Operating Systems Lab Assignment - 2B

Group 22

Rishav Mondal (190101072)
Siddharth Charan (190101085)
Shrey Verma (190101083)
Karan Raj Sharma (190101043)

1 Task 1

In this task we were asked to analyze the current scheduling policy used by xv6 and also implement our own scheduling policies.
We can locate the code in the file `proc.c`.

```
494     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
495       if(p->state != RUNNABLE)
496         continue;
497
498       // Switch to chosen process.  It is the process's job
499       // to release ptable.lock and then reacquire it
500       // before jumping back to us.
501       c->proc = p;
502       switchuvm(p);
503       p->state = RUNNING;
504
505       swtch(&(c->scheduler), p->context);
506       switchkvm();
507
508       // Process is done running for now.
509       // It should have changed its p->state before coming back.
510       c->proc = 0;
```

We can clearly see in line 494 , that there is a loop which simply selects any process at the start of ready queue, which is ready for processing. Hence, we can say that default policy is round robin policy, and initial time quanta is set to one clock tick.
Now we try to answer the questions given to us:

- **Which process does the policy select for running?**
  The current scheduling policy in use by vanilla xv6 operating system is **Round Robin Scheduling**. The default algorithm implemented in xv6 it's one of the simplest (with FCFS) and relies on the Round-Robin policy, basically it loops through all the process which are available to run state and give access to CPU at each one of them one at a time. To schedule processes fairly, a round-robin scheduler generally employs time-sharing, giving each job a time slot or quantum (its allowance of CPU time), and interrupting the job if it is not completed by then. The job is resumed next time a time slot is assigned to that process. If the process terminates or changes its state to waiting during its attributed time quantum, the scheduler selects the first process in the ready queue to execute. In the absence of time-sharing, or if the quanta were large relative to the sizes of the jobs, a process that produced large jobs would be favoured over other processes. Round-robin scheduling is simple, easy to implement, and starvation-free.

- **What happens when a process returns from I/O?**
  When a process returns after completing it's I/O, then it is promoted from waiting state to ready state and put at the end of the round robin queue. Now the process awaits it's turn; and the scheduler always picks the first process at the head of the queue.

- **What happens when a new process is created?**
  When a new process is created it will go to the job queue; where it will await work. The number of processes in the job queue depends on the *degree of multiprogramming*. When there is available space inside the ready queue, then accordingly the process will be promoted to the ready queue.

- **When/how often does the scheduling take place?**
  Whenever the time quanta is over for a particular process(initially this was set to preemption at every clock tick), then that process is preempted out and placed at the tail of the ready queue. Also, when a process is over, then a new process can be promoted to the ready queue. Also during I/O, a process can be preempted out and then also scheduling takes place.

```
ifndef SCHEDFLAG
SCHEDFLAG := DEFAULT
endif
```

We now modify the `Makefile` to support **SCHEDFLAG** – a macro for quick compilation of the appropriate scheduling scheme. We also set the value to `DEFAULT`.

Now we will implement three other scheduling policies in addition to the default one. To achieve this we will heavily use the `ifdef` macros. For Task-1 the following files were edited:

1. `proc.h` : In recently created structure proc, we added another objects to extract more information.
2. `proc.c` : This is the main part of code. Here, we declared how scheduler works. Here, we used ifdef to distinguish between differentscheduling policies. We wrote different logic for each scheduling policy. Inside the scheduler function we defined the following:
   (a) **Default policy** : We slightly changed the code and increased time quanta for this. In this, file no code change for this particular policy was needed.

   (b) **FCFS** : We defined minP in proc.h, which will give us process with least time. Here, we used that to get our desired process.

   (c) **SML** : In this we used set_prio function defined in same proc.c file to see if thereis any process in higher priority queue. Hence, we are manually checking the priority. If it is there we will select process from that queue using FCFS.

   (d) **DML** : In this we simply twisted the code used in C part. Here, priority is also changing. Hence, we wrote logic for that too. We can note that in attached screenshot
3. `param.h`: we defined quanta here.
4. `Makefile` : We defined flag(to choose scheduling policy) here.
5. `sh.c` : This file is used to give conformation about choice of user regarding scheduling policy.
6. `syscall.h` : Here we added another systemcall at 24th position named sys_set_prio.
7. `sysproc.c` : For system call sys_set_prio we added code in this file. We simply trasfered the call to set_prio function defined as user call in user.h file.
8. `user.h` and `usys.S` : In these files we defined the user call set_prio which will be used for implementation of system call sys_set_prio.

```
#ifdef DEFAULT
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    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);
  switchkvm();

  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
}
#else
```

Figure 1: Default

```
#ifdef FCFS

  struct proc *minP = NULL;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)

  {

    if (p->state == RUNNABLE)

    {

      if (minP != NULL)

      {

        if (p->ctime < minP->ctime)

          minP = p;

      }

      else

        minP = p;

    }

  }

  if (minP != NULL)

  {

    p = minP; //the process with the smallest creation time

    c->proc = p;

    switchuvm(p);

    p->state = RUNNING;

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

    switchkvm();

    // Process is done running for now.

    // It should have changed its p->state before coming back.

    c->proc = 0;

  }
#else
```

Figure 2: FCFS

```
#ifdef SML
uint priority = 3;
p = findreadyprocess(&index1, &index2, &index3, &priority);
if (p == 0) {
  release(&ptable.lock);
  continue;
}
c->proc = p;
switchuvm(p);
p->state = RUNNING;
swtch(&(c->scheduler), p->context);
switchkvm();
c->proc = 0;
#else
```

Figure 3: SML

```
#ifdef DML
uint priority = 3;
p = findreadyprocess(&index1, &index2, &index3, &priority);
if (p == 0) {
  release(&ptable.lock);
  continue;
}
proc = p;
switchuvm(p);
p->state = RUNNING;
p->tickcounter = 0;
swtch(&cpu->scheduler, proc->context);
switchkvm();
proc = 0;
#endif
```

Figure 4: DML

2 Task 2

In this task we shall be adding the `yield` system call. The job of the `yield` system call is to yield execution to another process. For this we make change inside the following files:

1. `proc.c` : Here we include our code for the `yield()` function which takes no arguments and returns 1 if yielding execution to another process is successful else 0. Here we give up the CPU for one scheduling round (make state of some other process = `RUNNABLE`) and make sure we do it inside a lock.
2. `user.h`: Here we define the user level system call definition.
3. `syscall.c`: Here we define the position of system call inside the system call vector.
4. `sysproc.c`: Here we define the system call `int sys_yield(void)`

```c
// Give up the CPU for one scheduling round.
void yield(void)
{
  acquire(&ptable.lock); //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

```c
int sys_yield(void) {
  yield();
  return 0;
}
```

3 Task 3

In this particular task we added two user-space programs (`sanity.c` and `SMLsanity.c`) that will test the impact of each scheduling policy. We need to add both of these programs in the `UPROGS` section of the `Makefile`. Now in `sanity.c`, the `main()` function accepts the a number n as argument and then fork($3\times n$) processes and wait till all of them finish, for each child process that end print its statistics. Now we have a switch case which takes decision regarding what to do for process of each type. Each of the processes can be of three types : **CPU bound processes**, **short task based CPU bound process**, **I/O bound process**.
We create `SMLsanity.c` for **static multilevel queue scheduling** as for different CPU bound processes we give them different priorities manually. Here we also added a `fact(int x)` function as if we did not add anything inside the dummy for loop then due to compiler optimizations we get the runtime as 0.

3.1 Analysis

1. Only **I/O bound processes** have a non zero sleeping time (waiting time for I/O) as expected, as no other process is waiting for I/O.
2. **FCFS** or **first come first serve scheduling** as the name suggests serves the processes coming earlier first, so the process coming first will have 0 ready time and the processes coming later will have to wait in the ready queue, and thus have non-zero ready time.
3. In the default (**Round Robin**) scheduling, preemption will take place after a fixed quantum of 5 time units. Short Processes will benefit from this and will terminate soon; whereas longer processes will be context switched multiple times.
4. In **SML** Scheduling, first the processes are scheduled to one of the queues and each queue has a different priority and queue is selected according to their priorities and within each queue, **FCFS** scheduling is done.
5. In **DML** Scheduling is like SML Scheduling but in this process can move between the queues. DML keeps analyzing the behavior (time of execution) of processes and according to which it changes its priority.

3.2 Printing the statistics

For each terminated process, we were told to print:

- The process id and it's type (CPU / S-CPU / IO)
- The wait time, run time and I/O time
- Sleep time for each type
- Ready time for each type
- Turnaround time for each type

Following are the screenshots which illustrate this:

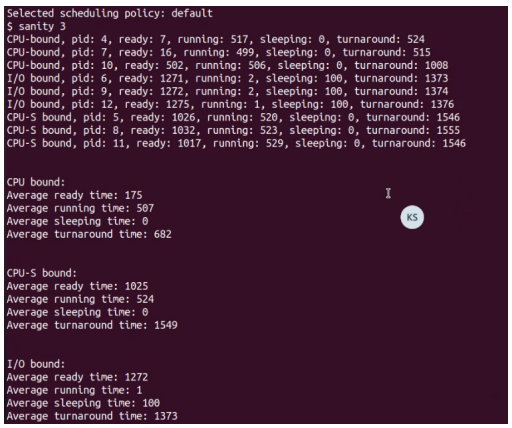

Figure 5: Default Sanity

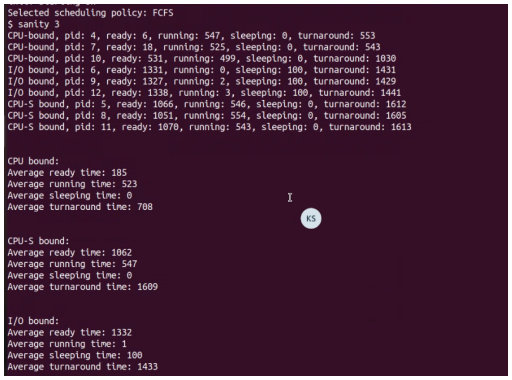

Figure 6: FCFS Sanity

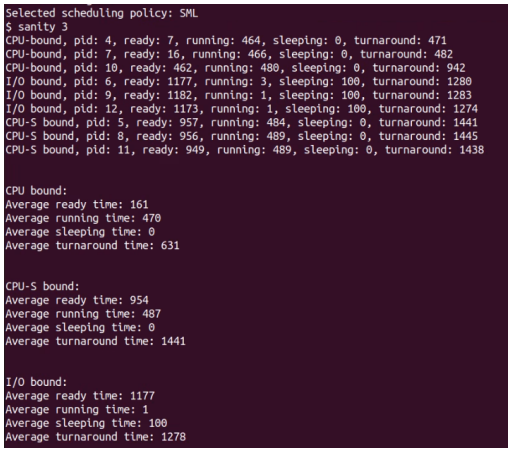

Figure 7: SML Sanity

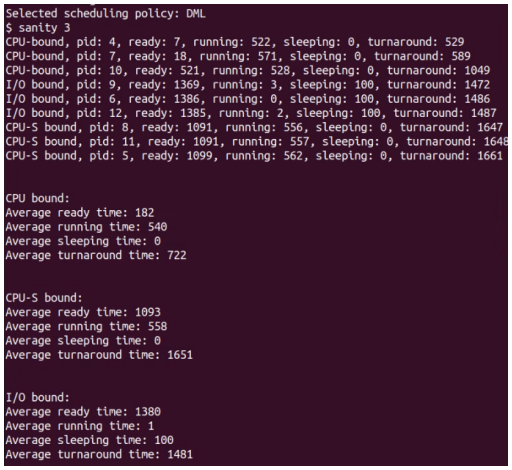

Figure 8: DML Sanity

Screenshot for `SMLsanity` used for testing task 3:

```
$ SMLsanity 3
Priority 2, pid: 14, ready: 0, running: 468, sleeping: 0, turnaround: 468
Priority 3, pid: 15, ready: 4, running: 463, sleeping: 0, turnaround: 467
Priority 2, pid: 17, ready: 466, running: 477, sleeping: 0, turnaround: 943
Priority 3, pid: 18, ready: 466, running: 467, sleeping: 0, turnaround: 933
Priority 1, pid: 19, ready: 938, running: 479, sleeping: 0, turnaround: 1417
Priority 2, pid: 20, ready: 936, running: 463, sleeping: 0, turnaround: 1399
Priority 1, pid: 16, ready: 1433, running: 456, sleeping: 0, turnaround: 1889
Priority 3, pid: 21, ready: 1399, running: 452, sleeping: 0, turnaround: 1851
Priority 1, pid: 22, ready: 1846, running: 438, sleeping: 0, turnaround: 2284


Priority 1:
Average ready time: 1405
Average running time: 457
Average sleeping time: 0
Average turnaround time: 1862


Priority 2:
Average ready time: 467
Average running time: 469
Average sleeping time: 0
Average turnaround time: 936


Priority 3:
Average ready time: 623
Average running time: 460
Average sleeping time: 0
Average turnaround time: 1083
```

4 For running the code

To run the code we need to run:

1. `make clean`
2. `make`
3. `make qemu SCHEDFLAG = <scheduling policy name>`

When we run these commands we can see in the terminal that particular scheduling policy has been successfully selected.

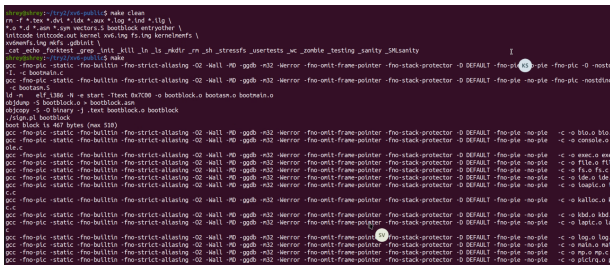

Figure 9: using make and make clean

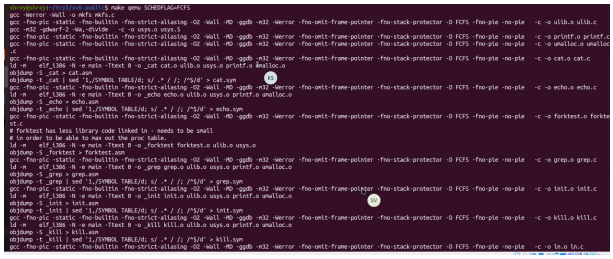

Figure 10: make clean in FCFS