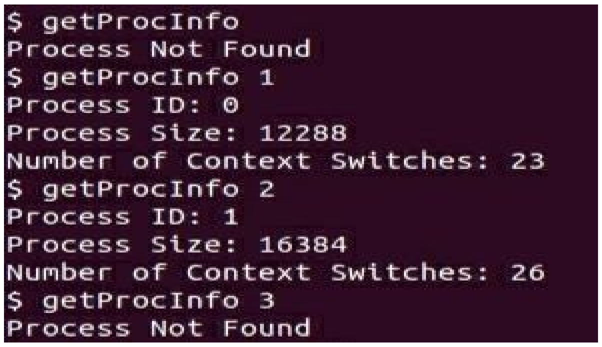


```
$ getNumProc
init 1
sh 2
getNumProc 4
Number of Active Process: 3
$ getMaxPid
Max PID: 5
```

Output of Part A section 1

getNumProc call outputs all the active system processes along with their Process IDs and total number of active processes.

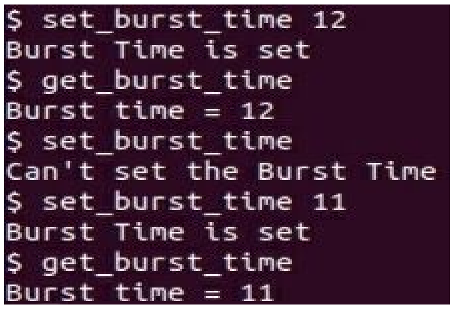
getMaxPid outputs the maximum Process Id.



```
$ getProcInfo
Process Not Found
$ getProcInfo 1
Process ID: 0
Process Size: 12288
Number of Context Switches: 23
$ getProcInfo 2
Process ID: 1
Process Size: 16384
Number of Context Switches: 26
$ getProcInfo 3
Process Not Found
```

Output of Part A section 2

If getProcInfo is called with no arguments or with a process Id not present in the system, it returns “Process Not Found” as no process Id is given. When called with Process Id, it returns process size, no of context switches and process Id - 1.



```
$ set_burst_time 12
Burst Time is set
$ get_burst_time
Burst time = 12
$ set_burst_time
Can't set the Burst Time
$ set_burst_time 11
Burst Time is set
$ get_burst_time
Burst time = 11
```

Output of Part A section 3

When set_burst_time is called with no arguments, “Can’t set burst time” is displayed as shown. But when called with parameter (burst time), Burst Time is set for the currently running process. And it is verified by calling get_burst_time.

PART B:

The current scheduler in xv6 is an unweighted round robin scheduler. Hence we modified the scheduler and some other functions in proc.c and some files for Shortest Job First and Hybrid (SJF + RR) scheduling.
Below are some changes and scheduling algorithm implementations.
We have used flags to switch between algorithms.
To run the kernel and for running the scheduler() function as SJF, Hybrid and DEFAULT, use below commands:

- **SJF:** make qemu SCHEDFLAG=SJF
- **Hybrid Scheduling (HBSJF):** make qemu SCHEDFLAG=HBSJF
- **DEFAULT Scheduling (RR):** “make qemu” or “make qemu SCHEDFLAG=DEFAULT”

1) param.h:

We changed the number of CPUs to 1 in the param.h file.

```
#define NCPU 1                // maximum number of CPUs
```

2) trap.c:

We changed the second last conditional statement of trap.c which was causing context switching to run our both scheduling algorithms.

```
#ifdef SJF              // when SJF will run
        // no context switching
#else
#ifdef DEFAULT          // when DEFAULT will run
        // statement is given in trap.c
#else
#ifdef HBSJF            // when Hybrid will run
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER && time_checker())
 //added a time_checker() in default if condition
  yield();
#endif
#endif
#endif
```

3) proc.c:

int TimeQuanta = 2000 is added for setting default value which will get overwritten in Hybrid scheduling.

in the allocproc() function, below lines were added so that when a process changes its state from UNUSED to EMBRYO, these added fields get their default values.

```
        p->burstTime = 0;          // default value for burst time
        p->numOfSwitches = 0;      // initial number of context switches = 0
        p->alreadyRun = 0;         // since it was unused, it didn't ran previously
        p->runningTime = 0;        // time for which it has runned = 0
```

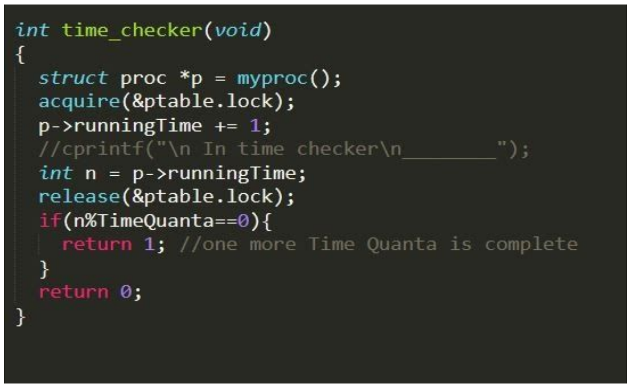
These fields are also declared in **proc.h.**

In the scheduler() function, we changed the code to make it run for both SJF and Hybrid.
Algorithm:
Let p be the pointer to a process.
For all the processes in the table do the following:
#ifdef SJF
Find the process from the process table having lowest burst time and setting p to that process.
#else
#ifdef HBSJF
1) Find the minimum burst time process by running a for loop.
2) Find the lowest burst time process which is not being run in the current round. If all processes are run one time in a round then we start with the next round.
3) If no such process is found in round 2), start the next round with the process found in 1).
4) Set p to the found process in the above steps.
#endif
#endif
Then Switch to the chosen process p and change its state to RUNNING and it's alreadyRun field value to 1.
End of algorithm.

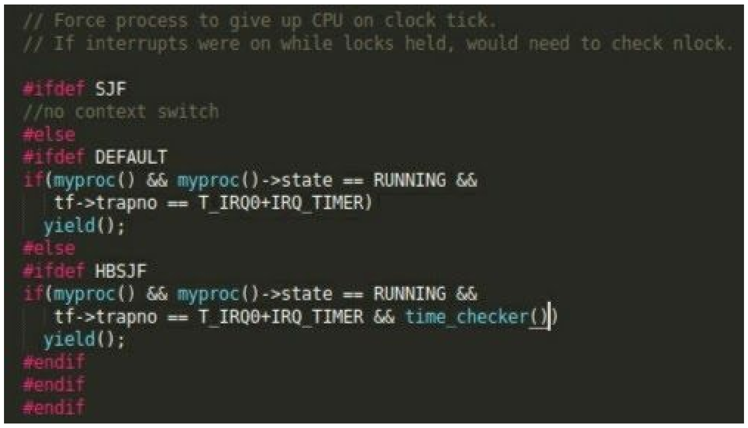
Both the **SJF and Hybrid have a worst case time complexity of O(N)** for picking one process as for finding a process with minimum burst time, we are going through all the processes in the table.
To print each process status we used a function named ps().

For implementing a hybrid scheduler we also built a function **time_checker()** {check in the image attached}. This function checks whether the given process has run for time equal to TimeQuanta. It increases the process variable **runningTime**(we added this variable in struct proc of proc.h}. And whenever the runningTime is multiple of TimeQuanta it returns 1.

This returned value is used in trap.c where we checked this value in condition checker and yield() the process accordingly. { yielding the process means changing the state from RUNNING to RUNNABLE }



```
int time_checker(void)
{
  struct proc *p = myproc();
  acquire(&ptable.lock);
  p->runningTime += 1;
  //cprintf("\n In time checker\n________");
  int n = p->runningTime;
  release(&ptable.lock);
  if(n%TimeQuanta==0){
    return 1; //one more Time Quanta is complete
  }
  return 0;
}
```



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

#ifdef SJF
//no context switch
#else
#ifdef DEFAULT
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
#else
#ifdef HBSJF
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER && time_checker())
  yield();
#endif
#endif
#endif
```

4) sysproc.c:

2 more system calls were added by changing the appropriate files (like that we did in part A) given below.
int sys_yield(void) {}
int sys_ps(void) {}

- **yield()** is used for changing the state of a process from RUNNING to RUNNABLE. yield() was already defined in proc.c() we only made it a system call to use it in test_scheduler files.

- **ps()** system call is defined in proc.c(). This system call is basically used to print process details. It's name, pid, state and burst number. It has no direct link with the assignment or scheduler but only used for debugging purposes.

TEST CASES:

There are 2 test cases for both Shortest Job First (SJF) scheduling and Hybrid SJF (HBSJF). In the first test case file **Test_scheduler_one.c**, the 2 test cases are present and we allocated burst time to each fork of the process in the *decreasing order* in this case. While in the second test case file **Test_scheduler_two.c,** the 2 test cases are present and we allocated burst time to each fork of the process *in the random order* in this case.

In each test case file we have included two test cases, **one with only cpu bound process and second with half cpu bound and half I/O bound processes.** In the second test case, for odd numbered process Id's, we have made them CPU bound processes while for even numbered process Id's, we have made them I/O bound processes.

We introduced delays for making processes bounded. We used delay() function for making processes CPU bound and io_delay() function for making processes I/O bound.

Process creation in test cases: We used the **fork()** function to generate child processes. Then we stored the PIDs of created processes in an array pid[] in the order in which their fork is completed.

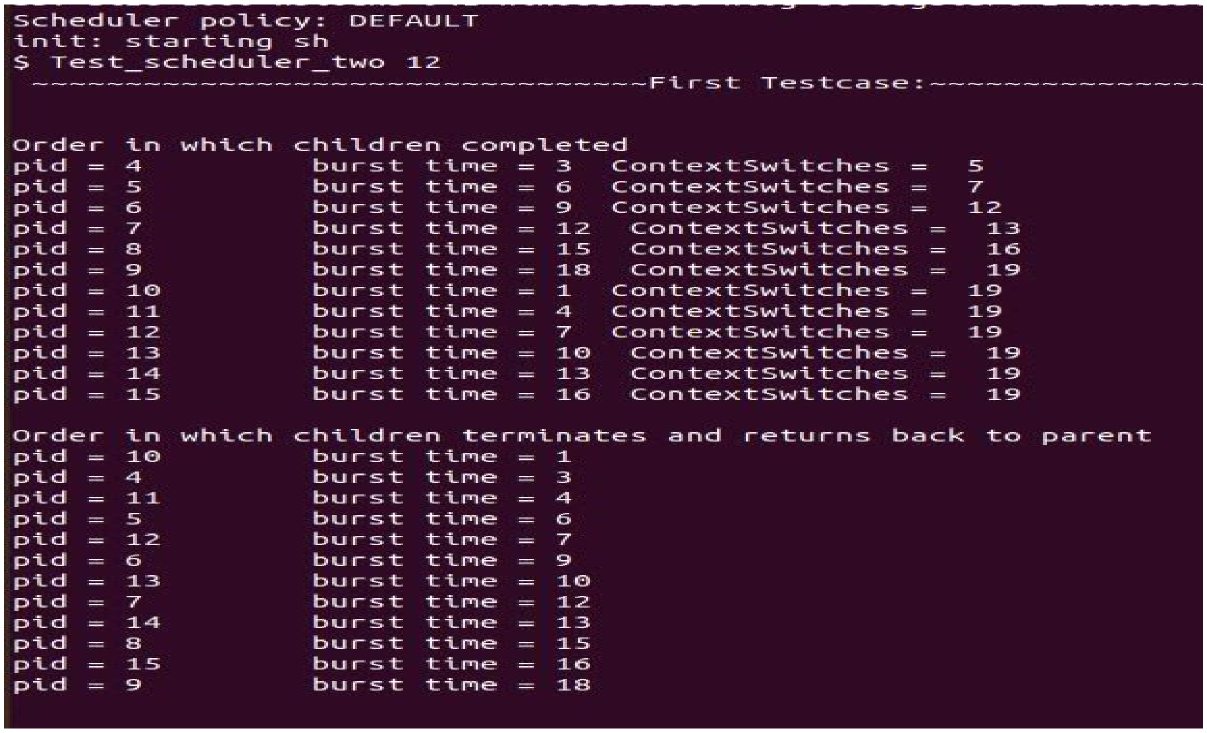
Then we stored the termination order of processes in another array namely rets[] using function wait(). wait() returns pid of the process when it is terminated. Then we printed both the completion and termination order of the processes.

Here is the **link to google drive** which contains the output screenshots of both the test cases of default, SJG, Hybrid cases.
Link: https://drive.google.com/drive/folders/1aMlkAtZxb_xTImzoSWW3jq6UPJIu2NmW

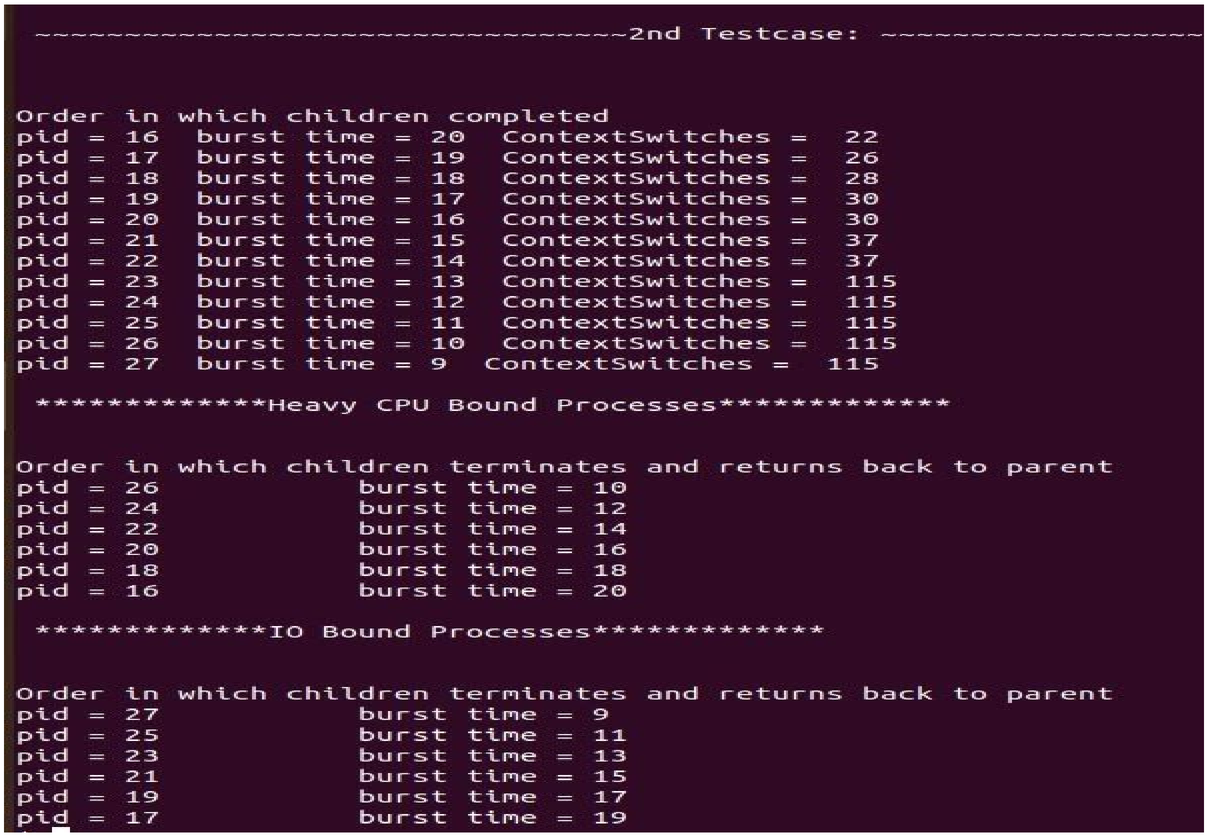
In the below images, the example of **test case file 2** is shown. We used 12 processes in all the cases and created their children using fork(). The burst times are in decreasing order of process Ids. We used wait() to store the order of termination of children.

1) DEFAULT CASE

In default mode we have context switches **more than or equal to one** whereas termination order is not fixed. But it is more or less in order of increasing **actual** burst time taken by the process as preempted round robin scheduling is followed.

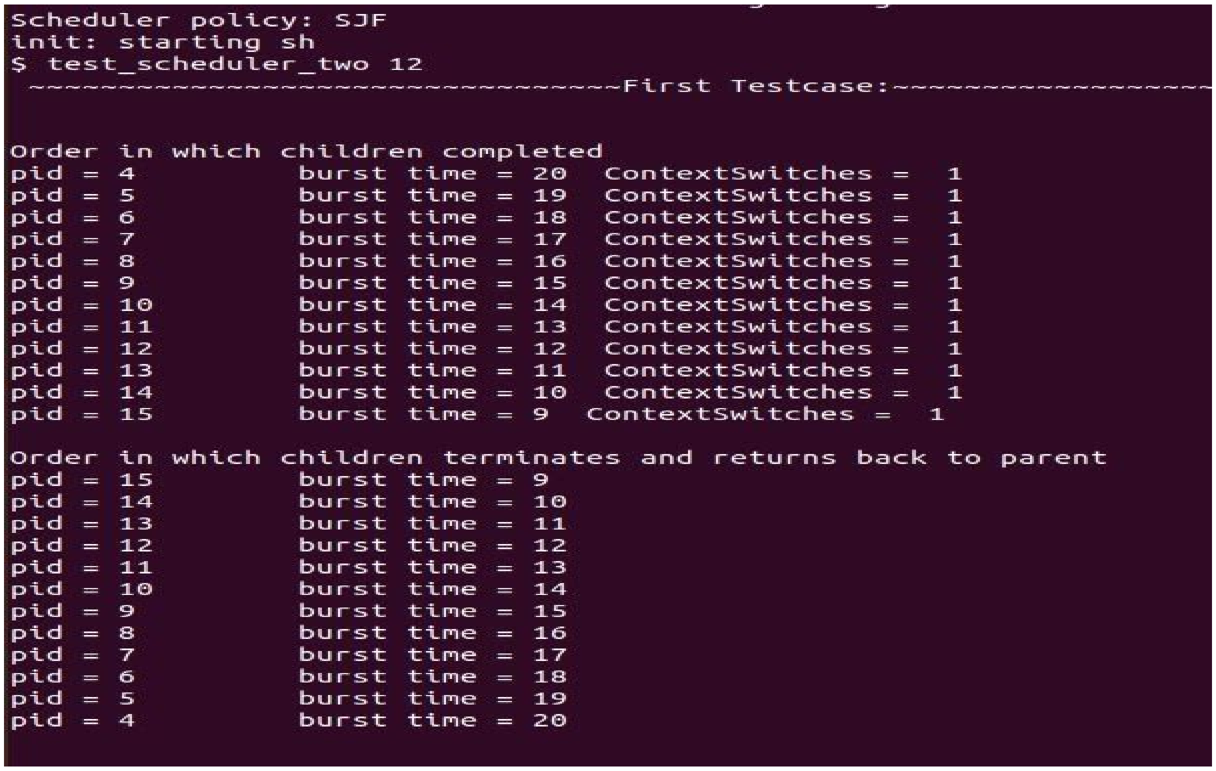


```
Scheduler policy: DEFAULT
init: starting sh
$ Test_scheduler_two 12
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~First Testcase:~~~~~~~~~~~~~~

Order in which children completed
pid = 4         burst time = 3  ContextSwitches =  5
pid = 5         burst time = 6  ContextSwitches =  7
pid = 6         burst time = 9  ContextSwitches =  12
pid = 7         burst time = 12  ContextSwitches =  13
pid = 8         burst time = 15  ContextSwitches =  16
pid = 9         burst time = 18  ContextSwitches =  19
pid = 10        burst time = 1  ContextSwitches =  19
pid = 11        burst time = 4  ContextSwitches =  19
pid = 12        burst time = 7  ContextSwitches =  19
pid = 13        burst time = 10  ContextSwitches =  19
pid = 14        burst time = 13  ContextSwitches =  19
pid = 15        burst time = 16  ContextSwitches =  19

Order in which children terminates and returns back to parent
pid = 10        burst time = 1
pid = 4         burst time = 3
pid = 11        burst time = 4
pid = 5         burst time = 6
pid = 12        burst time = 7
pid = 6         burst time = 9
pid = 13        burst time = 10
pid = 7         burst time = 12
pid = 14        burst time = 13
pid = 8         burst time = 15
pid = 15        burst time = 16
pid = 9         burst time = 18
```

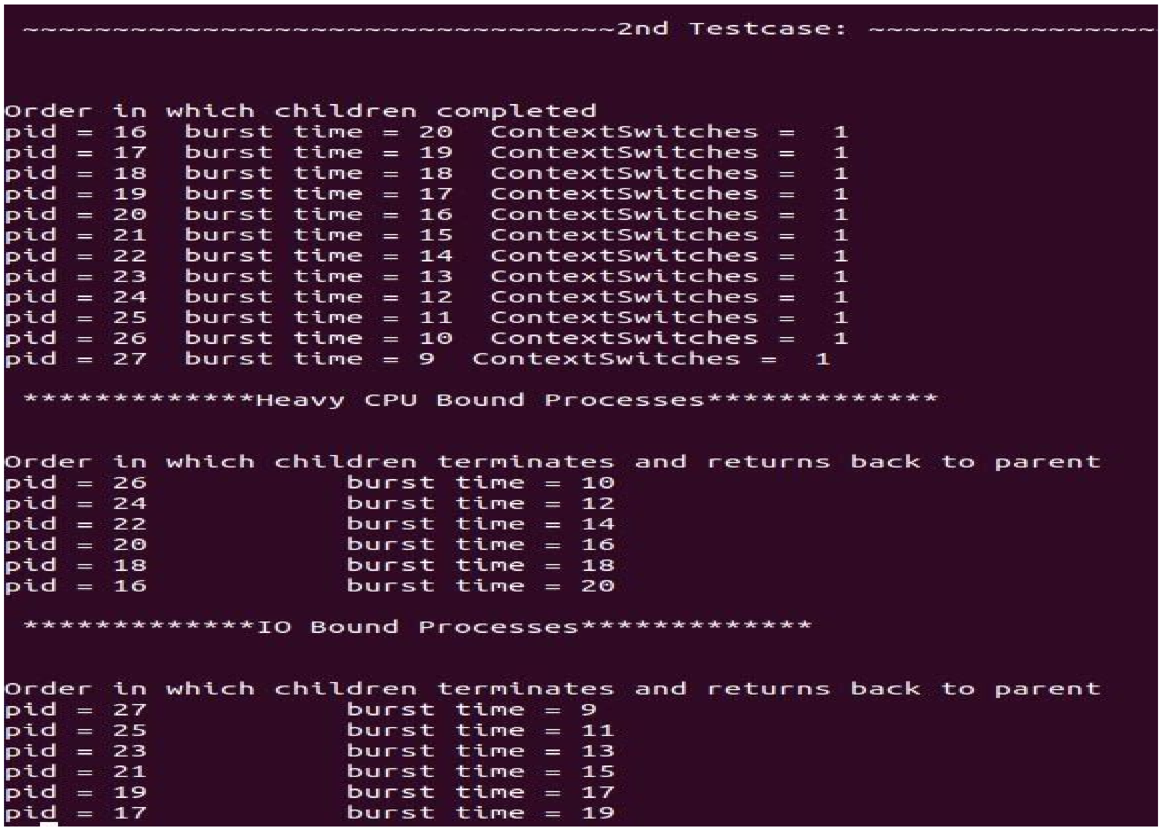


```
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~2nd Testcase: ~~~~~~~~~~~~~~~~~~


Order in which children completed
pid = 16  burst time = 20  ContextSwitches =  22
pid = 17  burst time = 19  ContextSwitches =  26
pid = 18  burst time = 18  ContextSwitches =  28
pid = 19  burst time = 17  ContextSwitches =  30
pid = 20  burst time = 16  ContextSwitches =  30
pid = 21  burst time = 15  ContextSwitches =  37
pid = 22  burst time = 14  ContextSwitches =  37
pid = 23  burst time = 13  ContextSwitches =  115
pid = 24  burst time = 12  ContextSwitches =  115
pid = 25  burst time = 11  ContextSwitches =  115
pid = 26  burst time = 10  ContextSwitches =  115
pid = 27  burst time = 9  ContextSwitches =  115

 *************Heavy CPU Bound Processes*************

Order in which children terminates and returns back to parent
pid = 26           burst time = 10
pid = 24           burst time = 12
pid = 22           burst time = 14
pid = 20           burst time = 16
pid = 18           burst time = 18
pid = 16           burst time = 20

 *************IO Bound Processes*************

Order in which children terminates and returns back to parent
pid = 27           burst time = 9
pid = 25           burst time = 11
pid = 23           burst time = 13
pid = 21           burst time = 15
pid = 19           burst time = 17
pid = 17           burst time = 19
```

2) SHORTEST JOB FIRST (SJF)



```
Scheduler policy: SJF
init: starting sh
$ test_scheduler_two 12
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~First Testcase:~~~~~~~~~~~~~~~~~

Order in which children completed
pid = 4          burst time = 20  ContextSwitches =  1
pid = 5          burst time = 19  ContextSwitches =  1
pid = 6          burst time = 18  ContextSwitches =  1
pid = 7          burst time = 17  ContextSwitches =  1
pid = 8          burst time = 16  ContextSwitches =  1
pid = 9          burst time = 15  ContextSwitches =  1
pid = 10         burst time = 14  ContextSwitches =  1
pid = 11         burst time = 13  ContextSwitches =  1
pid = 12         burst time = 12  ContextSwitches =  1
pid = 13         burst time = 11  ContextSwitches =  1
pid = 14         burst time = 10  ContextSwitches =  1
pid = 15         burst time = 9  ContextSwitches =  1

Order in which children terminates and returns back to parent
pid = 15         burst time = 9
pid = 14         burst time = 10
pid = 13         burst time = 11
pid = 12         burst time = 12
pid = 11         burst time = 13
pid = 10         burst time = 14
pid = 9          burst time = 15
pid = 8          burst time = 16
pid = 7          burst time = 17
pid = 6          burst time = 18
pid = 5          burst time = 19
pid = 4          burst time = 20
```

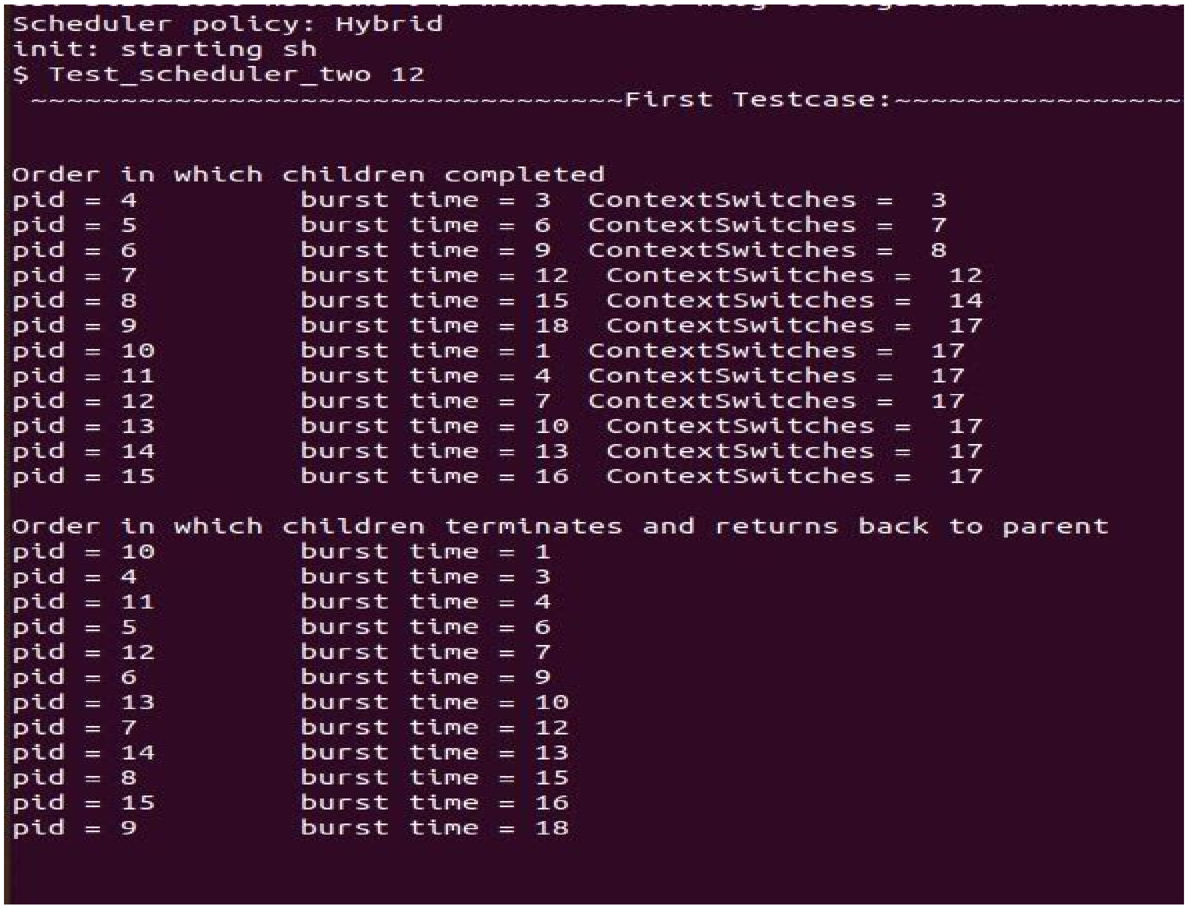


```
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~2nd Testcase: ~~~~~~~~~~~~~~~


Order in which children completed
pid = 16  burst time = 20  ContextSwitches =  1
pid = 17  burst time = 19  ContextSwitches =  1
pid = 18  burst time = 18  ContextSwitches =  1
pid = 19  burst time = 17  ContextSwitches =  1
pid = 20  burst time = 16  ContextSwitches =  1
pid = 21  burst time = 15  ContextSwitches =  1
pid = 22  burst time = 14  ContextSwitches =  1
pid = 23  burst time = 13  ContextSwitches =  1
pid = 24  burst time = 12  ContextSwitches =  1
pid = 25  burst time = 11  ContextSwitches =  1
pid = 26  burst time = 10  ContextSwitches =  1
pid = 27  burst time = 9  ContextSwitches =  1

 *************Heavy CPU Bound Processes*************


Order in which children terminates and returns back to parent
pid = 26            burst time = 10
pid = 24            burst time = 12
pid = 22            burst time = 14
pid = 20            burst time = 16
pid = 18            burst time = 18
pid = 16            burst time = 20

 *************IO Bound Processes*************


Order in which children terminates and returns back to parent
pid = 27            burst time = 9
pid = 25            burst time = 11
pid = 23            burst time = 13
pid = 21            burst time = 15
pid = 19            burst time = 17
pid = 17            burst time = 19
```

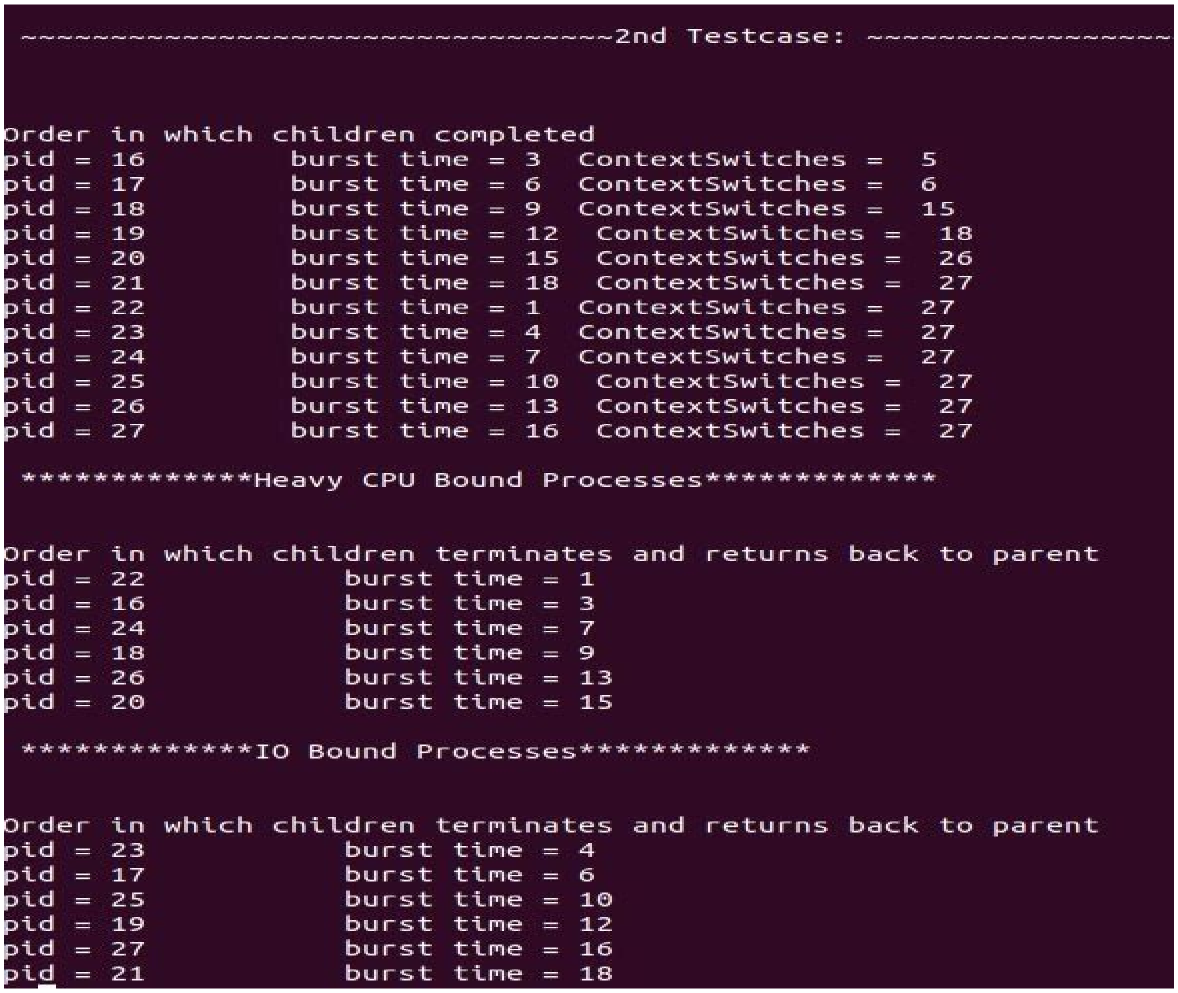
In the SJF mode, termination **is strictly in the order of increasing burst time** as we schedule the process with minimum burst time first.
Also context switch is only 1 for all the process as we don't preempt the process in the middle of execution. Once it's scheduled it gets completed.

3) HYBRID SHORTEST JOB FIRST (HBSJF)

It takes into account the shortest job first and time quantum for preemption of processes. In Hybrid we schedule the process with the shortest burst time first but we context switch it after a certain time quanta thus the process still ends in increasing order of time quanta but it has a number of context switches greater than one.



```
Scheduler policy: Hybrid
init: starting sh
$ Test_scheduler_two 12
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~First Testcase:~~~~~~~~~~~~~~~~


Order in which children completed
pid = 4          burst time = 3   ContextSwitches =  3
pid = 5          burst time = 6   ContextSwitches =  7
pid = 6          burst time = 9   ContextSwitches =  8
pid = 7          burst time = 12   ContextSwitches =  12
pid = 8          burst time = 15   ContextSwitches =  14
pid = 9          burst time = 18   ContextSwitches =  17
pid = 10         burst time = 1   ContextSwitches =  17
pid = 11         burst time = 4   ContextSwitches =  17
pid = 12         burst time = 7   ContextSwitches =  17
pid = 13         burst time = 10   ContextSwitches =  17
pid = 14         burst time = 13   ContextSwitches =  17
pid = 15         burst time = 16   ContextSwitches =  17

Order in which children terminates and returns back to parent
pid = 10         burst time = 1
pid = 4          burst time = 3
pid = 11         burst time = 4
pid = 5          burst time = 6
pid = 12         burst time = 7
pid = 6          burst time = 9
pid = 13         burst time = 10
pid = 7          burst time = 12
pid = 14         burst time = 13
pid = 8          burst time = 15
pid = 15         burst time = 16
pid = 9          burst time = 18
```



```
 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~2nd Testcase: ~~~~~~~~~~~~~~~~


Order in which children completed
pid = 16        burst time = 3  ContextSwitches =  5
pid = 17        burst time = 6  ContextSwitches =  6
pid = 18        burst time = 9  ContextSwitches =  15
pid = 19        burst time = 12  ContextSwitches =  18
pid = 20        burst time = 15  ContextSwitches =  26
pid = 21        burst time = 18  ContextSwitches =  27
pid = 22        burst time = 1  ContextSwitches =  27
pid = 23        burst time = 4  ContextSwitches =  27
pid = 24        burst time = 7  ContextSwitches =  27
pid = 25        burst time = 10  ContextSwitches =  27
pid = 26        burst time = 13  ContextSwitches =  27
pid = 27        burst time = 16  ContextSwitches =  27

 *************Heavy CPU Bound Processes*************


Order in which children terminates and returns back to parent
pid = 22           burst time = 1
pid = 16           burst time = 3
pid = 24           burst time = 7
pid = 18           burst time = 9
pid = 26           burst time = 13
pid = 20           burst time = 15

 *************IO Bound Processes*************


Order in which children terminates and returns back to parent
pid = 23           burst time = 4
pid = 17           burst time = 6
pid = 25           burst time = 10
pid = 19           burst time = 12
pid = 27           burst time = 16
pid = 21           burst time = 18
```