**Report**

**Part 1:**

usr/src/include/minix/include/com.h -> Set\_tickets and set\_torpil are added.

usr/src/minix/servers/pm/schedule.c -> sched\_tickets and sched\_torpil are added.

usr/src/minix/servers/pm/proto.h -> setticket and settorpil prototypes are added.

usr/src/minix/servers/sched/main.c -> Where we handle the system call sent through pm

usr/src/minix/servers/sched/schedule.c -> Where lottery is implemented and do\_torpil, do\_tickets

usr/src/minix/config.h -> Where SCHEDULING\_SET\_TICKETS and TORPIL added.

usr/src/include/unistd.h -> User libraries

usr/src/minix/kernel/proc.c -> Where torpil is implemented.

usr/src/minix/servers/sched/schedproc.h -> Ticket number torpil and is\_sys\_proc added.

**Modifications**

When system call settickets is called, it goes to pm. The related file setticket.c gets the call. Since we cannot reach directly to sched, which has the tickets information, we need to send our call through pm to sched and handle the call there. We modified schedule.c in pm for this purpose and added sched\_tickets just like sched\_nice which sends a call to sched. We catch that call in main.c in sched and send it to schedule.c in sched where we created do\_tickets which changes the ticket number of a process. In order to find which process we used endpoint of the calling process to get the process number and sent the information all the way to do\_tickets. Settorpil is exactly the same logic.

We added 3 fields to schedproc.h in sched. tickets, torpil and is\_sys\_proc, which holds number of tickets, whether a process has torpil, and whether a process is system process or user process relatively.

We added lottery method to schedule.c in sched which basically chooses a winner and puts it to 12th queue. We modified do\_no\_quantum and do\_start\_scheduling and also other methods to be able to implement the algorithm. For futher information you can look at the comments in the code since it is hard to explain things seperately here.

For torpil case we changed proc.c in kernel basically if 14th queue has any process we chose it before 12th and 13th.

**Part 2:**

**Programs:**

Cpu bound program : Basically this does ALU operation in a while loop for so many times so that we can see the difference. This program is cpu bounded because it only does ALU operation which requires cpu usage.

IO bound program: This program writes to a file for so many iterations. We write to the same output1.txt to avoid using too much memory. This is a io bounded program because most of the time it makes io request to write to a file.

**Measuring Time:**

In order to measure the times to a log file we print when they started and when they finished. (For the first scenario we only print total running time in logs.). Cpu bounded processes' logs are put into CPULogTest.txt and io bounded processes' logs are put into IOLogTest.txt.. In log files you will see the number of a process also pid of a process with start and finish times.

**Running Programs:**

Each scenario has scripts. Simply write "sh cpuTest.sh" for cpu bounded processes and "sh ioTest.sh" for io bounded processes.

**Input Parameters:**

You should pass a number between 1-30 both inclusive to settickets. If you pass something less than 1 than it will set it to 1. If you pass something greater than 30 then it will set it to 30.

You should pass 0 or 1 to settorpil.

Initial ticket number of a process is 10.

Winner process is put to 12 th queue. Loser processes are in 13 th queue. Torpilled processes are in 14 th queue.

**Results:**

**Scenario 1:**

We need to test that their running times are almost equal since they will have the same amount of tickets. For **CPU Bound process:**

**Run 1:**

Number 3:1234 has finished ; Running time: 2.620000

Number 2:1233 has finished ; Running time: 2.630000

Number 1:1232 has finished ; Running time: 2.630000

**Run 2:**

Number 1:1238 has finished ; Running time: 2.620000

Number 3:1240 has finished ; Running time: 2.610000

Number 2:1239 has finished ; Running time: 2.640000

For **IO Bound Process:**

**Run 1:**

Number 1 pid 1159: finished ;Running time: 1.880000

Number 3 pid 1161: finished ;Running time: 2.000000

Number 2 pid 1160: finished ;Running time: 1.940000

**Run 2:**

Number 1 pid 1165: finished ;Running time: 1.710000

Number 3 pid 1167: finished ;Running time: 1.820000

Number 2 pid 1166: finished ;Running time: 1.880000

**Run 3:**

Number 2 pid 1171: finished ;Running time: 1.730000

Number 3 pid 1172: finished ;Running time: 1.960000

Number 1 pid 1170: finished ;Running time: 1.940000

We see that they are almost equal to each other. So lottery algorithm works fine with the scenario.

**Scenario 2:**

We need to test when a process has more tickets it will finish before than others.

For a **CPU bound process:**

**Run 1:**

Number 3 with 27 tickets 454 has started at Sat Dec 24 02:31:49 2016

Number 3 with 27 tickets:454 has finished at Sat Dec 24 02:32:04 2016

Number 2 with 15 tickets 453 has started at Sat Dec 24 02:31:48 2016

Number 2 with 15 tickets:453 has finished at Sat Dec 24 02:32:07 2016

Number 1 with 5 tickets 452 has started at Sat Dec 24 02:31:49 2016

Number 1 with 5 tickets:452 has finished at Sat Dec 24 02:32:14 2016

**Run 2:**

Number 3 with 27 tickets 458 has started at Sat Dec 24 02:32:29 2016

Number 3 with 27 tickets:458 has finished at Sat Dec 24 02:32:41 2016

Number 2 with 15 tickets 457 has started at Sat Dec 24 02:32:28 2016

Number 2 with 15 tickets:457 has finished at Sat Dec 24 02:32:48 2016

Number 1 with 5 tickets 456 has started at Sat Dec 24 02:32:29 2016

Number 1 with 5 tickets:456 has finished at Sat Dec 24 02:32:55 2016

For **IO bounded process:**

**Run 1:**

Number 3 with 30 tickets has started 421 pid at Sat Dec 24 02:24:45 2016

Number 3 with 30 tickets pid 421: finished Sat Dec 24 02:24:52 2016

Number 2 with 14 tickets has started 420 pid at Sat Dec 24 02:24:45 2016

Number 2 with 14 tickets pid 420: finished Sat Dec 24 02:24:54 2016

Number 1 with 3 tickets has started 419 pid at Sat Dec 24 02:24:46 2016

Number 1 with 3 tickets pid 419: finished Sat Dec 24 02:24:58 2016

**Run 2:**

Number 3 with 30 tickets has started 427 pid at Sat Dec 24 02:25:38 2016

Number 3 with 30 tickets pid 427: finished Sat Dec 24 02:25:44 2016

Number 2 with 14 tickets has started 426 pid at Sat Dec 24 02:25:37 2016

Number 2 with 14 tickets pid 426: finished Sat Dec 24 02:25:48 2016

Number 1 with 3 tickets has started 425 pid at Sat Dec 24 02:25:36 2016

Number 1 with 3 tickets pid 425: finished Sat Dec 24 02:25:50 2016

From these tests for both io and cpu bounded programs we see that process with more tickets finishes before. So settickets is implemented correctly, and lottery is implemented correctly.

**Scenario 3:**

Here we need to test when we give torpil the process with torpil will finish before other processes without torpil no matter what their ticket number is. In order to test this we simply choose the process with least number of tickets and give it a torpil.

For a **CPU bounded process:**

**Run 1:**

Number 1 with 5 tickets WITH TORPIL 511 has started at Sat Dec 24 02:40:32 2016

Number 1 with 5 tickets WITH TORPIL :511 has finished at Sat Dec 24 02:40:41 2016

Number 2 with 15 tickets 512 has started at Sat Dec 24 02:40:36 2016

Number 2 with 15 tickets:512 has finished at Sat Dec 24 02:40:57 2016

Number 3 with 27 tickets 513 has started at Sat Dec 24 02:40:41 2016

Number 3 with 27 tickets:513 has finished at Sat Dec 24 02:40:57 2016

**Run 2:**

Number 1 with 5 tickets WITH TORPIL 517 has started at Sat Dec 24 02:41:53 2016

Number 1 with 5 tickets WITH TORPIL :517 has finished at Sat Dec 24 02:42:03 2016

Number 3 with 27 tickets 519 has started at Sat Dec 24 02:41:56 2016

Number 3 with 27 tickets:519 has finished at Sat Dec 24 02:42:16 2016

Number 2 with 15 tickets 518 has started at Sat Dec 24 02:42:06 2016

Number 2 with 15 tickets:518 has finished at Sat Dec 24 02:42:19 2016

We see that process with torpil finishes before even though it has least number of tickets.

For **IO bounded process:**

**Run 1:**

Number 1 with 3 tickets WITH TORPIL has started 534 pid at Sat Dec 24 02:44:15 2016

Number 1 with 3 tickets WITH TORPIL pid 534: finished Sat Dec 24 02:44:20 2016

Number 3 with 30 tickets has started 536 pid at Sat Dec 24 02:44:14 2016

Number 3 with 30 tickets pid 536: finished Sat Dec 24 02:44:26 2016

Number 2 with 14 tickets has started 535 pid at Sat Dec 24 02:44:14 2016

Number 2 with 14 tickets pid 535: finished Sat Dec 24 02:44:27 2016

**Run 2:**

Number 1 with 3 tickets WITH TORPIL has started 538 pid at Sat Dec 24 02:44:30 2016

Number 1 with 3 tickets WITH TORPIL pid 538: finished Sat Dec 24 02:44:35 2016

Number 2 with 14 tickets has started 539 pid at Sat Dec 24 02:44:30 2016

Number 2 with 14 tickets pid 539: finished Sat Dec 24 02:44:43 2016

Number 3 with 30 tickets has started 540 pid at Sat Dec 24 02:44:35 2016

Number 3 with 30 tickets pid 540: finished Sat Dec 24 02:44:43 2016

We see that processes with torpil finish before than others.

**Note:** The reason that in the second run ticket 14 has finished before 30 is because the iteration number is not that much, so it is like that by chance. In order to get better results one can increase the number of iterations, but it is not a concern since it is not the test purpose of this scenario.

**Scenario 4:**

We need to test if dynamic is 1 in schedule.c in sched then IO bounded processes' tickets will increase dynamically and cpu bounded processes' tickets will decrease. For this purpose we test it with dynamic field = 0 and with 1:

**When dynamic is 0:**

**Run 1:**

CPU BOUND with initially 20 tickets pid 359 started at Wed Dec 21 19:21:08 2016

CPU BOUND with initial 20 tickets:359 has finished at Wed Dec 21 19:21:25 2016

IO BOUND with initially 10 tickets pid 358: started at Wed Dec 21 19:21:08 2016

IO BOUND with initially 10 tickets pid 358: finished at : Wed Dec 21 19:21:34 2016

**Run 2:**

CPU BOUND with initially 20 tickets pid 364 started at Wed Dec 21 19:21:46 2016

CPU BOUND with initial 20 tickets:364 has finished at Wed Dec 21 19:22:04 2016

IO BOUND with initially 10 tickets pid 363: started at Wed Dec 21 19:21:46 2016

IO BOUND with initially 10 tickets pid 363: finished at : Wed Dec 21 19:22:12 2016

**Run 3:**

CPU BOUND with initially 20 tickets pid 367 started at Wed Dec 21 19:22:18 2016

CPU BOUND with initial 20 tickets:367 has finished at Wed Dec 21 19:22:38 2016

IO BOUND with initially 10 tickets pid 366: started at Wed Dec 21 19:22:19 2016

IO BOUND with initially 10 tickets pid 366: finished at : Wed Dec 21 19:22:46 2016

**When dynamic is 1 :**

**Run 1:**

IO BOUND with initially 10 tickets pid 245: started at Wed Dec 21 19:27:02 2016

IO BOUND with initially 10 tickets pid 245: finished at : Wed Dec 21 19:27:25 2016

CPU BOUND with initially 20 tickets pid 246 started at Wed Dec 21 19:27:02 2016

CPU BOUND with initial 20 tickets:246 has finished at Wed Dec 21 19:27:28 2016

**Run 2:**

IO BOUND with initially 10 tickets pid 254: started at Wed Dec 21 19:28:51 2016

IO BOUND with initially 10 tickets pid 254: finished at : Wed Dec 21 19:29:14 2016

CPU BOUND with initially 20 tickets pid 255 started at Wed Dec 21 19:28:51 2016

CPU BOUND with initial 20 tickets:255 has finished at Wed Dec 21 19:29:17 2016

We see that when it is not dynamic CPU bounded processes finish before but when it is dynamic IO bounded processes finish before.