CS344 Assignment 2B

Group members:

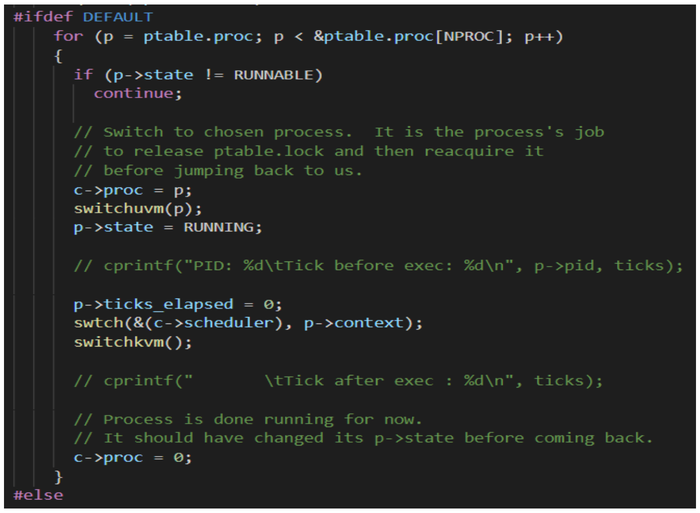
★ Saket Kumar Singh	190101081
★ Pradnesh Prasad Kalkar	190101103
★ Mesharya M Choudhary	190101053
★ Parag Panigrahi	190101111

Task 1

1. Scheduling

Q 1. Which process does the policy select for running?

A. xv6 is following a round-robin policy here. The time quantum after which a process is context switched is 1 tick. It can be seen that in the function scheduler, we are having an infinite for loop and inside that we have one more for loop over the array of processes - ptable.proc. This array is acting as the queue for the round robin. And we are pre-empting processes after every clock tick which can be seen in the file trap.c.



```
#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;

      // cprintf("PID: %d\tTick before exec: %d\n", p->pid, ticks);

      p->ticks_elapsed = 0;
      swtch(&(c->scheduler), p->context);
      switchkvm();

      // cprintf("       \tTick after exec : %d\n", ticks);

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

Q 2. What happens when a process returns from I/O?

A. There are a number of events triggered spanning across various files which are summarized as a table below:

S.No.	File	Event(s)
1.	trap.c	Handle the interrupt by invoking the appropriate ISR (Interrupt Service Routine)
2.	trap.c	Call exit() if the current process has exhausted its CPU cycles.
3.	trap.c	Call yield() if the process has to be preempted which changes the state to RUNNABLE (i.e Ready state) and calls sched() in proc.c
4.	proc.c	Restores intena (interrupt enable) and context switch the current process with next process given by scheduler (by calling swtch).
5.	swtch.S	Restores the state registers after the interrupt of I/O has been served to resume the execution of the next process.

Q 3. What happens when a new process is created?

A. There are a number of events triggered when a new process is created using the fork method in proc.c:

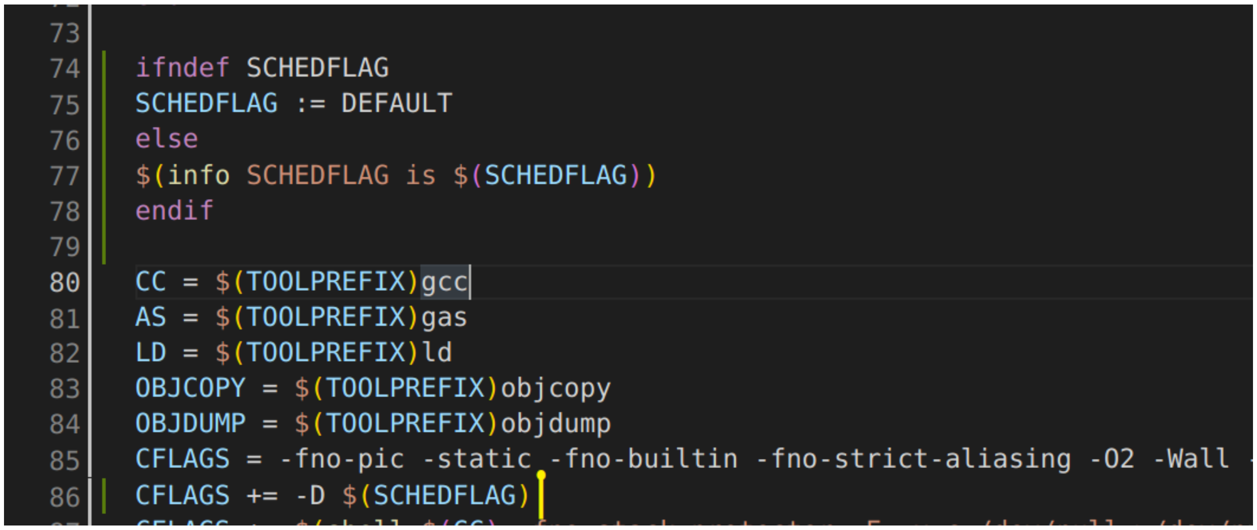
- allocproc: This allocates the process by finding the first UNUSED process in the ptable and changing it's state to EMBRYO. Following that, it initializes the kernel stack, stack pointer and context of the process to default values.
- setupkvm: This set's the kernel part of the page table (i.e the kernel virtual memory).
- inituvm: This loads the initcode to the address 0 of the page (pgdir) (i.e initializing the user virtual memory).

Q. 4 When/how often does the scheduling take place?

A. The default policy is round robin and the default time quanta is 1 clock tick. Thus the scheduler is invoked after every clock tick.

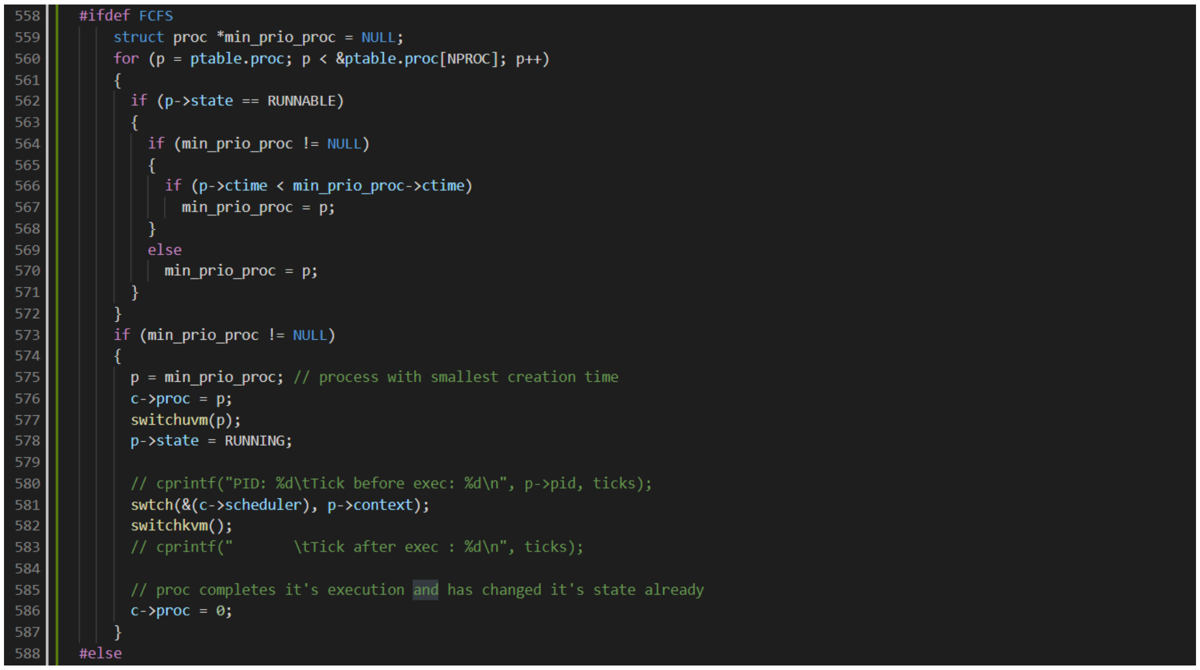
Makefile modifications done - addition of SCHEDFLAG

The first block checks whether SCHEDFLAG is defined (either as a command line argument or as a result of the previous make statement). If it isn't then it is set to DEFAULT. If not then, an info statement is printed for the user's reference informing the value of SCHEDFLAG. In line 86, the -D and the SCHEDFLAG arguments are added to the CFLAGS so that the it is defined and the #ifdef statements with the SCHEDFLAG are true in the source code.



```
73
74  ifndef SCHEDFLAG
75  SCHEDFLAG := DEFAULT
76  else
77  $(info SCHEDFLAG is $(SCHEDFLAG))
78  endif
79
80  CC = $(TOOLPREFIX)gcc
81  AS = $(TOOLPREFIX)gas
82  LD = $(TOOLPREFIX)ld
83  OBJCOPY = $(TOOLPREFIX)objcopy
84  OBJDUMP = $(TOOLPREFIX)objdump
85  CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall
86  CFLAGS += -D $(SCHEDFLAG)
```

FCFS policy -

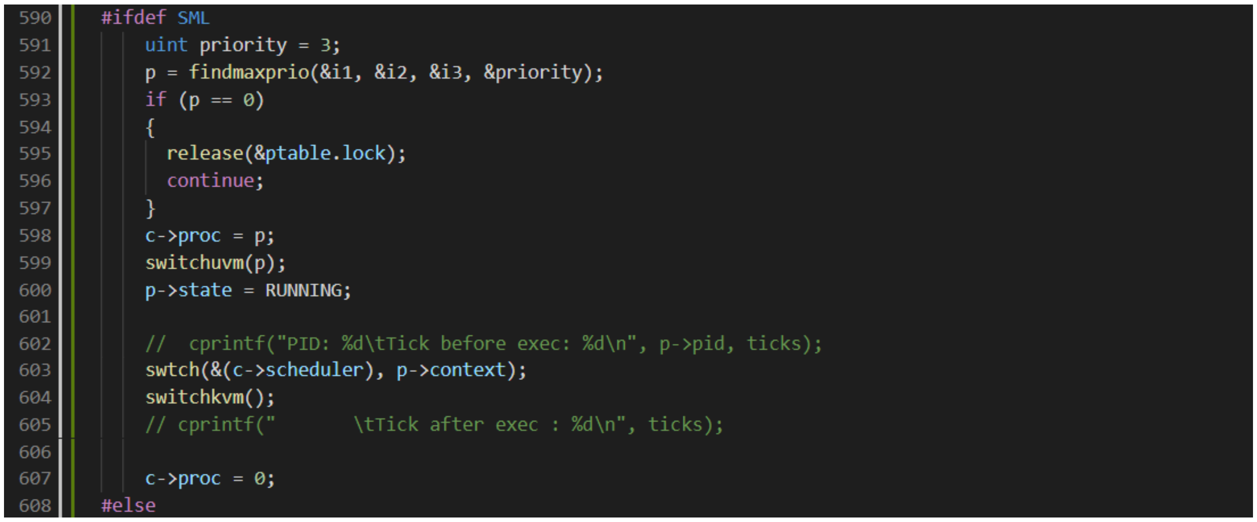


```
#ifdef FCFS
    struct proc *min_prio_proc = NULL;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state == RUNNABLE)
      {
        if (min_prio_proc != NULL)
        {
          if (p->ctime < min_prio_proc->ctime)
            min_prio_proc = p;
        }
        else
          min_prio_proc = p;
      }
    }
    if (min_prio_proc != NULL)
    {
      p = min_prio_proc; // process with smallest creation time
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      // cprintf("PID: %d\tTick before exec: %d\n", p->pid, ticks);
      swtch(&(c->scheduler), p->context);
      switchkvm();
      // cprintf("       \tTick after exec : %d\n", ticks);

      // proc completes it's execution and has changed it's state already
      c->proc = 0;
    }
#else
```

A. In this we simply loop over all the processes in the process table and note the process with state = RUNNABLE and with least creation time. If the process is NULL, i.e there aren't any RUNNABLE processes available, then don't do anything, else just switch to the process we just found.

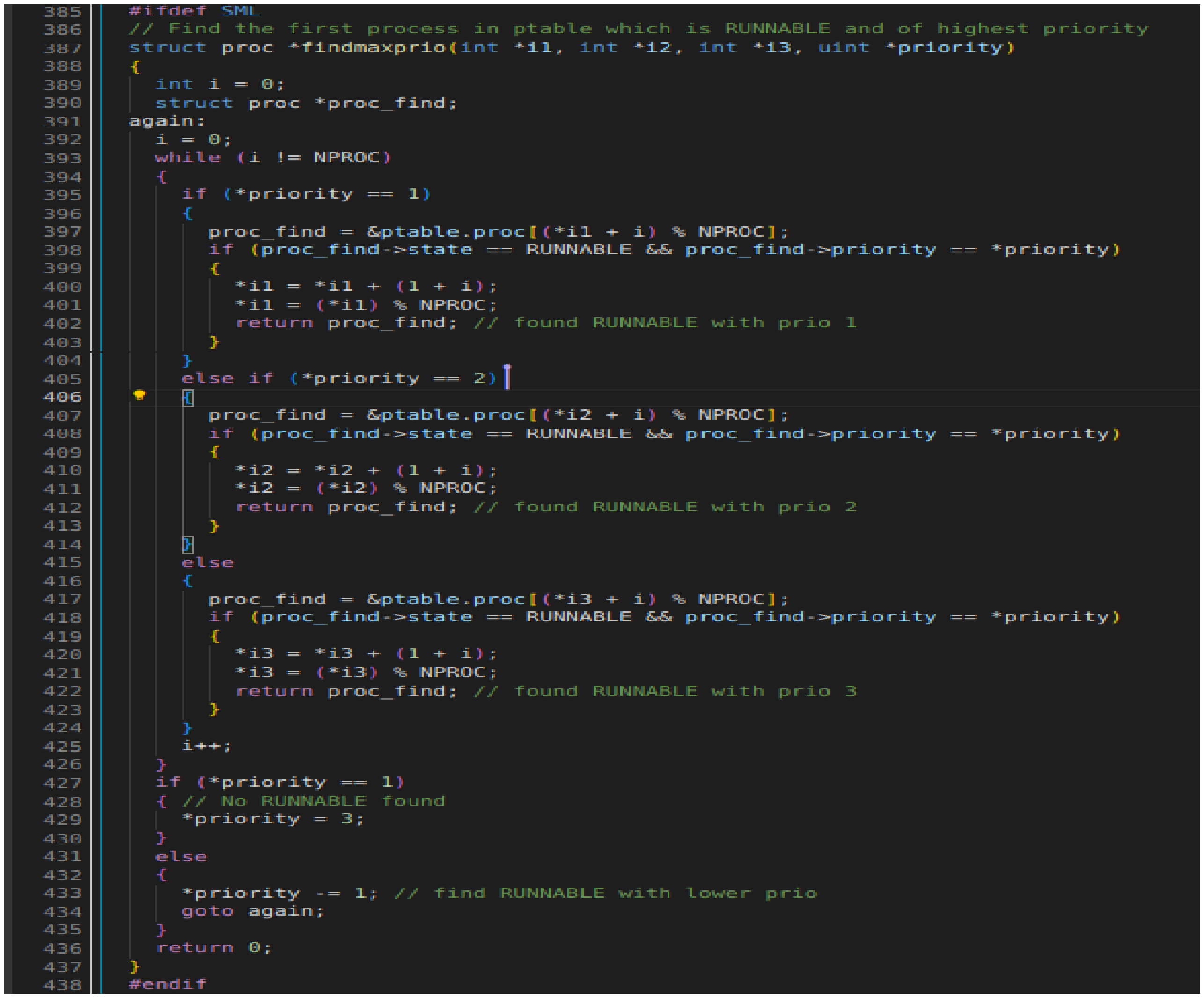
SML policy -

In this Static Multi-Level queue scheduling, we first find a process with highest priority using the findmaxprio() function.



```
#ifdef SML
    uint priority = 3;
    p = findmaxprio(&i1, &i2, &i3, &priority);
    if (p == 0)
    {
      release(&ptable.lock);
      continue;
    }
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

    //  cprintf("PID: %d\tTick before exec: %d\n", p->pid, ticks);
    swtch(&(c->scheduler), p->context);
    switchkvm();
    // cprintf("       \tTick after exec : %d\n", ticks);

    c->proc = 0;
#else
```

findmaxprio(): This function takes initially 3 as the priority parameter, and searches for processes of that priority. If any process is found then it returns that process, else the priority is decremented by one and the search continues. If no process is found for all the three priorities, then we just return 0.



```c
#ifdef SML
// Find the first process in ptable which is RUNNABLE and of highest priority
struct proc *findmaxprio(int *i1, int *i2, int *i3, uint *priority)
{
  int i = 0;
  struct proc *proc_find;
again:
  i = 0;
  while (i != NPROC)
  {
    if (*priority == 1)
    {
      proc_find = &ptable.proc[(*i1 + i) % NPROC];
      if (proc_find->state == RUNNABLE && proc_find->priority == *priority)
      {
        *i1 = *i1 + (1 + i);
        *i1 = (*i1) % NPROC;
        return proc_find; // found RUNNABLE with prio 1
      }
    }
    else if (*priority == 2)
    {
      proc_find = &ptable.proc[(*i2 + i) % NPROC];
      if (proc_find->state == RUNNABLE && proc_find->priority == *priority)
      {
        *i2 = *i2 + (1 + i);
        *i2 = (*i2) % NPROC;
        return proc_find; // found RUNNABLE with prio 2
      }
    }
    else
    {
      proc_find = &ptable.proc[(*i3 + i) % NPROC];
      if (proc_find->state == RUNNABLE && proc_find->priority == *priority)
      {
        *i3 = *i3 + (1 + i);
        *i3 = (*i3) % NPROC;
        return proc_find; // found RUNNABLE with prio 3
      }
    }
    i++;
  }
  if (*priority == 1)
  { // No RUNNABLE found
    *priority = 3;
  }
  else
  {
    *priority -= 1; // find RUNNABLE with lower prio
    goto again;
  }
  return 0;
}
#endif
```

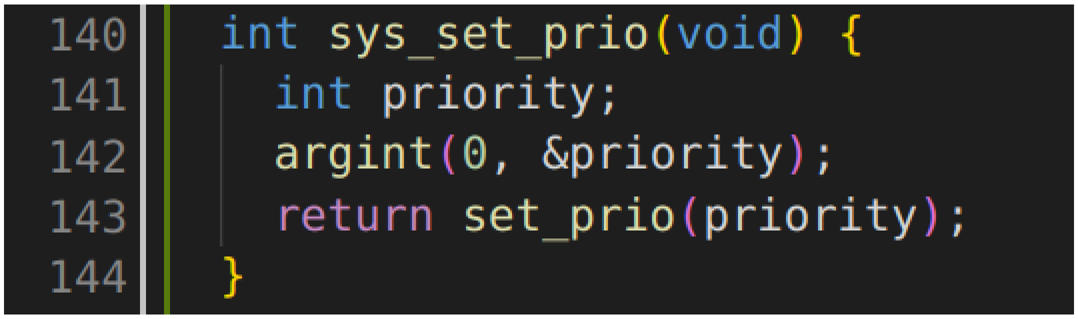
After getting the process from the above function, we check if it is 0 or not. If 0, then we just release the process table lock and return. Else the process's state is changed to RUNNING and we context switch to the new process using the swtch() function.

set_prio() system call -

For creating the set_prio() system call we had to change these files-

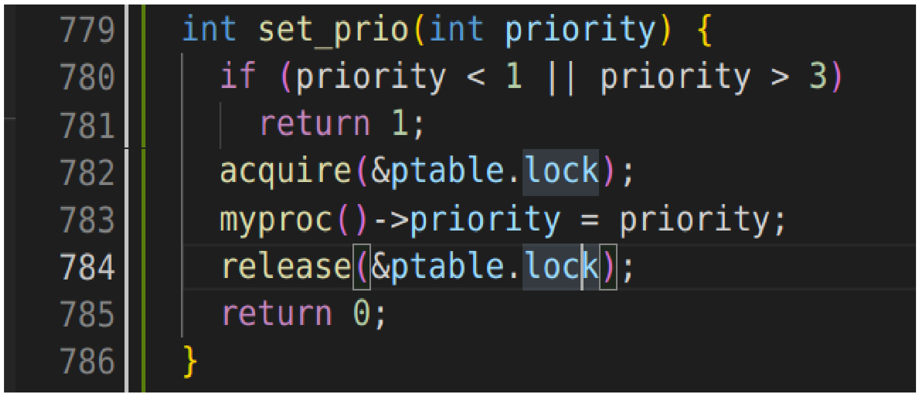
1. user.h - The function prototype for set_prio system call (for user-space) is added in this file. (declaration of set_prio function).
2. usys.S - The system call name (SYS_set_prio) is added in this file.
3. syscall.h - The mapping from system call name (SYS_set_prio) to system call number (23) is added in this file.
4. syscall.c - The mapping from system call number (23) to system call function (sys_set_prio) is added in this file.

5. sysproc.c - The definition of the sys_set_prio system call is added in this file as shown below. This will read the priority parameter value from set_prio defined in user.h. And in line 143, it calls set_prio function from defs.h (kernel-space declaration).



```
140  int sys_set_prio(void) {
141    int priority;
142    argint(0, &priority);
143    return set_prio(priority);
144  }
```

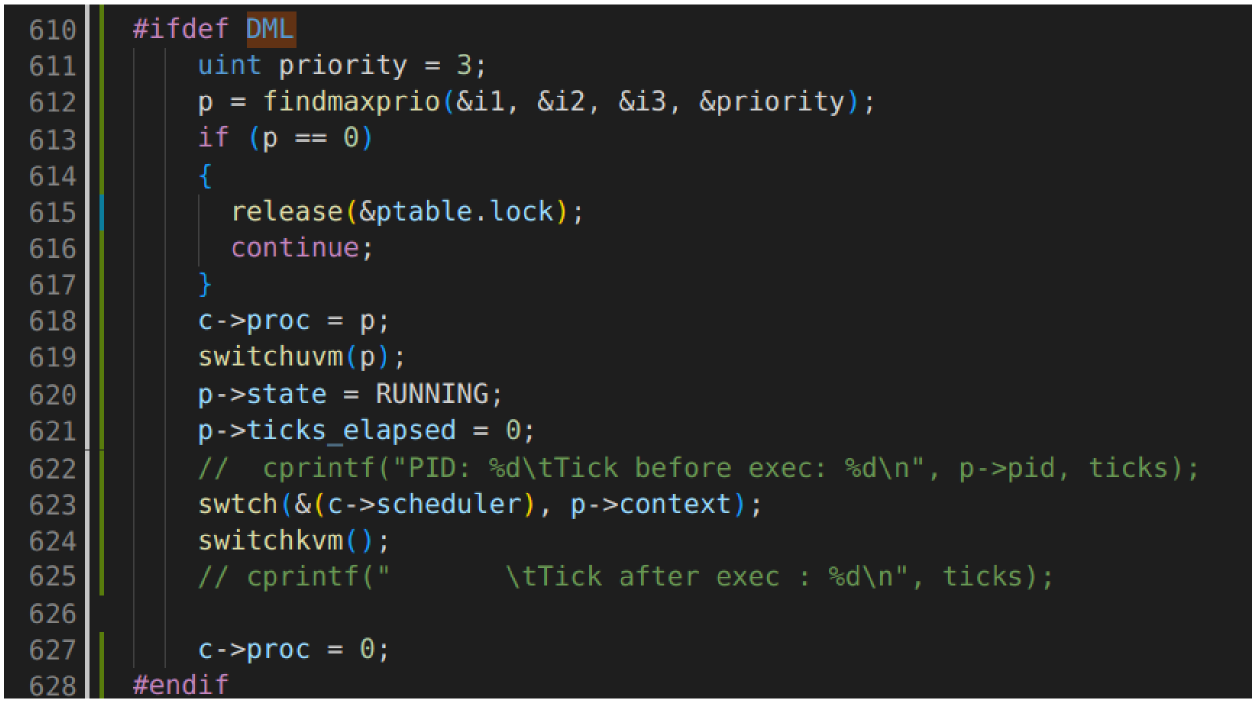
6. defs.h and proc.c - The function prototype for set_prio system call (for kernel-space) is added in the file defs.h (line 124); and this is implemented in the file proc.c as shown in the figure below. In this definition, first we check for the correctness of the priority value (it should be between 1 and 3), if incorrect we return 1. Else, we acquire the lock on the process table, modify the priority of the current process by assigning it the value of the parameter priority and then release the lock. And then return 0.



```
int set_prio(int priority) {
  if (priority < 1 || priority > 3)
    return 1;
  acquire(&ptable.lock);
  myproc()->priority = priority;
  release(&ptable.lock);
  return 0;
}
```

DML policy -

The working in DML is similar to that of SML. In addition to that, we maintain the process's tickcount (represented by p->tick_elapsed). Before we send it to the dispatcher, we set the tick_elapsed to 0. Then for every clock tick, the value gets incremented. After it reaches the QUANTA (which is 5 as defined in param.h), we decrement the priority of the process and again the scheduling takes place.



```
#ifdef DML
    uint priority = 3;
    p = findmaxprio(&i1, &i2, &i3, &priority);
    if (p == 0)
    {
      release(&ptable.lock);
      continue;
    }
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    p->ticks_elapsed = 0;
    //  cprintf("PID: %d\tTick before exec: %d\n", p->pid, ticks);
    swtch(&(c->scheduler), p->context);
    switchkvm();
    // cprintf("       \tTick after exec : %d\n", ticks);

    c->proc = 0;
#endif
```

Task 2

2. Yield System Call

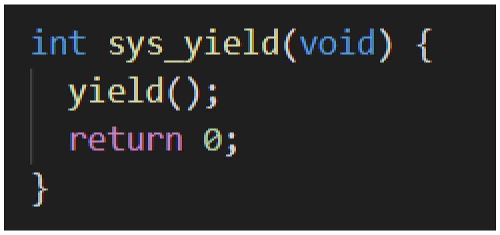
System call implementation of yield() -

For creating the yield() system call we had to change these files-

1. user.h - The function prototype for yield system call (for user-space) is added in this file. (declaration of yield function).
2. syscall.h - The mapping from system call name (SYS_yield) to system call number (24) is added in this file.
3. syscall.c - The mapping from system call number (24) to system call function

(sys_yield) is added in this file.
4. usys.S - The system call name is added in this file.
5. sysproc.c - The definition of the sys_yield system call is added in this file.

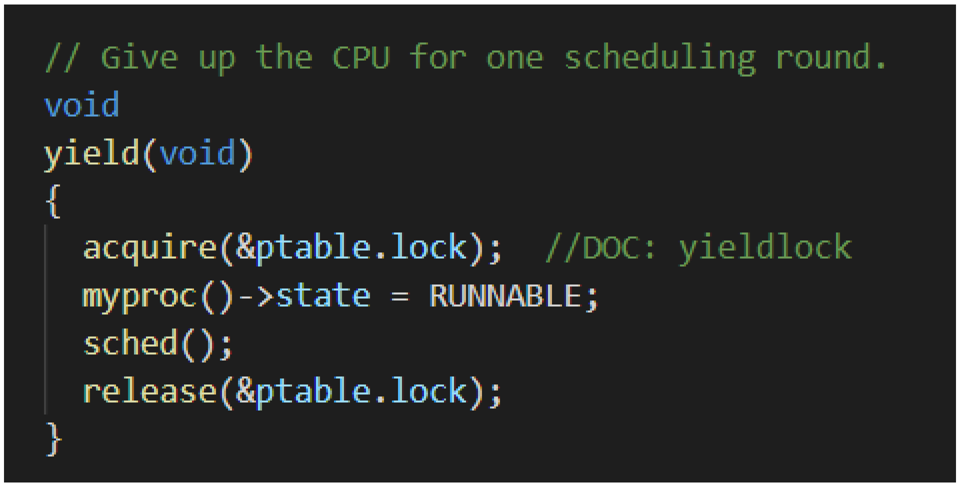
System call (sys_yield) -



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

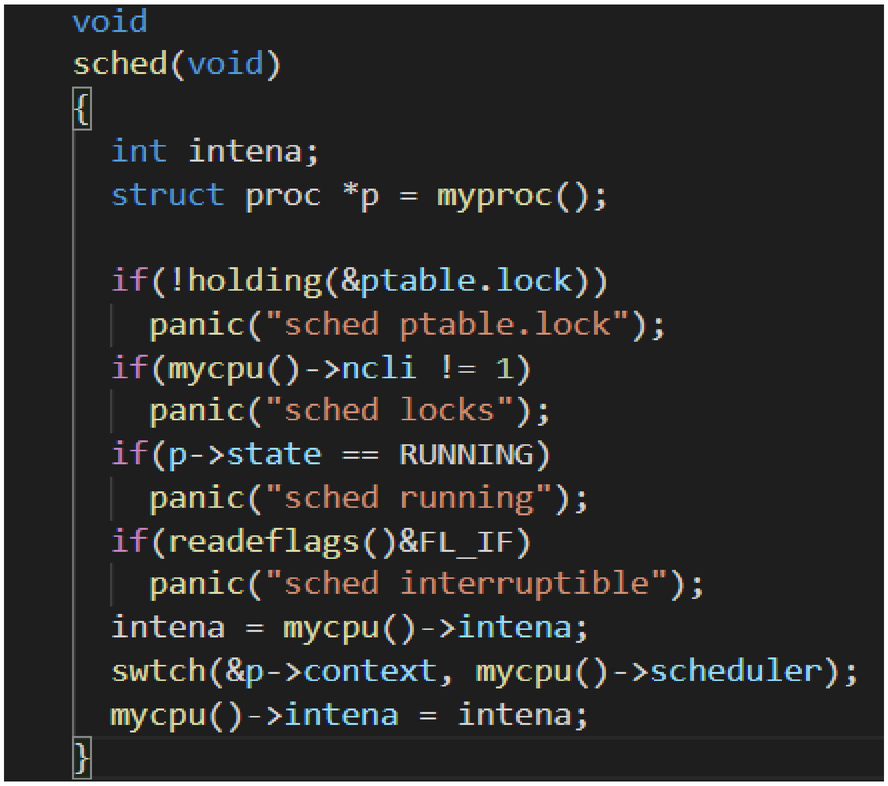
sys_yield calls the kernel-level function yield.

Function yield -



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

This function sets the state of the running process to RUNNABLE and calls the sched function which is responsible for performing a context switch and executing the scheduler to bring the next process into the running state.



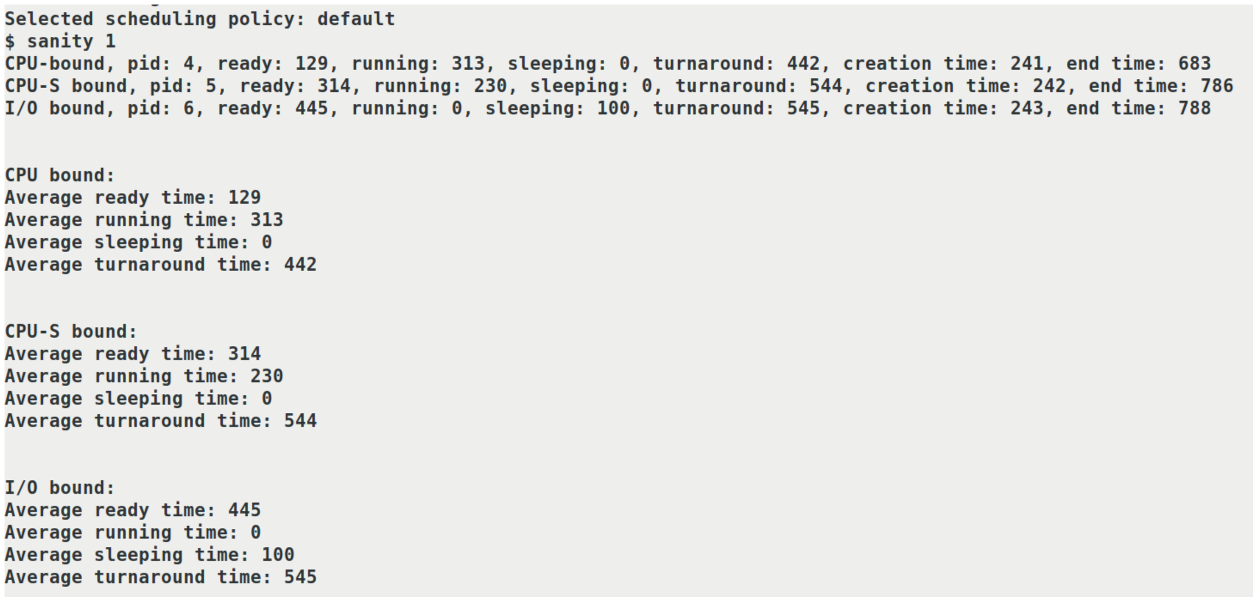
```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

Task 3

3.1 General Sanity Test :

Policies:-

(i) Default Scheduling Policy with QUANTA:

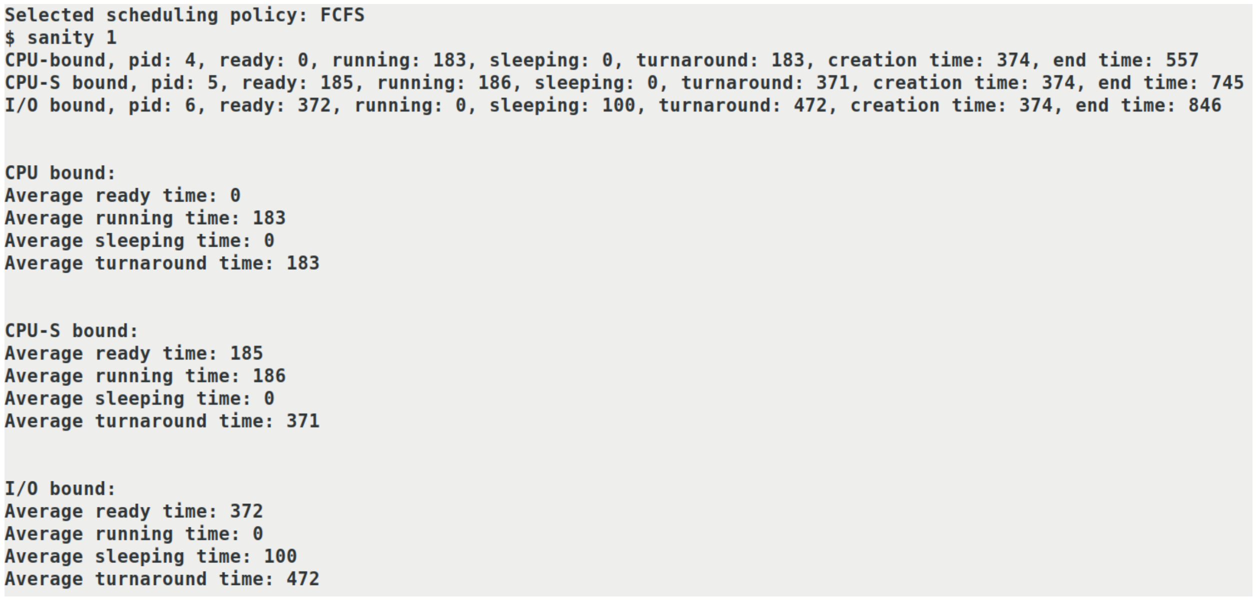


```
Selected scheduling policy: default
$ sanity 1
CPU-bound, pid: 4, ready: 129, running: 313, sleeping: 0, turnaround: 442, creation time: 241, end time: 683
CPU-S bound, pid: 5, ready: 314, running: 230, sleeping: 0, turnaround: 544, creation time: 242, end time: 786
I/O bound, pid: 6, ready: 445, running: 0, sleeping: 100, turnaround: 545, creation time: 243, end time: 788


CPU bound:
Average ready time: 129
Average running time: 313
Average sleeping time: 0
Average turnaround time: 442


CPU-S bound:
Average ready time: 314
Average running time: 230
Average sleeping time: 0
Average turnaround time: 544


I/O bound:
Average ready time: 445
Average running time: 0
Average sleeping time: 100
Average turnaround time: 545
```

We defined QUANTA as 5 in the file param.h; so, the above screenshot shows the default policy (round robin) and we context switch the processes after every 5 ticks. (Without QUANTA the policy was context switching after every clock tick).

So, when we run "sanity 1", we are creating 3 processes, with pids 4, 5, 6.
We did add some cprintf statements in order to understand the working of every scheduling policy in the function scheduler() in proc.c . Now, we have commented those cprintf lines for the output to be clean since they were added only for understanding / debugging purposes. From these lines, we understood the following:
Firstly the process with pid 4 enters the running state and runs for 5 ticks. Then it yields (after QUANTA) and the process with pid 5 enters. This process is observed to run for 3 ticks (less than 5) since in sanity.c, as per the assignment instructions we included yield() system call inside the for loop itself. So, the scheduler picks the I/O bound process with the pid 6 and just executes for 1 tick since it involves the sleep() instructions and goes to the waiting state. This goes on until the processes finish and we obtain the statistics as shown above in the screenshot.

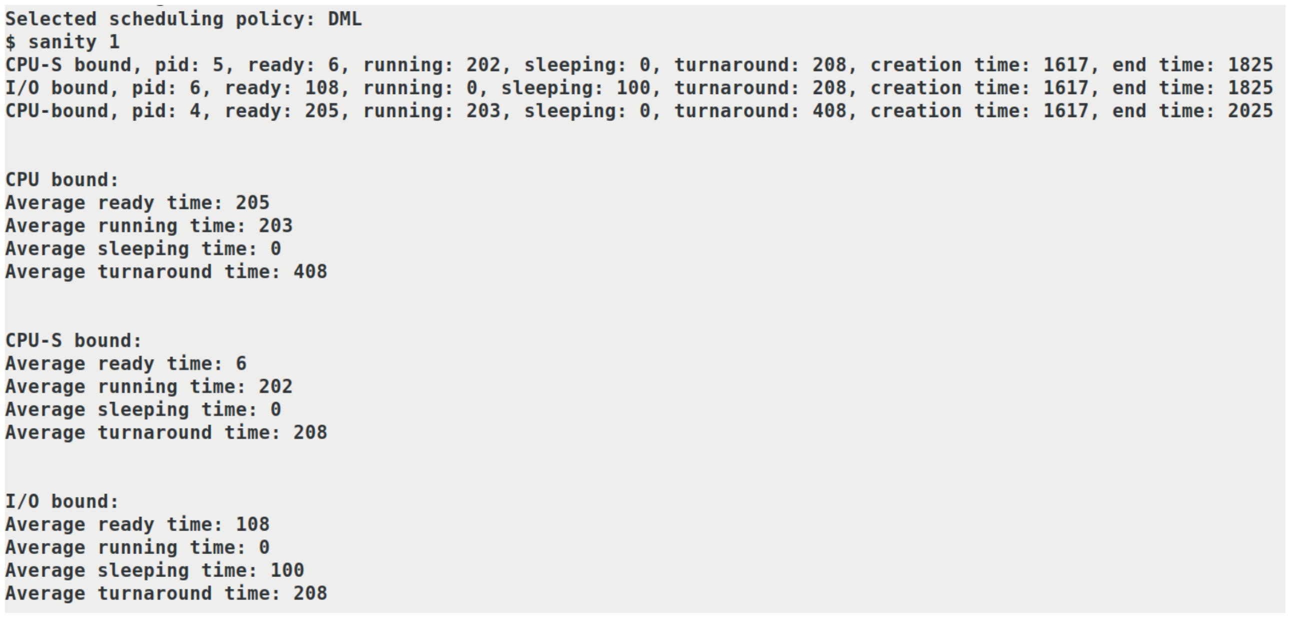
(ii) FCFS -



```
Selected scheduling policy: FCFS
$ sanity 1
CPU-bound, pid: 4, ready: 0, running: 183, sleeping: 0, turnaround: 183, creation time: 374, end time: 557
CPU-S bound, pid: 5, ready: 185, running: 186, sleeping: 0, turnaround: 371, creation time: 374, end time: 745
I/O bound, pid: 6, ready: 372, running: 0, sleeping: 100, turnaround: 472, creation time: 374, end time: 846


CPU bound:
Average ready time: 0
Average running time: 183
Average sleeping time: 0
Average turnaround time: 183


CPU-S bound:
Average ready time: 185
Average running time: 186
Average sleeping time: 0
Average turnaround time: 371


I/O bound:
Average ready time: 372
Average running time: 0
Average sleeping time: 100
Average turnaround time: 472
```

We create 3 processes i.e. 1 CPU-bound, 1 CPU-S bound and 1 I/O bound.
The CPU-bound process is not preempted before completion and is the first one to start execution thus it finishes execution the earliest. After which CPU-S bound process proceeds with execution with context switches in between due to yield() call every 10^6 iterations which would result in the same process being scheduled again, once this process finishes I/O bound process begins execution which goes into sleep after every iteration but no other process remains in the ready queue so the processor remains free during that duration after which I/O bound process is scheduled on the processor again and this goes on till it finishes execution. As we can see from the end time the three processes finish execution in the same order in which they are created by the sanity.c program.

(iii)DML-

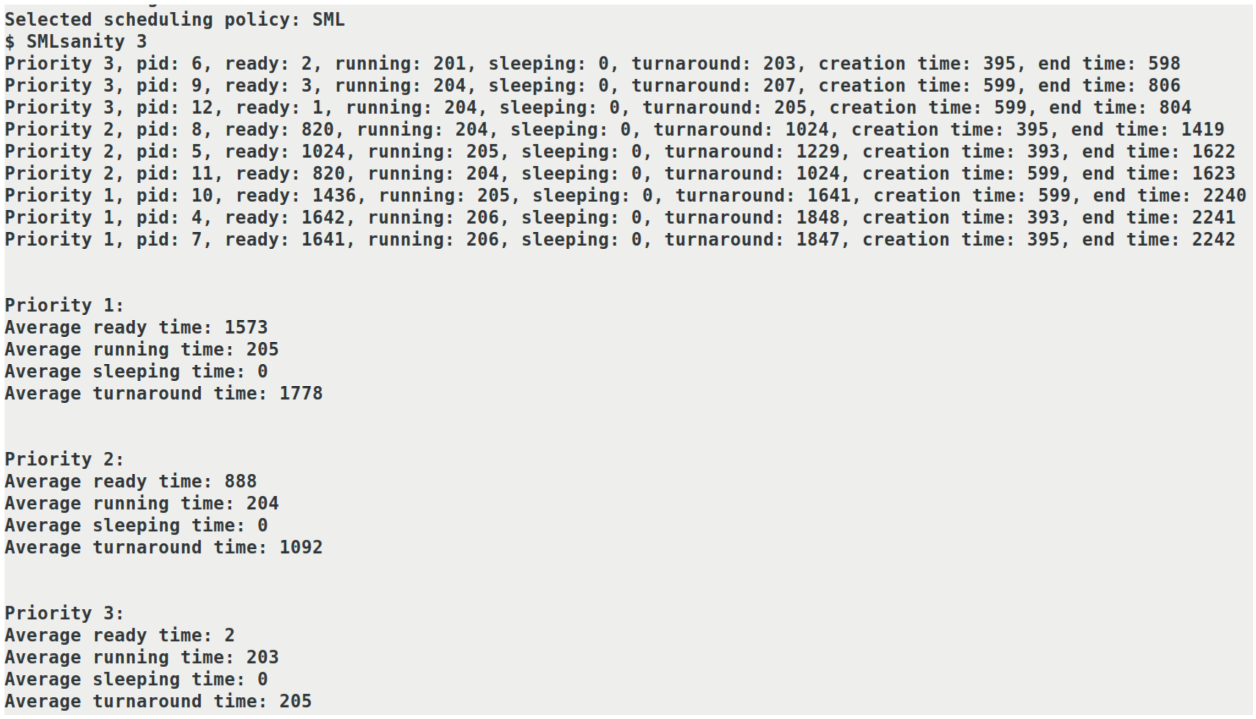


```
Selected scheduling policy: DML
$ sanity 1
CPU-S bound, pid: 5, ready: 6, running: 202, sleeping: 0, turnaround: 208, creation time: 1617, end time: 1825
I/O bound, pid: 6, ready: 108, running: 0, sleeping: 100, turnaround: 208, creation time: 1617, end time: 1825
CPU-bound, pid: 4, ready: 205, running: 203, sleeping: 0, turnaround: 408, creation time: 1617, end time: 2025


CPU bound:
Average ready time: 205
Average running time: 203
Average sleeping time: 0
Average turnaround time: 408


CPU-S bound:
Average ready time: 6
Average running time: 202
Average sleeping time: 0
Average turnaround time: 208


I/O bound:
Average ready time: 108
Average running time: 0
Average sleeping time: 100
Average turnaround time: 208
```

Here also we create 3 processes i.e. 1 CPU-bound, 1 CPU-S bound and 1 I/O Bound. In this case the default priority of each of the processes is 2 and decreases by 1 in the case that the process runs for 5 consecutive clock cycles (1 Quanta) and is set to 3 (highest priority) after the process wakes up from the sleeping state. In this case we observe that the end time of all the processes are very close. This happens mainly because at the start all of the processes have the same priority and the one that is encountered first while iteration in the PCB looking for runnable processes is executed after which whenever a process has run for 5 consecutive clock cycles, it's preempted and it doesn't get scheduled in the CPU in the next turn itself. Also the I/O bound process sleeps after every iteration and CPU-S bound process has to yield after every 10^6 iterations. So basically no process is able to hold the CPU for a long duration continuously due to these factors and hence the observed close proximity of end times was expected.

3.2 Priority schedule Test :

SML -



```
Selected scheduling policy: SML
$ SMLsanity 3
Priority 3, pid: 6, ready: 2, running: 201, sleeping: 0, turnaround: 203, creation time: 395, end time: 598
Priority 3, pid: 9, ready: 3, running: 204, sleeping: 0, turnaround: 207, creation time: 599, end time: 806
Priority 3, pid: 12, ready: 1, running: 204, sleeping: 0, turnaround: 205, creation time: 599, end time: 804
Priority 2, pid: 8, ready: 820, running: 204, sleeping: 0, turnaround: 1024, creation time: 395, end time: 1419
Priority 2, pid: 5, ready: 1024, running: 205, sleeping: 0, turnaround: 1229, creation time: 393, end time: 1622
Priority 2, pid: 11, ready: 820, running: 204, sleeping: 0, turnaround: 1024, creation time: 599, end time: 1623
Priority 1, pid: 10, ready: 1436, running: 205, sleeping: 0, turnaround: 1641, creation time: 599, end time: 2240
Priority 1, pid: 4, ready: 1642, running: 206, sleeping: 0, turnaround: 1848, creation time: 393, end time: 2241
Priority 1, pid: 7, ready: 1641, running: 206, sleeping: 0, turnaround: 1847, creation time: 395, end time: 2242


Priority 1:
Average ready time: 1573
Average running time: 205
Average sleeping time: 0
Average turnaround time: 1778


Priority 2:
Average ready time: 888
Average running time: 204
Average sleeping time: 0
Average turnaround time: 1092


Priority 3:
Average ready time: 2
Average running time: 203
Average sleeping time: 0
Average turnaround time: 205
```

Here we created a total of 9 processes, 3 each for priority 3, 2 and 1. In this case, all the processes are CPU bound processes without any sleep() invocations. So sleeping time would be 0 for all the processes.

Now for this policy, processes with higher priority can preempt the ones with lower priority. So processes with priority 3 should complete first, so it would have the lowest turnaround time in comparison to the other two. Then with the same argument we can say that processes with priority 2 would have lesser turnaround time than those with priority 1.

For ready time, as processes with priority 3 would only wait if another process of same priority is running, else it would just preempt out the other process. So processes with priority 3 would have the least ready time, then those with priority 2, then 1.

As we can see from the statistics from the screenshot, the average ready time and turnaround time is lowest for processes with priority 3, then priority 2 and then priority 1.
