Assignment 2

Group Number: C31

Chandrabhushan Reddy	**200101027**
Billa Pramodh	**200101025**
Kalangi Sathvika	**200101048**

Part A:

1) getNumProc():

This system call returns the total number of active processes in the system.
We loop through the entire process table and count the number of processes which are active i.e., unused.

```
int getNumProc(void)
{
  struct proc *p;

  int count = 0;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED)
    {
      count++;
    }
  }
  release(&ptable.lock);
  return count;
}
```

getMaxPid():

This system call returns the maximum PID amongst the PIDs of all currently active processes in the system.
We acquire this by looping through the entire process table and taking the max of PIDs of all the currently active processes in the system.

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

  int max = -1;
  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;
}
```

2) getProcInfo(pid, &processInfo):

This system call takes as arguments an integer PID and a pointer to a structure processInfo. It returns parent PID, the number of times the process was context switched in by the scheduler, and the process size in bytes.

This is done by copying parent's PID, process size and number of context switches for the process into a buffer, and prints this data. We need to add new fields to keep track of some extra information. We've added data member in process structure to store the number of context switches.

```
int getProcInfo(int pid, struct processInfo* st)
{
  struct proc *p;
  int flag = -1;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid)
    {
      st->ppid = 0;
      // check if parent exists
      if(p->parent != 0)
      {
          st->ppid = p->parent->pid;
      }
      st->psize = p->sz;
      st->numberContextSwitches = p->contextswitches;
      flag = 0;
      break;
    }
  }
  release(&ptable.lock);
  return flag;
}
```

3) set_burst_time(n):
This system call sets burst time of the calling process to the input value.
We've added data member in process structure to store the burst time.

```
int set_burst_time(int bt)
{

  myproc()->burst = bt;
  // skip one CPU scheduling round.
  yield();
  return 0;
}
```

get_burst_time():
This system call returns the burst time of the calling process.

```
int
sys_get_burst_time()
{
  return myproc()->burst;
}
```

Files updated to add system calls:
To implement the above discussed system calls the following files need to be updated:

1. **Makefile :** Makefile needs to be edited before our user program is available for xv6 source code for compilation. We made only one change in Makefile i.e included _<userprogram_name>\ in the USER PROGRAMS (UPROGS).

2. **syscall.c :** Contains helper functions to parse system call arguments, and pointers to the actual system call implementations.
3. **syscall.h :** Contains a mapping from system call name to system call number.
4. **user.h :** Contains the system call definitions in xv6 code was added here for the new system calls.
5. **usys.S :** Contains a list of system calls exported by the kernel.
6. **sysproc.c :** Contains the implementations of process related system calls. System call code was added here.
7. **proc.c :** Contains the function scheduler which performs scheduling and context switching between processes.
8. **proc.h :** Contains the struct proc structure. Changes to this structure were made to track any extra information about a process

Files created to add user programs:
To test the above implemented system calls the following files are created:

1. getNumProc.c
2. getMaxPid.c
3. getProcInfo.c
4. set_burst_time.c

Outputs:
The outputs for the above discussed system calls are as follows:

```
$ getNumProc
Number of currently active processes: 3
$ getMaxPid
Greatest PID: 27
$ getProcInfo2
exec: fail
exec getProcInfo2 failed
$ getProcInfo 2
PPID: 1
Psize: 16384
Context switches: 31
$ set_burst_time 5
Burst time set to 5.
$ _
```

Part B:

Implementing SJF scheduler in xv6:

1. The shortest job scheduler function was implemented in the scheduler( ) function in proc.c file.
2. The initial burst time of all the processes when processes were created in system was set to zero as we want the system processes to run first i.e before all our processes. This was done under the allocproc( ) function in proc.c.
3. The lines 105-107 in the file trap.c were commented to turn off the preemption in the scheduler as SJF is non-preemptive in nature

```
105    // if(myproc() && myproc()->state == RUNNING &&
106    //    tf->trapno == T_IRQ0+IRQ_TIMER)
107    //   yield();
```

4. In the function sys_set_burst_time( ) in file sysproc.c the yield() function was added to give up the CPU for one scheduling round.
5. In the file param.h the NCPU parameter was set to 1 as we wanted to simulate only one CPU. Multiple CPUs are simulated, so if one CPU is running a process, other will not take that into account while finding a new one with the shortest burst.

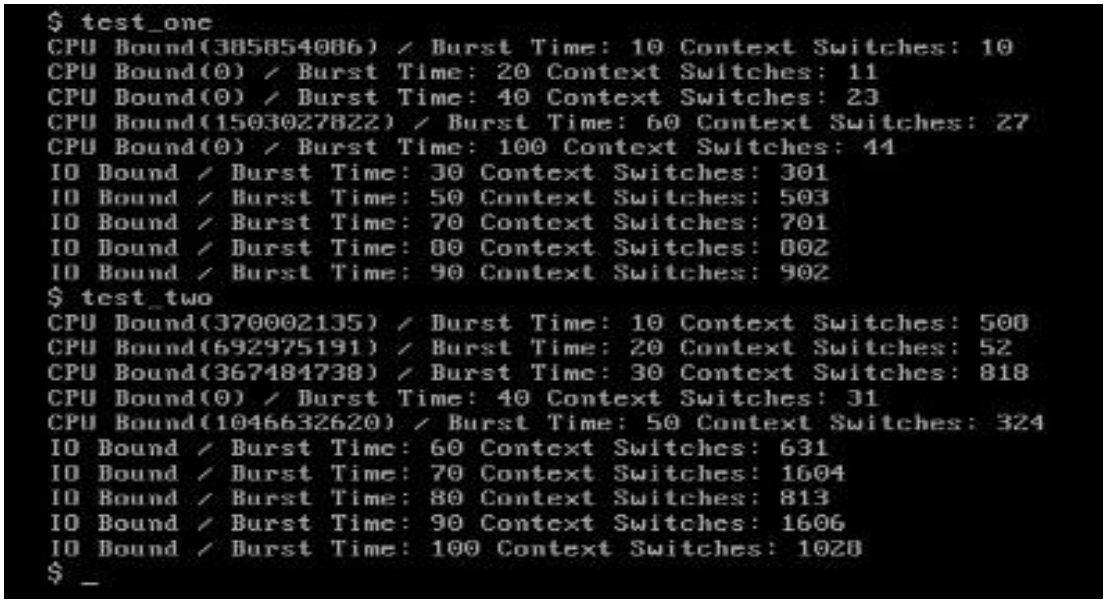
```
// #define NCPU          8  // maximum number of CPUs
#define NCPU             1  // maximum number of CPUs
#define NOFILE          16  // open files per process
```

6. We have iterated over all the processes which are RUNNABLE and chose the one with the smallest burst time for scheduling, if we have n processes then our scheduler has a time complexity of O(n).

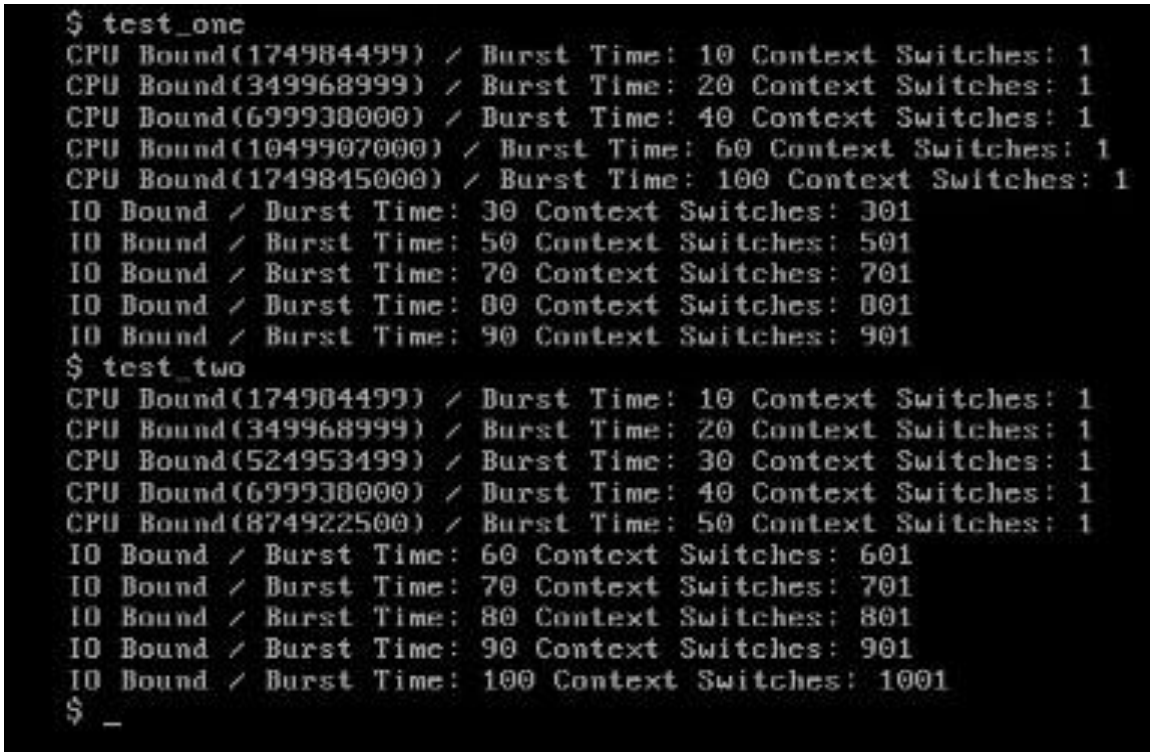
```
    acquire(&ptable.lock);
    // To store the job with least burst time
    struct proc *shortest_job = 0;

    // Find the job with least burst time
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state != RUNNABLE)
        continue;

      if (!shortest_job)
      {
        shortest_job = p;
      }
      else
      {
        if (p->burst < shortest_job->burst)
        {
          shortest_job = p;
        }
      }
    }

    if (shortest_job)
    {
      p = shortest_job;
      //cprintf("BT%d \n", p->burst);
      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

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

      // Increment number of context switches
      p->contextswitches = p->contextswitches + 1;
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);
```

Testing the SJF scheduler in xv6:

1. User programs test_one and test_two are created to test the SJF scheduler.
2. It is a simple program which forks an even mix of CPU bound and I/O bound processes, assigns them burst times using the set_burst_time( ) system call in Part A and then prints the burst time, type of process (I/O bound or CPU bound) and the number of context switches when the forked processes are about to terminate.
3. The CPU bound processes use a loop with a large number of iterations and the I/O bound processes use the sleep( ) system call (so that the process waits for I/O operations).
4. When we execute test_one and test_two, we see both CPU bound processes and I/O bound processes terminate in ascending order of burst times among themselves, showcasing a successful SJF scheduling algorithm.
5. The screenshots below show the order of execution of processes in the default Round Robin algorithm and the SJF algorithm.



Round Robin Algorithm



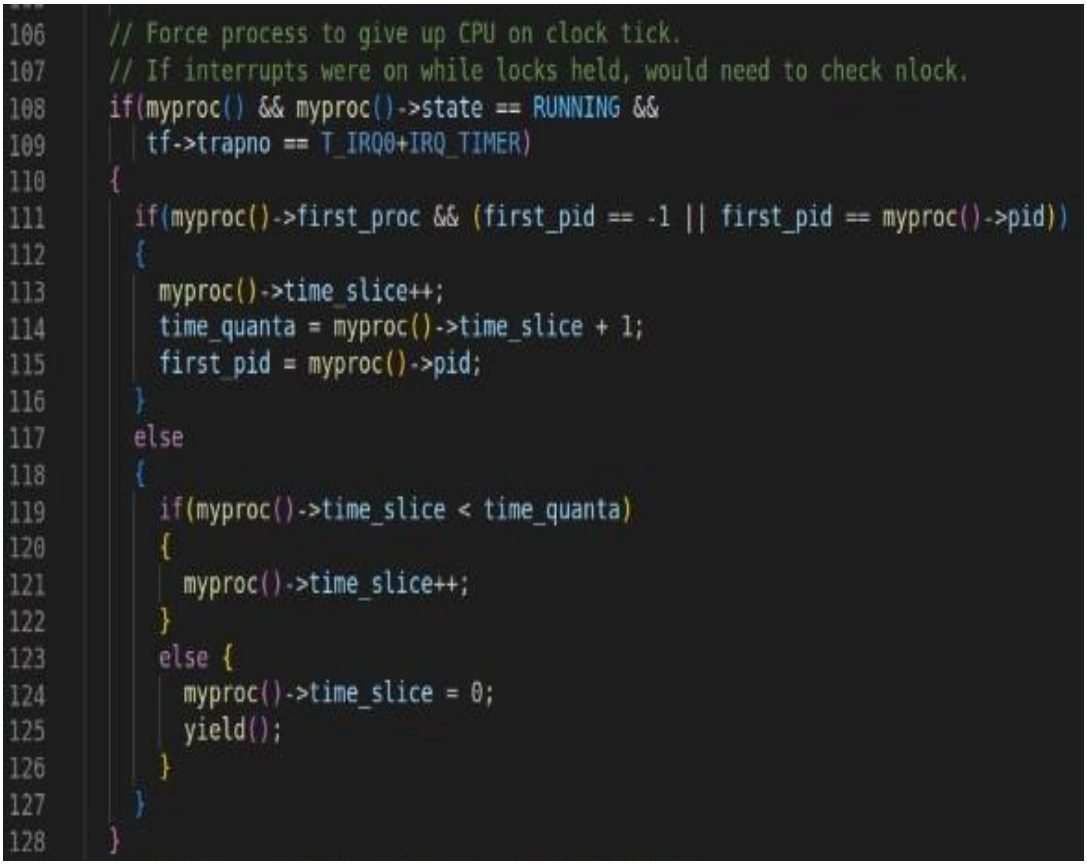
Shortest Job First Algorithm

Adding a Hybrid (SJF+Round Robin) Scheduler in xv6 :

1. A ready queue was created in the the function scheduler( ) in the file proc.c wherein all the processes with their state as runnable were pushed into the queue.
2. Then the ready queue was sorted according to the burst times of the processes present in it using the BUBBLE SORT algorithm.

```
441   // Set up Ready Queue
442   struct proc * RQ[NPROC];
443
444   int k = 0;
445
446   for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
447   {
448     if(p->state == RUNNABLE)
449     {
450       RQ[k++] = p;
451     }
452   }
453   struct proc *t;
454   // Sort Ready Queue
455   for (int i = 0; i < k; i++)
456   {
457     for(int j = i + 1; j < k; j++)
458     {
459       if(RQ[i]->burst > RQ[j]->burst)
460       {
461         t = RQ[i];
462         RQ[i] = RQ[j];
463         RQ[j] = t;
464       }
465     }
466   }
```

3. The proc struct was also modified wherein additional members such as time_slice and first_proc were included to keep the track of the time slice taken by a process and keep track of the shortest process so as to change the time_quanta variable as the time_slice required for the first_proc i.e the shortest burst time process.
4. The code of trap.c was modified to account for the same.



```
106   // Force process to give up CPU on clock tick.
107   // If interrupts were on while locks held, would need to check nlock.
108   if(myproc() && myproc()->state == RUNNING &&
109      tf->trapno == T_IRQ0+IRQ_TIMER)
110   {
111     if(myproc()->first_proc && (first_pid == -1 || first_pid == myproc()->pid))
112     {
113       myproc()->time_slice++;
114       time_quanta = myproc()->time_slice + 1;
115       first_pid = myproc()->pid;
116     }
117     else
118     {
119       if(myproc()->time_slice < time_quanta)
120       {
121         myproc()->time_slice++;
122       }
123       else {
124         myproc()->time_slice = 0;
125         yield();
126       }
127     }
128   }
```

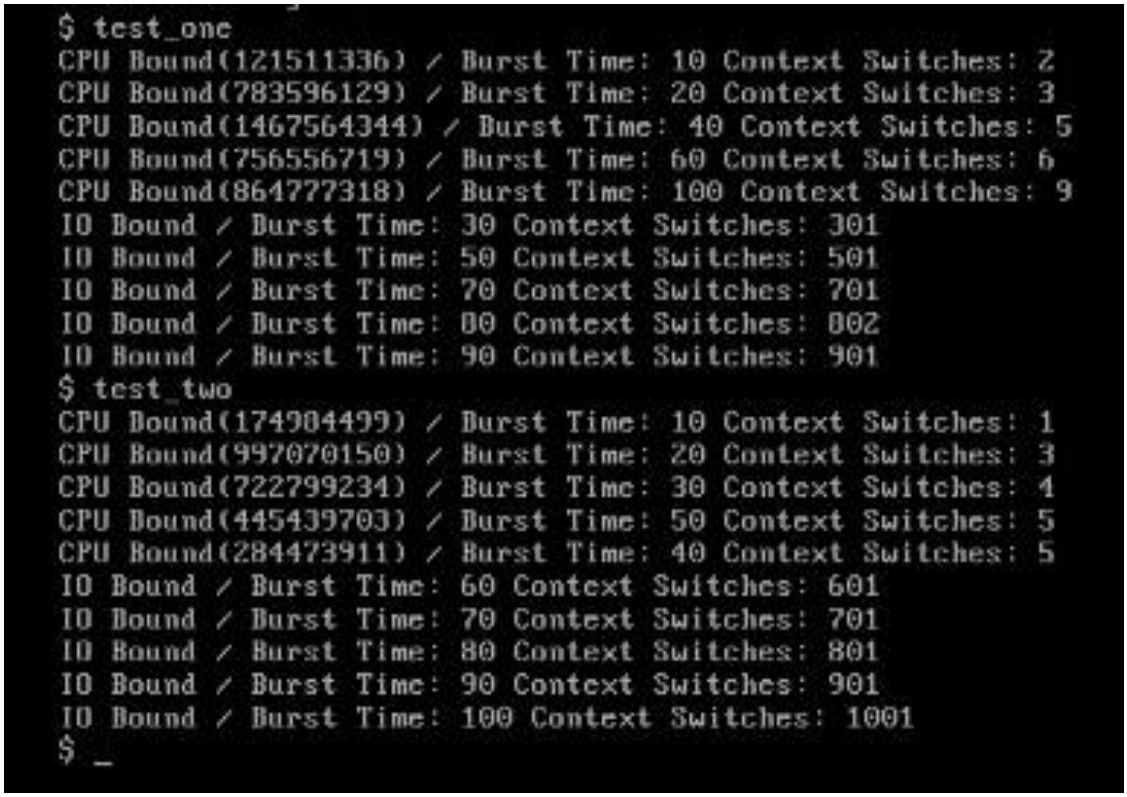
5. We have sorted all the processes which are RUNNABLE using BUBBLE SORT and chose the one with the smallest burst time for scheduling, if we have n processes then our scheduler has a time complexity of O(n^2) to sort the processes

Testing the Hybrid (SJF+Round Robin) Scheduler in xv6:

1. User programs test_one and test_two are created to test the hybrid scheduler.
2. It is a simple program which forks an even mix of CPU bound and I/O bound processes, assigns them random burst times using the set_burst_time( ) system call in Part A and then prints the burst times when the forked processes are about to terminate.
3. The CPU bound processes use a loop with a large number of iterations and the I/O bound processes use the sleep( ) system call (so that the process waits for I/O operations). Along with

the burst times, we print the number of context switches for each process as well.

4. The SJF scheduler was non-preemptive and hence every CPU bound process had a small number of context switches. However, the hybrid scheduler is preemptive, and the number of context switches is roughly proportional to the burst time of the process. I/O bound processes have a large number of context switches because of lots of I/O delays.
5. When we execute test_one and test_two, we see that all CPU bound processes terminate before I/O bound processes, and both CPU bound processes and I/O bound processes terminate in ascending order of burst times among themselves, showcasing a successful hybrid scheduling algorithm.
6. The screenshot below shows the order of execution of processes in the hybrid algorithm, along with the number of context switches for each process.



Note:

1) The files scheduler.patch and hybrid.patch which are submitted along with the report are the patch files for (system calls + SJF Scheduler) and (Hybrid Scheduler) respectively.
2) On applying the above patch files on the original xv6 directory we get our desired xv6 schedulers.
3) On running make command if we encounter the following error,

   “make: execvp: ./sign.pl: Permission denied”

   Then run the following command and restart qemu again

   “chmod 777 -R yourDirName”

   Here in this case yourDirName is sign.pl