CS344 : OS Lab, Assignment-2
Group-C33

Name: Mukul Garg Roll no. : 200101068
Name: Arpan Khanna Roll no. : 200101121
Name: Kartikey Gupta Roll no. : 200101055

PART A:

1. getNumProc: Prints the total number of active processes by looping through the process table.

```
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;
}
```

2. getMaxPid: Prints the Process ID of the process with the maximum process ID in the process table.

```
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;
}
```

3. getProcInfo: Copies parent's PID, process size and a number of context switches for the process into a buffer and prints this data. Added data members 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)
    {
      // check if parent exists
      if (p->parent != 0)
      {
        st->ppid = p->parent->pid;
      }
      else
      {
        st->ppid = 0;
      }

      st->psize = p->sz;
      st->numberContextSwitches = p->contextswitches;
      flag = 0;
      break;
    }
  }
  release(&ptable.lock);
  return flag;
}
```

4. set_burst_time: Sets the burst time of the calling process to the input value. 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;
}
```

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

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

Files created to add system calls and user programs

- getNumProc.c
- getMaxPid.c
- getProcInfo.c
- set_burst_time.

OUTPUT

```
$ getNumProc
Number of currently active processes: 3
$ getMaxPid
Greatest PID: 5
$ getProcInfo 6
PPID: 2
Psize: 45056
Context switches: 6
$ set_burst_time 6 10
Burst time set to 6.
```

PART B

Initially in xv6, Round-Robin Scheduler is implemented, which preempts the process after 1 clock cycle, i.e. the value of time quantum is equal to 1 clock cycle. This part is implemented in trap.c file in line no. 103-107 where we call yield() function, which 6 forces the process to give up CPU. At the end of each interrupt, trap calls yield. Yield in turn calls sched, which calls switch() to save the current context in proc->context and switch to the scheduler context previously saved in CPU->scheduler.

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

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.

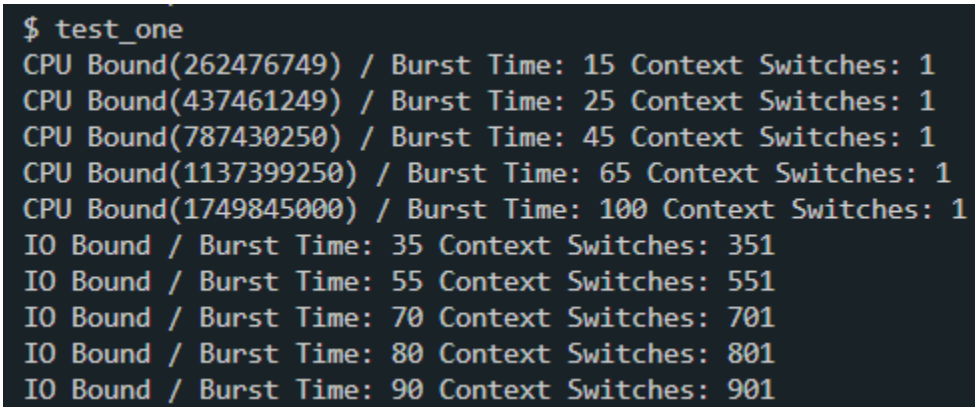
```
#define NCPU          1  // maximum number of CPUs
```

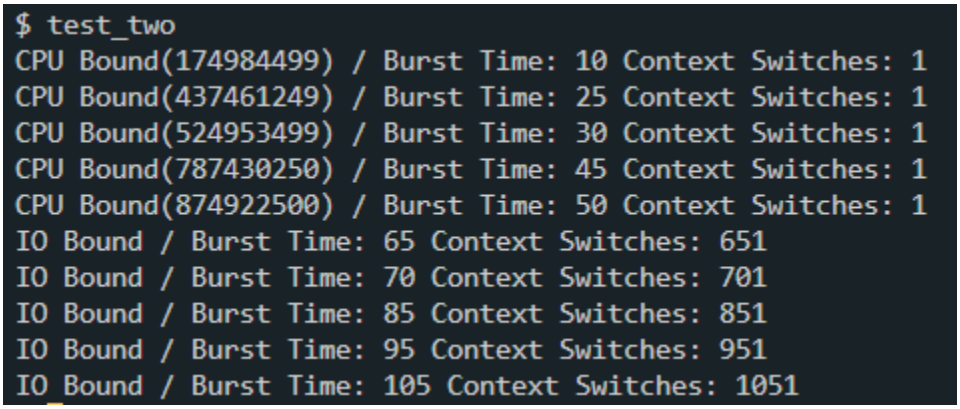
The shortest job scheduler function was implemented in the scheduler( ) function in proc.c file. The initial burst time of all the processes when processes were created in system was set to zero as we want the system processes to run first i.e before all our processes.This was done under the allocproc( ) function in proc.c. 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).

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

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

    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;

      //  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);
  }
}
```

OUTPUT

User programs test_one and test_two are created to test the Shortest Job First (SJF) scheduler. 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. 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). 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 SJF scheduling algorithm.





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

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.The code of trap.c was modified to account for the same

```
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
{
  if(myproc()->first_proc && (first_pid == -1 || first_pid == myproc()->pid))
  {
    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();
    }
  }
}
```

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. Then the ready queue was sorted according to the burst times of the processes present in it using the **BUBBLE SORT** algorithm.

```
struct proc *t;
// Sort Ready Queue
for (int i = 0; i < k; i++)
{
  for (int j = i + 1; j < k; j++)
  {
    if (RQ[i]->burst > RQ[j]->burst)
    {
      t = RQ[i];
      RQ[i] = RQ[j];
      RQ[j] = t;
    }
  }
}
if (k && flag)
{
  RQ[0]->first_proc = 1;
  flag = 0;
}
```

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.

OUTPUT

User programs test_one and test_two are created to test the hybrid scheduler. 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. 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. 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. 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.

```
$ test_one
CPU Bound(303577949) / Burst Time: 15 Context Switches: 2
CPU Bound(1189995076) / Burst Time: 25 Context Switches: 3
CPU Bound(1026890693) / Burst Time: 45 Context Switches: 4
CPU Bound(885871630) / Burst Time: 65 Context Switches: 5
CPU Bound(1107913416) / Burst Time: 100 Context Switches: 8
IO Bound / Burst Time: 35 Context Switches: 351
IO Bound / Burst Time: 55 Context Switches: 551
IO Bound / Burst Time: 70 Context Switches: 701
IO Bound / Burst Time: 80 Context Switches: 801
IO Bound / Burst Time: 90 Context Switches: 901
```

```
$ test_two
CPU Bound(221263315) / Burst Time: 10 Context Switches: 2
CPU Bound(1184560582) / Burst Time: 25 Context Switches: 3
CPU Bound(299487721) / Burst Time: 30 Context Switches: 3
CPU Bound(776333747) / Burst Time: 45 Context Switches: 4
CPU Bound(373325266) / Burst Time: 50 Context Switches: 4
IO Bound / Burst Time: 65 Context Switches: 651
IO Bound / Burst Time: 70 Context Switches: 701
IO Bound / Burst Time: 85 Context Switches: 851
IO Bound / Burst Time: 95 Context Switches: 951
IO Bound / Burst Time: 105 Context Switches: 1051
```