OS 344 Assignment 2

Group Number: C17
Prakhar Pandey: 200101081
Pratham Pekamwar: 200101087
Aadi Aarya Chandra: 200101002

Part A

Functions Implemented

1. `getNumProc` : This functions goes through the `ptable` that contains all the PCBs (*Process Control Blocks*) and gives the total number of active processes.

   > *Note:* Ptable lock has been used while the function is counting the number of processes. This is done to avoid inaccurate counting caused by a change in the ptable during the counting process.

2. `getMaxPid` : This method returns the highest PID among all presently active processes in the `ptable`.

   > *Note*: Ptable lock has been used here too. This is to prevent the corner case that while we are performing the getMaxPid method, a new process with higher process gets entered into the Ptable, and thus the output is no longer correct.

3. `getProcInfo` : Copies the parent's PID, process size, and number of context switches into a buffer (instance of `struct processInfo`) and outputs this information. The number of context switches is now stored in a data member (`int numContextSwitchesIn`) in the PCB structure that is provided in `struct proc` in *proc.h*.

4. `set_burst_time` : Sets the caller process's burst time to the input/argument value. In the process structure (`struct proc`), a data member (`int burstTime`) was added to hold the burst time.

5. `get_burst_time` : Returns burst time of the calling process.

Files Modified

1. `Makefile` : Before our xv6 OS source code is ready for compilation, the Makefile must be changed. The test user application is now in UPROGS, and the corresponding source C file is in EXTRAS.
2. `test.c` : The user space program's source code, that contains the functions mentioned above.
3. `user.h` : This file contains the system call declarations that a user process can utilise. Because the functions indicated above are implemented as system calls, the declarations were inserted here.
4. `usys.S` : The assembly code for the system call to be implemented. Contains the move instruction to the correponding instruction portion of the System Call followed by the `int` interrupt signal. The above implemented function are also added to the system call assembly template `SYSCALL()`
5. `syscall.h` : Contains the system call numbers for specifying the type of interrupt or the argument of `int` called in `usys.S`. Numbers 22 - 26 are for the above implemented functions.
6. `syscall.c` : Contains the `syscall(void)` method that is common for all the system calls. Specific system calls are implemented using the array of functions `int (*syscalls[])(void)`. The indexes are used from `syscall.h` while the wrapper functions are provided in `sysproc.c`.
7. `sysproc.c` : Added the wrapper functions as mentioned above for the implemented functions.
8. `proc.h` : Contains the structure of PCB in `struct proc`. Added `int numContextSwitchesIn`, `int burstTime` for implementation of the given functions.
9. `proc.c` : Contains the definitions regarding process management of the operating system. The main definition of the functions to be implemented are provided in this file since all the required funcions belong to the process management domain.

10. `defs.h` : This file is included in `proc.c`. This links the declarations in `user.h` to the definitions in `proc.c`

Testing the working of the functions

C code

```c
#include "user.h"
#define stdout 1

int main()
{
    printf(stdout, "The total number of processes are: %d\n", getNumProc());
    printf(stdout, "Maximum Process ID: %d\n", getMaxPid());                    // make better test cases
    int curProcID = getpid();
    printf(stdout, "This process has process ID: %d\n", curProcID);

    struct processInfo *input = (struct processInfo* )malloc(sizeof(struct processInfo));
    input->ppid = 0;
    input->psize = 0;
    input->numberContextSwitches = 0;
    printf(stdout, "Getting Process Information...\n");
    getProcInfo(curProcID, input);
    printf(stdout, "Information Retrieval Complete\n");
    printf(stdout, "Parent ID: %d\t Size: %d\nNumber of times this process was context switched in: %d\n", input->ppid, input->psize,
    input->numberContextSwitches);

    printf(stdout, "Burst time of this process (pid = %d) initially (should be 0): %d\n", curProcID, get_burst_time());
    printf(stdout, "Setting burst time of this process to 14...\t");
    set_burst_time(14);
    printf(stdout, "Success!\n");
    printf(stdout, "Current burst time of this process (pid = %d) is: %d\n", curProcID, get_burst_time());

    printf(stdout, "Setting burst time of this process to 20...\t");
    set_burst_time(20);
    printf(stdout, "Success!\n");
    printf(stdout, "Current burst time of this process (pid = %d) is: %d\n", curProcID, get_burst_time());
    exit();
}
```

output

```
SeaBIOS (version 1.15.0-1)


iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00



Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ test
The total number of processes are: 3
Maximum Process ID: 3
This process has process ID: 3
Getting Process Information...
Information Retrieval Complete
Parent ID: 2     Size: 45056
Number of times this process was context switched in: 9
Burst time of this process (pid = 3) initially (should be 0): 0
Setting burst time of this process to 14...     Success!
Current burst time of this process (pid = 3) is: 14
Setting burst time of this process to 20...     Success!
Current burst time of this process (pid = 3) is: 20
$
```

Conclusion: The test file calls the functions getNumProc, getMaxPid first, and the corresponding output can be seen as 3,3 respectively. Then, we fetch the process ID of the test process itself, and use that id to retrieve the process information using getProcInfo method. Then, we set the burst time of the test process to 14, and then 20 using the set_burst_time function. And later check the success of the funciton using the get_burst_time function.

PART B

Shortest Job First Scheduling

Implementation

- The implementation of shortest job first involves finding the process with smallest burst time. This is done using a basic for loop. The for loop accounts for a complexity **O(n)** where n is the number of runnable processes.

```
//choose the process with minimum burst time.
int minBurstTime = 1000000;
struct proc *minBurstProc = 0;

//iterator variable p1
struct proc *p1;
for (p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++)
{
  if (p1->state != RUNNABLE)
    continue;
  if (p1->burstTime < minBurstTime)
  {
    minBurstTime = p1->burstTime;
    minBurstProc = p1;
  }
}
```

- The yield() method was added to the function sys set burst time() in the file sysproc.c to give up the CPU for one scheduling round.

```
int set_burst_time(int n)
{
  int ans = -1;
  acquire(&ptable.lock);
  struct cpu* c = mycpu();
  if(c->proc->state != RUNNING) ans = -1;
  else{
    ans = 0;
    c->proc->burstTime = n;
  }
  release(&ptable.lock);
  //return back to this process after round of scheduler
  yield();
  return ans;
}
```

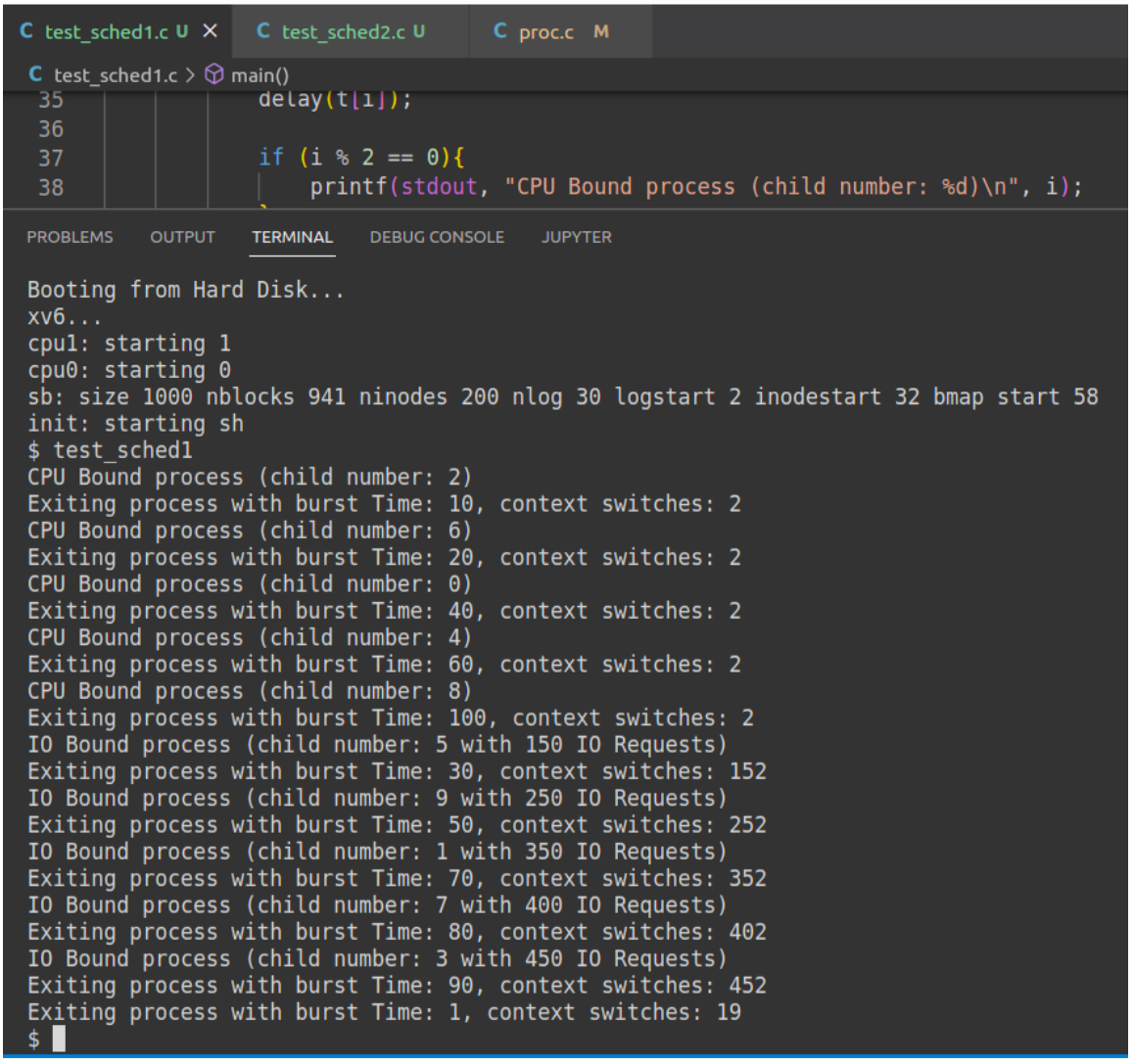
- Modified trap.c so as to make the scheduling non-pre emptive

```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to chec
// if(myproc() && myproc()->state == RUNNING &&
//    tf->trapno == T_IRQ0+IRQ_TIMER)
//   yield();
```

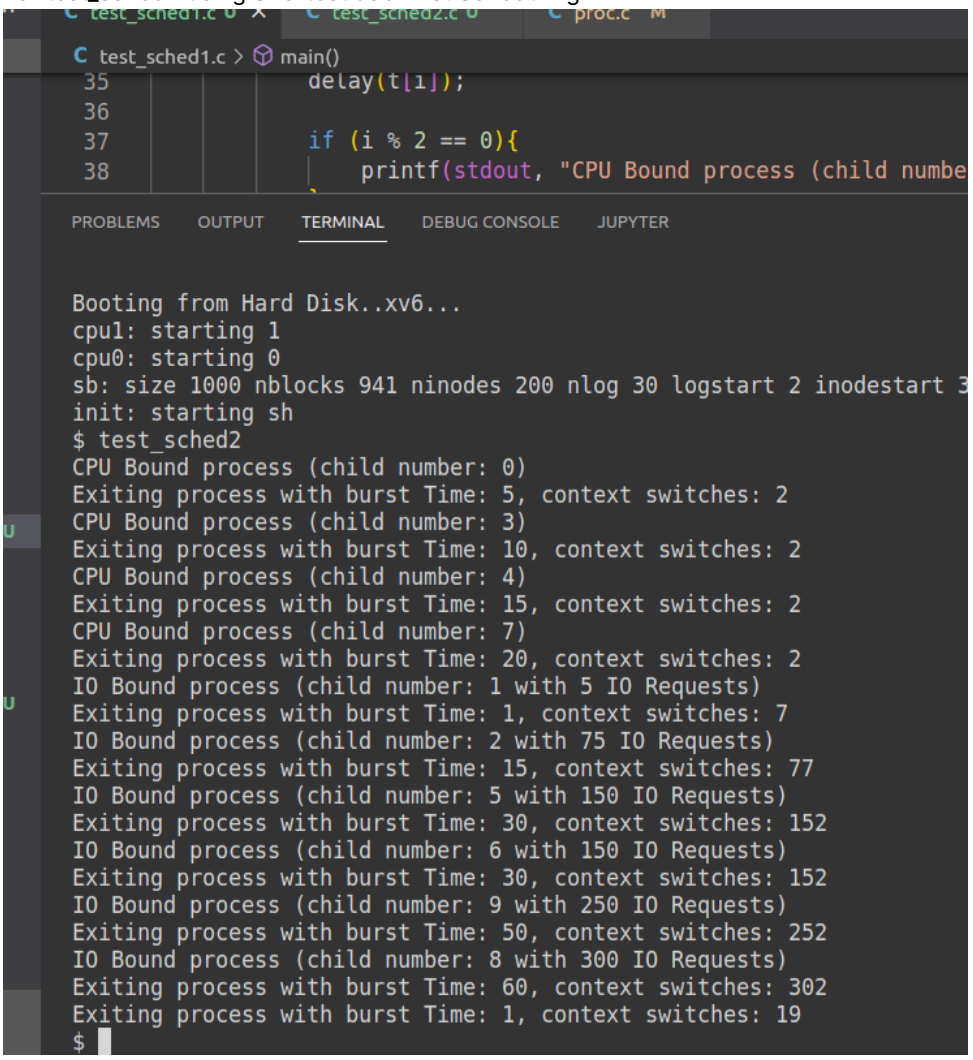
Testing

> OUTPUTS

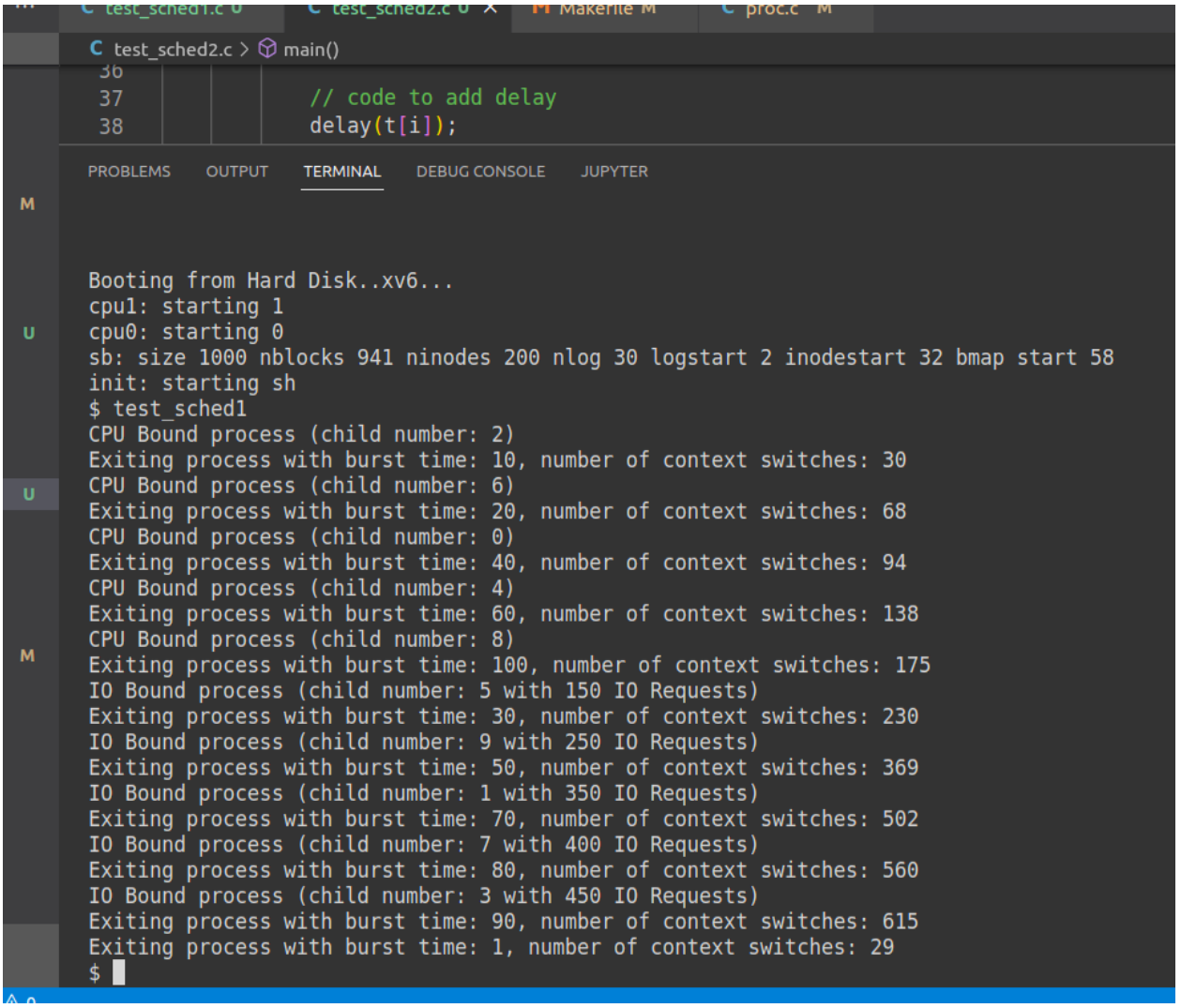
1. For test_sched1 on Shortest job first



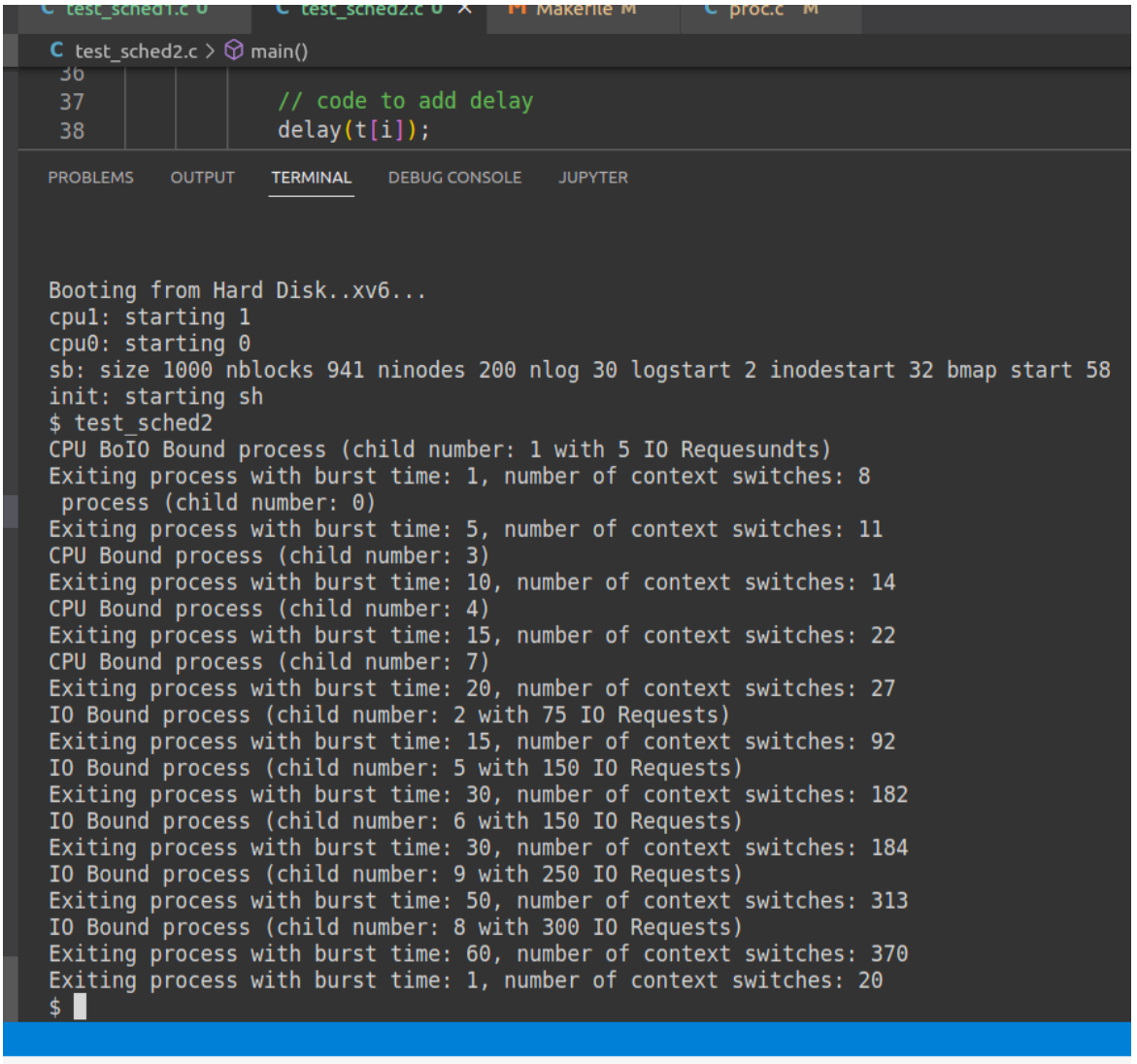
2. For test_sched2 using Shortest Job First Scheduling



3. test_sched1 on round robin



4. Test_sched 2 on Round Robin



> Comparison and Analysis

Structure of test case

We consider an array of burst time. The parent test case process then forks 10 child test process. The burst time for each is set using the set_burst_time method from the array of burst time.
The odd number child are made to mimic the I/O operations. Since while waiting for an I/O the process can be treated as sleeping, sleep is used for the simulating I/O. Different processes also occupy CPU and I/O for different duration, depending on their index. Finally, the exit() call has been modified to print the details of a process while it exits. This gives us an account of the process while leaving, and also the order in which child processes leave.

On Exit Order

- Firstly, the output shows when was exit called for each child process. Notice, that for both, round robin and shortest job first, the exit order seems similar. This is because of the design of the test processes.
- The delay function in the test cases is to ensure that the corner case that all the processes have same run time despite highly varying burst time, does not occur.
- The delay function basically engages the cpu for some time, the time for which it keeps the cpu busy being specified as the argument.

- Since the process having higher burst time would thus have higher run time, in round robin scheduling, such a process would exit later. This is because, in round robin scheduling, all the runnable processes get equal time slice to run, thus a longer process would require more such slices.
- In the shortest job first, although the method is non-preemptive, the scheduling ensures that the process with smallest burst time runs first. This process, then runs upto its execution, and then the other processes start.
- The processes having IO Bound end later because they go to sleep multiple times in between.

On Context Switches

- There is a stark difference in the two scheduling methods in this aspect. The number of context switches for SJF (Shortest Job First) can be given by a simple equation

$$numContextSwtiches = 2 + numTimesSleepCalled$$

  The 2 is because the yeild() that was called in the set_burst_time method and then, when finally scheduling the process.
- For the round robin, the number of context switches increase with the burst time. This is because it is pre-emptive and the longer a process runs, the more time it will be thrown out and thus, the more the context switches.

> Note: Some corner cases that have been taken care of include
>
> 1. When a new process enters with a shorter burst time. Note that since the scheduling is pre-emptive, we would not stop the current process. However, since the shortest process is calculated in each loop, therefore, this newly entered process would get scheduled next.
> 2. A process enters with no set burst time.
>   In this case, we simply give the process the default burst time of 1. This is ensured in allocproc function. Thus the process would most likely be executed next. An alternative approach could be to add another attribute in the PCB or struct proc, that indicates if the burst time is set or should it be treated as not set.

Hybrid Scheduling

Implementation

- This is a hybrid of Round Robin and the Shortest Job First Scheduling.
- A ready queue was implemented in `scheduler()` to achieve this.
- The ready queue is basically a sorted array of PCBs that are in the RUNNABLE state. The sorting is done using the `merge sort` algorithm. This thus accounts into **O(nlogn)** complexity

- The `mergeSort()` and `merge()` functions have been used in the file `proc.c` to implement the merge sort in the scheduling

```
void mergeSort(struct proc *arr[], int l, int r)
{
  if (l < r)
  {
    // Same as (l+r)/2, but avoids overflow for
    // large l and h
    int m = l + (r - l) / 2;

    // Sort first and second halves
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    merge(arr, l, m, r);
  }
}
```

- The proc struct was also modified, with new members such as time slice and first proc added to keep track of the time slice taken by a process and the shortest process so that the time quanta variable could be changed as the time slice required for the first proc, i.e. the shortest burst time process.

- Further, there are modifications in `trap.c`. These are primarily to set the time quanta assigned to the smallest burst time in the ready queue.

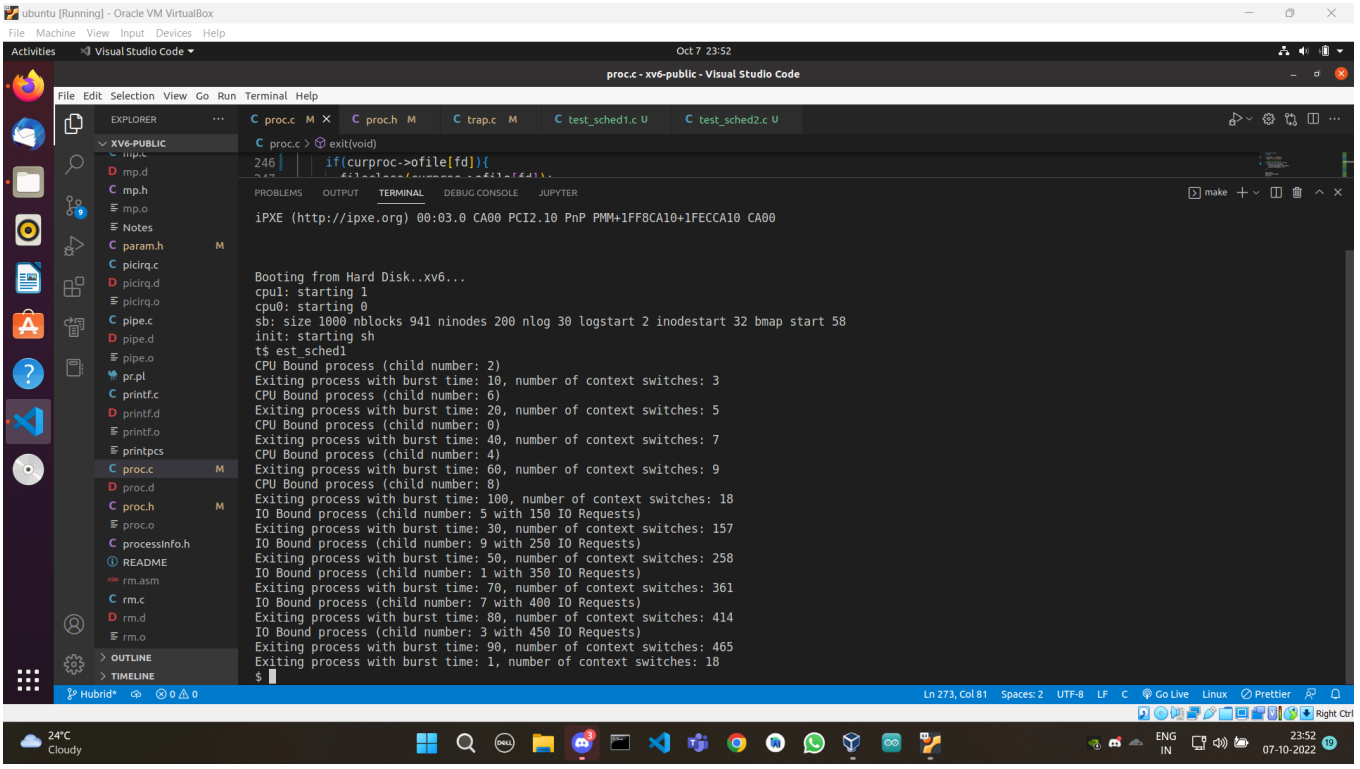
```
  if (myproc() && myproc()->state == RUNNING &&
      tf->trapno == T_IRQ0 + IRQ_TIMER)
  {
    if (myproc()->first_proc && (first_pid == -1 || first_pid == myproc()-
    {
      myproc()->time_slice++;
      time_quanta = myproc()->time_slice + 1;
      first_pid = myproc()->pid;
    }
    else
    {
      if (myproc()->time_slice < time_quanta)
      {
        myproc()->time_slice++;
      }
      else
      {
        myproc()->time_slice = 0;
        yield();
      }
    }
  }
  // Check if the process has been killed since we yielded
  if (myproc() && myproc()->killed && (tf->cs & 3) == DPL_USER)
    exit();
}
```

Testing

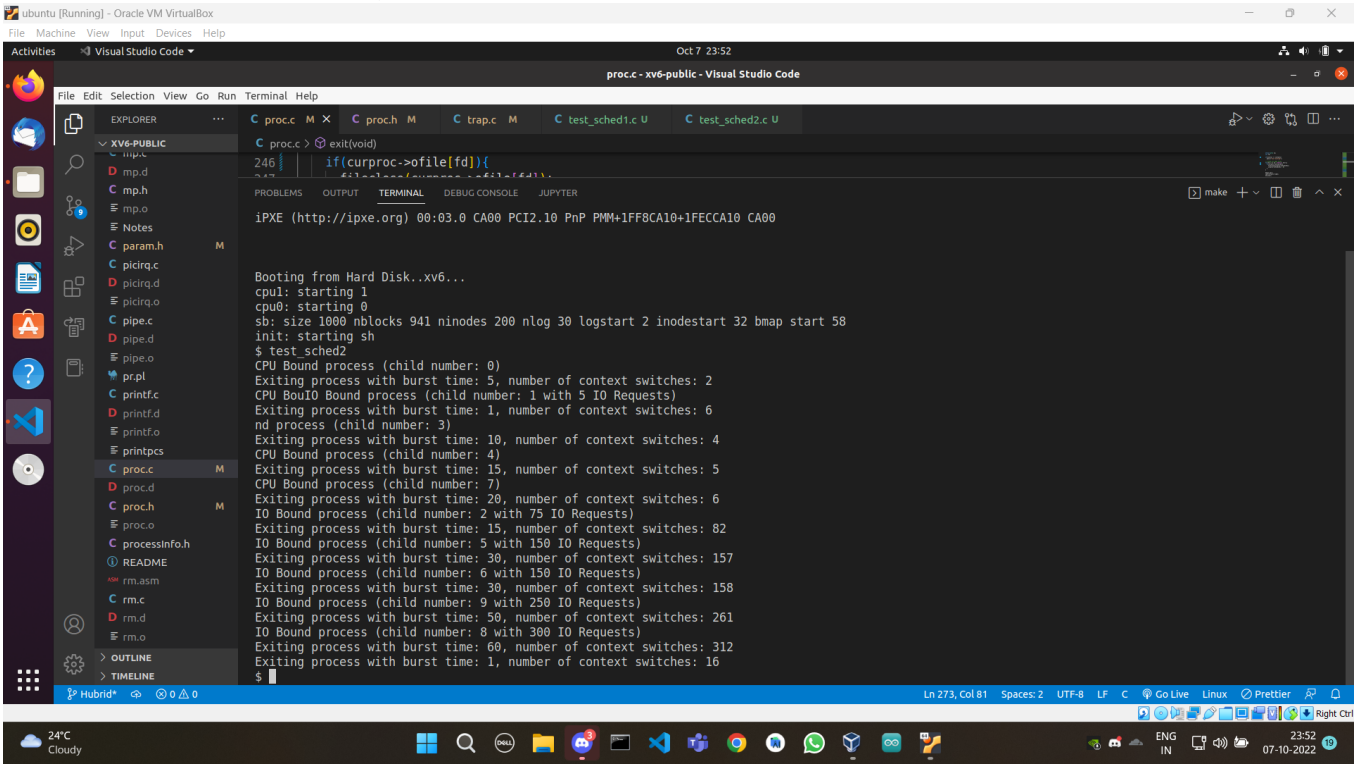
- The CPU bound processes employ a loop with many iterations, but the I/O bound processes use the sleep() system function (to wait for I/O operations). In addition to the burst timings, we publish the number of context shifts for each process.

- Further, the test cases are similar to the previous one.

- Test_sched1 on Hybrid Scheduling



- Test_sched2 on Hybrid Scheduling



OUTPUT ANALYSYS

- Note that here the context switches lie between the values of pure round robin and shortest job first. Also, we see that on each pass, the process with the smallest burst time is eliminated from the ready queue.
- The overall performance is better than the previous methods, as is visible by max context switches.