OS-344 Assignment-2

Group C25

Name	Roll
Anindya Vijayvargeeya	200101015
Gunjan Dhanuka	200101038
Pranjal Singh	200101084

Part A

1. **getNumProc() and getMaxPid()** - For these, we iterated through the process table and ignored all processes with state UNUSED. We access ptable only after acquiring its lock, and release it as soon as we're done. Here is the implementation code for these functions:

```c
int getNumProc(void)
{
  int num_processes = 0;

  struct proc *p;

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

  return num_processes;
}
```

```c
int getMaxPid(void)
{
  int max_pid = -1;

  struct proc *p;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->state != UNUSED)
      if(p->pid > max_pid){
        max_pid = p->pid;
      }
  }
  release(&ptable.lock);

  return max_pid;
}
```

2. **getProcInfo()** - We included the provided file processInfo.h in user.h and proc.c, and then we modified proc.h to include an extra field contextSwitches in the proc struct using which we track the number of context switches. Size and parent process fields are already available in the proc struct so for that we just access these already available fields, and to count context switches we
    - Changed the allocproc function where we initialized it to 0

```
p->contextSwitches = 0; // set contextSwitches to zero initially
```

    - Changed the scheduler function where we incremented this variable each time we context switch out of the process

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

    Both these changes were done in proc.c file.

    Below is the implementation of getProcInfo.

```
int getProcInfo(int targetPid, struct processInfo *pinfo)
{
  struct proc *p;
  int processFound = 0;

  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid != targetPid)
      continue;

    processFound = 1;

    if (p->parent != 0)
      pinfo->ppid = p->parent->pid;
    pinfo->psize = p->sz;
    pinfo->numberContextSwitches = p->contextSwitches;
    break;
  }
  release(&ptable.lock);

  if (processFound == 0)
    return -1; //error

  return 0; //success
}
```

3. **set_burst_time() and get_burst_time()** - we modified the proc struct in proc.h file to add a field ‘burst’ indicating the approximate burst time. Then we added system calls to set and get burst time. Below is their codes. myproc returns the current process’ proc sturct, whose burst field is then accessed.

```
int set_burst_time(int n){
  myproc()->burst = n;

  //skip one CPU scheduling round
  yield();
  return 0;
}
```

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

Testing these system calls:

We added user level programs to test all system calls separately, and added them to the Makefile.In total 5 files were added to test the above mentioned 5 system calls :

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

```
$ getNumProc
Number of active processes: 3
$ getMaxPid
Maximum PID: 5
```

```
$ set_burst_time 20
Burst time set to 20
$ get_burst_time
Burst time=20
```

```
$ getProcInfo 1
PPID: 0
PSize: 12288
Context Switches: 19
```

```
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16416
echo           2 4 15268
forktest       2 5 9588
grep           2 6 18636
init           2 7 15856
kill           2 8 15300
ln             2 9 15156
ls             2 10 17784
mkdir          2 11 15400
rm             2 12 15376
sh             2 13 28020
stressfs       2 14 16288
usertests      2 15 67396
wc             2 16 17156
zombie         2 17 14968
getNumProc     2 18 15076
getMaxPid      2 19 15180
getProcInfo    2 20 16024
set_burst_time 2 21 15432
get_burst_time 2 22 15176
test           2 23 17840
console        3 24 0
```

The ‘test’ program was also added to test the scheduler, which we’ll talk about in the next part.

Part B

Shortest Job First Scheduler

To change the default round robin scheduler to a shortest job first one, first we had to disable timer interupts, which is used by kernel to preempt processes after a fixed duration, as needed in round robin scheduling. We commented the following 3 lines from trap function in trap.c file.

```
103   // Force process to give up CPU on clock tick.
104   // If interrupts were on while locks held, would need to check nlock.
105   if(myproc() && myproc()->state == RUNNING &&
106      tf->trapno == T_IRQ0+IRQ_TIMER)
107     yield();
108
```

Next, we need to have only one CPU instance, for which we changed the following line in param.h file.

```
2   #define KSTACKSIZE 4096  // size of per-process kernel stack
3   #define NCPU          1  // maximum number of CPUs
4   #define NOFILE       16  // open files per process
```

Now, to run the shortest job, we just iterated through the entire process table to find the job with shortest burst time value. Inside the infinite while loop of the scheduler, we changed the part where it iterates through the process table running each runnable job, and added the following code that first searches for the shortest job, then runs it.

```
// Loop over process table looking for process to run.
acquire(&ptable.lock);

// first find which process is the shortest
struct proc* my_p = 0;
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

  if (p->state != RUNNABLE)
    continue;

  if(my_p == 0) my_p = p;
  else if(my_p->burst > p->burst) my_p = p;
}
```

```
if(my_p != 0) {
  p = my_p;
  // 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);
  p->contextSwitches++;

  switchkvm();

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

Scheduler runs a process by first calling switchuvm function, which switches page table to that of that process, sets the state of that process to RUNNING, and calls swtch which does the context switching, saving state of scheduler and loads that of the process we wanted to run. Later, when the process finishes or is preempted, the control returns back to the scheduler where it calls switchkvm which changes page table to that of the kernel, and then again searches for a new process to run.

Runtime Complexity : O(N), where N is the max number of processes in the process table. This is because for each iteration of scheduling the scheduler has to go through the entire process table to find the process with shortest burst time.

Testing

To test the scheduler, we have created a test.c file and included it in Makefile. In it, we simulate a list of processes with different burst time values, assigned from a list of 15 values. We forked 10 times creating 10 child processes, half of them were given a computationally intensive task, simulating CPU bound processes, and other half were put to repeatedly put to sleep, simulating IO bound processes.

```
if(i%2==0){
  // CPU bound process
  double temp = 0;
  for(int j=0; j<500000*burst_vals[i]; j++){
    temp += 3.14*2.71;
  }
  int computed_val = ((int)temp)%9000 + 1000;
  printf(1, "%d\t CPU Bound(%d) \t", getpid() ,computed_val);
}
```

```
else {
  // IO bound process
  for(int j=0; j<10*burst_vals[i]; j++){
    sleep(1);
  }
  printf(1, "%d\t IO Bound    \t\t", getpid());
}
```

We get the following output when we run this on the **default round robin scheduler** of xv6 OS :

PID	Process type	Burst Time	Context Switches
6	CPU Bound(5179)	11	7
4	CPU Bound(2198)	40	19
8	CPU Bound(7972)	39	20
10	CPU Bound(8722)	61	29
12	CPU Bound(5788)	77	43
7	IO Bound	35	351
13	IO Bound	41	412
9	IO Bound	60	601
5	IO Bound	70	702
11	IO Bound	73	732

Here we see the context switches are in ascending order, while burst times are not.

On running with the **Shortest Job First scheduler** we implemented in this part, we get the following output:

PID	Process type	Burst Time	Context Switches
6	CPU Bound(2699)	11	1
8	CPU Bound(1300)	39	1
4	CPU Bound(7999)	40	1
10	CPU Bound(4699)	61	1
12	CPU Bound(3900)	77	1
7	IO Bound	35	351
13	IO Bound	41	411
9	IO Bound	60	601
5	IO Bound	70	701
11	IO Bound	73	731

Here we see the processes of one type (CPU bound or IO bound) complete in order of their burst time values. Also, CPU bound processes have only one context switch, since this type of scheduling is non-preemptive. For IO bound processes we were putting them to sleep, so we have context switches one more than the number of times we called sleep on them.

Hybrid Scheduler

We first added 2 fields in the proc struct definition in the proc.h file, time_slice and quanta_val. The former specifies time slice within a quanta that this process has completed, and the latter is the total length of the time quanta. We modified trap function in trap.c file so that each time the default timer interrupt is involved, we first increment the time_slice variable and then check if its value is equal to quanta_val. If not, we won't preempt that process.

```
103   // Force process to give up CPU on clock tick.
104   // If interrupts were on while locks held, would need to check nlock.
105   if(myproc() && myproc()->state == RUNNING &&
106      tf->trapno == T_IRQ0+IRQ_TIMER)
107   {
108     (myproc()->time_slice)++;
109     if(myproc()->time_slice == myproc()->quanta_val){
110       myproc()->time_slice = 0;
111       yield();
112     }
113   }
```

In the proc.c file, we changed the scheduler function as follows: first create an array RQ (ready queue) which takes in processes from process table that are runnable. Then, we keep track of non-runnable processes in the ready queue using variable num_stale, and make sure if and only if there are no runnable processes then we refill the ready queue.

When we refill, we sort the queue according to burst times values using bubble sort, and then set the time quanta as the first process (in the sorted array) with non-zero burst time value (since on process allocation we set burst time to 0). If there are no non-zero burst time value then we set the quanta to 1, thus getting the default round robin timer value.

```
// first refill the RQ if no processes left to execute
if(num_stale == num_p){
  // first store all runnable processes in the RQ array
  num_p = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state == RUNNABLE)
      RQ[num_p++] = p;
  }
  num_stale = 0;

  if(num_p == 0){
    release(&ptable.lock);
    continue;
  }

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

```
  int i=0;
  // now we set the time quanta value
  int quanta_val = 0;
  while(quanta_val==0 && i<num_p){
    quanta_val = RQ[i++]->burst;
  }
  if(quanta_val==0 ) quanta_val = 1;
  for(i=0; i<num_p; i++){
    RQ[i]->quanta_val = quanta_val;
  }
} else {
  num_stale = 0;
}
```

Once we have the RQ array filled up, we iterate through each process in this array and run it. Preemption is being taken care of by trap.c, as mentioned above.

```
  for(int i=0; i<num_p; i++){
    p = RQ[i];
    if(p->state != RUNNABLE) { num_stale++; continue;}

    // 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);
    p->contextSwitches++;

    switchkvm();

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

Runtime Complexity: O(N), since creation of ready queue takes O(N^2) time which gets amortized over N processes which we then subsequently run (without sorting the array again). So per process we have O(N) time of scheduling.

Testing

We run the same test program with this **hybrid scheduler**

```
PID      Process type     Burst Time    Context Switches
---      ------------     ----- ----    ------- --------
6        CPU Bound(7372)         11        6
8        CPU Bound(6846)         39        19
4        CPU Bound(5334)         40        20
10       CPU Bound(9340)         61        30
12       CPU Bound(9955)         77        36
7        IO Bound                35        351
13       IO Bound                41        411
9        IO Bound                60        601
5        IO Bound                70        701
11       IO Bound                73        731
```

Here the context switches of CPU bound processes are not 1, since they were evicted as the scheduling did involve round robin preemption, and still the output is in ascending order of burst times for each type of processes (CPU bound and IO bound). Thus we successfully implemented a hybrid of Round robin and Shortest job first scheduling.