OS-344

Assignment 2

Group Number : 2

Tejas Khairnar	180101081
Pooja Bhagat	180101057
Parth Bakare	180101056
Param Aryan Singh	180101055

14th October, 2020

PART A

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 data member 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 data member in process structure to store the burst time.
5. **get_burst_time :** Returns burst time of the calling process.

Files updated to add system calls and user programs

- **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).
- **syscall.c :** Contains helper functions to parse system call arguments, and pointers to the actual system call implementations.
- **syscall.h :** Contains a mapping from system call name to system call number.
- **user.h :** Contains the system call definitions in xv6 code was added here for the new system calls.
- **usys.S :** Contains a list of system calls exported by the kernel.
- **sysproc.c :** Contains the implementations of process related system calls. System call code was added here.
- **proc.c :** Contains the function scheduler which performs scheduling and context switching between processes.
- **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 system calls and user programs

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

Each user program was named as the system call's name that it tested.

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.
- 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**.
- 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();
```

- 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.
- 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
```

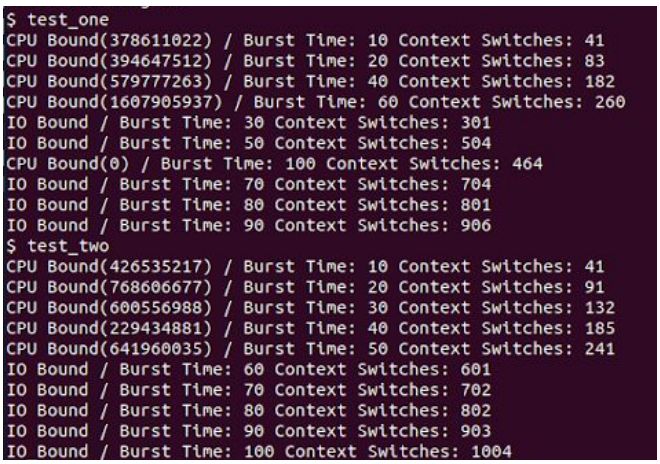
- We have iterated over all the processes which are **RUNNABLE** and chose the one with the smallest burst time for scheduling, if we have **n processes** then our scheduler has a time complexity of **O(n)**.

```
acquire(&ptable.lock);
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;
  switchuvm(p);
  p->state = RUNNING;
  swtch(&(c->scheduler), p->context);
  p->contextswitches = p->contextswitches + 1;
  switchkvm();
  c->proc = 0;
}
release(&ptable.lock);
```

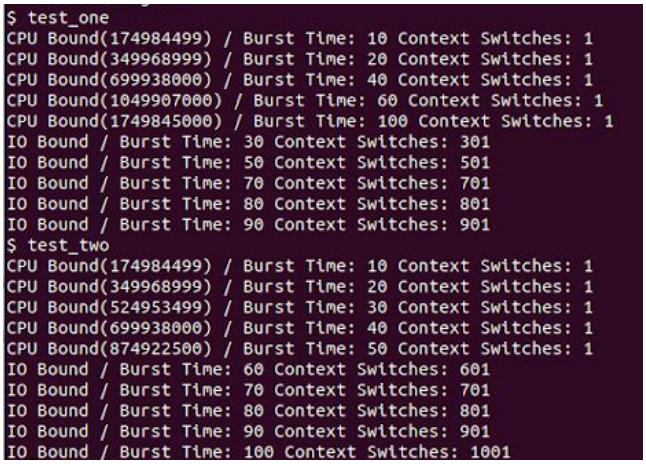
Testing the SJF scheduler in xv6 :

- 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 (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.
- The **screenshots** below show the order of execution of processes in the default Round Robin algorithm and the SJF algorithm.

Output for test_one & test_two



Round Robin algorithm



Shortest Job First algorithm

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.

```
// 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;
    }
  }
}
```

- 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();
           }
      }
}
```

- 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 :

- 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.
- The **screenshot** below shows the order of execution of processes in the hybrid algorithm, along with the number of context switches for each process.

Output for test_one & test_two

```
$ test_one
CPU Bound(472747854) / Burst Time: 10 Context Switches: 3
CPU Bound(940517224) / Burst Time: 20 Context Switches: 6
CPU Bound(1103082129) / Burst Time: 40 Context Switches: 11
CPU Bound(770788874) / Burst Time: 60 Context Switches: 16
CPU Bound(1155855985) / Burst Time: 100 Context Switches: 25
IO Bound / Burst Time: 30 Context Switches: 301
IO Bound / Burst Time: 50 Context Switches: 501
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(463759110) / Burst Time: 10 Context Switches: 3
CPU Bound(895099332) / Burst Time: 20 Context Switches: 6
CPU Bound(771189798) / Burst Time: 30 Context Switches: 10
CPU Bound(223578675) / Burst Time: 40 Context Switches: 11
CPU Bound(677000506) / Burst Time: 50 Context Switches: 14
IO Bound / Burst Time: 60 Context Switches: 601
IO Bound / Burst Time: 70 Context Switches: 701
IO Bound / Burst Time: 80 Context Switches: 801
IO Bound / Burst Time: 90 Context Switches: 901
IO Bound / Burst Time: 100 Context Switches: 1001
```