

CS344 Assignment-2

Name: Dhruv Tarsadiya Roll No: 200101123

Name: Vikram Singla Roll No: 200101117

Name: Harshwardhan Bhakkad Roll No: 200101040

Part-A

The following five system calls were created and the relevant files as given in the current and previous assignments were modified:

1. **getNumProc** : Prints total number of active processes by looping through the process table.
2. **getMaxPid** : Prints the Process ID of process with maximum process ID in the process table.
3. **getProcInfo** : Copies parent's PID, process size and number of context switches for the process into a buffer and prints this data. Added attribute in process structure to store the number of context switches.
4. **set_burst_time** : Sets burst time of the calling process to the input value. Added attribute in process structure to store the burst time.
5. **get_burst_time** : Returns burst time of the calling process.

Corresponding to these, 4 Test Programs were also created to test these system calls.

To create the system calls **getNumProc()** and **getMaxPid()**, a procedure similar to the one used to create a system call in the previous assignment is used. The functions access the process table by acquiring its lock and then loop through it to carry out their respective tasks.

The output obtained on calling the user programs is attached below. First, a `ls` command is run which shows the list of user programs available. This process is run with process ID 3. Because process ID 1 and 2 are allotted to system processes because of which the next available process ID is 3 which is allotted to `ls`. After completion of `ls` process, `getMaxPid` is run. The next available process ID is 4 which is allotted to `getMaxPid`. At this time, 3 processes (`ls` has already terminated) are currently running on the xv6 OS, the two system processes with PID 1 and PID 2 and `getMaxPid` with PID 4. Hence, the output is 4.

```

Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap sta8
init: starting sh
$ ls
.          1 1 512
..         1 1 512
README    2 2 2286
cat        2 3 16424
echo       2 4 15276
forktest   2 5 9592
grep       2 6 18644
init       2 7 15864
kill       2 8 15304
ln         2 9 15160
ls         2 10 17792
mkdir     2 11 15408
rm         2 12 15384
sh         2 13 28028
stressfs   2 14 16292
usertests  2 15 67404
wc         2 16 17160
zombie     2 17 14976
getNumProc 2 18 15020
getMaxPid  2 19 15184
getProcInfo 2 20 16004
set_burst_time 2 21 15428
test_one   2 22 16976
test_two   2 23 16968
console    3 24 0
$ getMaxPid
Greatest PID: 4
$ getNumProc
Number of currently active processes: 3

```

Figure 1: Output for Test Programs 1 and 2

Similarly, when `getNumProc` is run, 3 processes are running (2 system processes and 1 `getNumProc`). Hence the output is 3.

The first few steps to implement the system call `getProcInfo` is identical to those of previously made system calls. But here we also have to pass some arguments to the system call unlike the ones we previously made above. We use `flag` (a dummy variable) for the case that there exists no process with the given PID. The output obtained on running the corresponding Test program is given below:

```

$ getProcInfo 5
PPID: 2
Psize: 45056
Context switches: 0
$ getProcInfo 1
PPID: 0
Psize: 12288
Context switches: 13
$

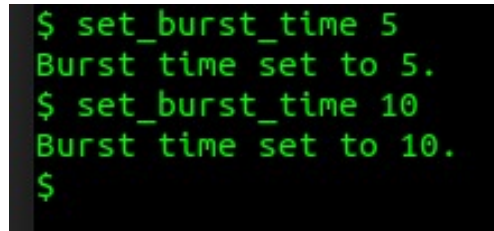
```

Figure 2: Output for Test Program 3

For the last part, an additional attribute namely `burst_time` has to be added to `proc` structure. We also have to implement 2 system calls namely, `set_burst_time` and `get_burst_time` which

will set and get the burst time of current process to a given value respectively.

We run a simple Test to check which utilises these two system calls. The output is given below.



```
$ set_burst_time 5
Burst time set to 5.
$ set_burst_time 10
Burst time set to 10.
$
```

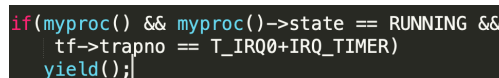
Figure 3: Output for Test Program 4

Part-B

Adding a Shortest Job First (SJF) scheduler in xv6

The shortest job scheduler function was implemented in the scheduler() function in proc.c file. Please note that the initial burst time of all the processes when processes were created in system was set to 0 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.

Remove the preemption of the current process (yield) on every OS clock tick so the current process completely finishes first. In the given round robin scheduler, the forced preemption of the current process with every clock tick is being handled in the trap.c file. We comment out the following lines from trap.c to fix this issue:



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

Figure 4: Commenting out the above lines in trap.c

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.

As we can see from Figure 5, each time the scheduler is called, 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 $O(n)$** .

Corner cases such as when ZOMBIE child processes are freed, are checked as we only use RUNNABLE processes in our loop. Note that we acquire lock to the process table before this. Also note that we could also have implemented a min-heap (priority queue) for the

```

acquire(&ptable.lock);
struct proc *shortest_job = 0;

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;
    c->proc = p;
    switchvm(p);
    p->state = RUNNING;

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

    p->contextswitches = p->contextswitches + 1;
    switchkvm();

    c->proc = 0;
}
release(&ptable.lock);

```

Figure 5: Implemented in proc.c

SJF, it would have had a time complexity of $O(\log(n))$. Since, the assignment had left the implementation details to us, we chose to not use priority queue for our implementation of SJF.

Testing the SJF scheduler in xv6 :

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 in order to simulate 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.**

The Figures show the order of execution of processes in the default Round Robin algorithm and the newly implemented SJF algorithm.

As we can see, since the Round Robin scheduler uses an FCFS queue, the order of execution

```

$ test_one
CPU Bound(426111965) / Burst Time: 10 Context Switches: 6
CPU Bound(894471173) / Burst Time: 20 Context Switches: 11
CPU Bound(1760099021) / Burst Time: 50 Context Switches: 27
CPU Bound(405800604) / Burst Time: 60 Context Switches: 34
CPU Bound(578478983) / Burst Time: 70 Context Switches: 41
IO Bound / Burst Time: 30 Context Switches: 301
IO Bound / Burst Time: 40 Context Switches: 401
IO Bound / Burst Time: 80 Context Switches: 801
IO Bound / Burst Time: 90 Context Switches: 901
IO Bound / Burst Time: 100 Context Switches: 1001

```

(a) Test 1 for Round Robin Scheduler

```

$ test_one
CPU Bound(174984499) / Burst Time: 10 Context Switches: 1
CPU Bound(349968999) / Burst Time: 20 Context Switches: 1
CPU Bound(874922500) / Burst Time: 50 Context Switches: 1
CPU Bound(1049907000) / Burst Time: 60 Context Switches: 1
CPU Bound(1224891500) / Burst Time: 70 Context Switches: 1
IO Bound / Burst Time: 30 Context Switches: 301
IO Bound / Burst Time: 40 Context Switches: 401
IO Bound / Burst Time: 80 Context Switches: 801
IO Bound / Burst Time: 90 Context Switches: 901
IO Bound / Burst Time: 100 Context Switches: 1001
$

```

(b) Test 1 for SJF Scheduler

```

$ test_two
CPU Bound(1334118049) / Burst Time: 20 Context Switches: 12
CPU Bound(964422837) / Burst Time: 30 Context Switches: 18
CPU Bound(1148092482) / Burst Time: 50 Context Switches: 26
CPU Bound(1410637336) / Burst Time: 90 Context Switches: 48
CPU Bound(1540538130) / Burst Time: 100 Context Switches: 55
IO Bound / Burst Time: 10 Context Switches: 101
IO Bound / Burst Time: 40 Context Switches: 401
IO Bound / Burst Time: 60 Context Switches: 601
IO Bound / Burst Time: 70 Context Switches: 701
IO Bound / Burst Time: 80 Context Switches: 801

```

(a) Test 2 for Round Robin Scheduler

```

$ test_two
CPU Bound(349968999) / Burst Time: 20 Context Switches: 1
CPU Bound(524953499) / Burst Time: 30 Context Switches: 1
CPU Bound(874922500) / Burst Time: 50 Context Switches: 1
CPU Bound(1574860500) / Burst Time: 90 Context Switches: 1
CPU Bound(1749845000) / Burst Time: 100 Context Switches: 1
IO Bound / Burst Time: 10 Context Switches: 101
IO Bound / Burst Time: 40 Context Switches: 401
IO Bound / Burst Time: 60 Context Switches: 601
IO Bound / Burst Time: 70 Context Switches: 701
IO Bound / Burst Time: 80 Context Switches: 801
$

```

(b) Test 2 for SJF Scheduler

is highly related to the PID of the process whereas in the SJF scheduler, the scheduling is happening by the burst times. Also, the number of context switches in the RR scheduler is very high.

Bonus Question

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

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.

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_quantum 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.

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.

Again as we have explained earlier, this could be made more efficient with a priority queue implementation.

```

// Set up Ready Queue
struct proc * RQ[NPROC];

int k = 0;

for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
    if(p->state == RUNNABLE)
    {
        RQ[k++] = p;
    }
}

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;
        }
    }
}
}

```

Figure 8: Ready Queue Implementation

```

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();
        }
    }
}
}

```

Figure 9: Changes in trap.c

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

User programs `test_one` and `test_two` are created to test the hybrid scheduler. These are simple programs which fork an even mix of CPU bound and I/O bound processes, assign them random burst times using the `set_burst_time()` system call in Part A and then print the burst times when the forked processes are about to terminate.

Similar to Part-B, 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 for CPU bound processes.

I/O bound processes have a large number of context switches because of lots of I/O delays. The I/O bound processes weren't affected by the preemption and they behaved just like they did in SJF scheduling. This is because they are sleeping most of the time. Their actual

execution time is very low. The likelihood of them experiencing quant clock ticks is very low since quant is set to be a 2-3 digit number. Hence, they didn't get forcefully preempted at regular intervals. They just went to sleep repeatedly. Therefore, their number of context switches remained the same as they would be in SJF scheduling.

```
$ test_one
CPU Bound(189707730) / Burst Time: 10 Context Switches: 2
CPU Bound(759489984) / Burst Time: 20 Context Switches: 3
CPU Bound(1887719312) / Burst Time: 50 Context Switches: 7
CPU Bound(602803054) / Burst Time: 60 Context Switches: 9
CPU Bound(321574720) / Burst Time: 70 Context Switches: 10
IO Bound / Burst Time: 30 Context Switches: 301
IO Bound / Burst Time: 40 Context Switches: 401
IO Bound / Burst Time: 80 Context Switches: 801
IO Bound / Burst Time: 90 Context Switches: 901
IO Bound / Burst Time: 100 Context Switches: 1001
$
```

(a) Test 1 for Hybrid Scheduler

```
$ test_two
CPU Bound(215443331) / Burst Time: 20 Context Switches: 3
CPU Bound(1262142690) / Burst Time: 30 Context Switches: 5
CPU Bound(1537590866) / Burst Time: 50 Context Switches: 8
CPU Bound(1829452763) / Burst Time: 90 Context Switches: 13
CPU Bound(389345552) / Burst Time: 100 Context Switches: 14
IO Bound / Burst Time: 10 Context Switches: 101
IO Bound / Burst Time: 40 Context Switches: 401
IO Bound / Burst Time: 60 Context Switches: 601
IO Bound / Burst Time: 70 Context Switches: 701
IO Bound / Burst Time: 80 Context Switches: 801
$
```

(b) Test 2 for Hybrid Scheduler

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**. The Figures below shows the order of execution of processes in the hybrid algorithm, along with the number of context switches for each process for both tests respectively.

In order to compare with Round Robin or SJF Scheduler, the figures from Part-B can be referred. From our observations we can conclude that Hybrid Scheduler does not let the bigger processes to starve while always waiting for the smaller processes to complete and it also decreases the average waiting time and average turn-around time for CPU bound processes.