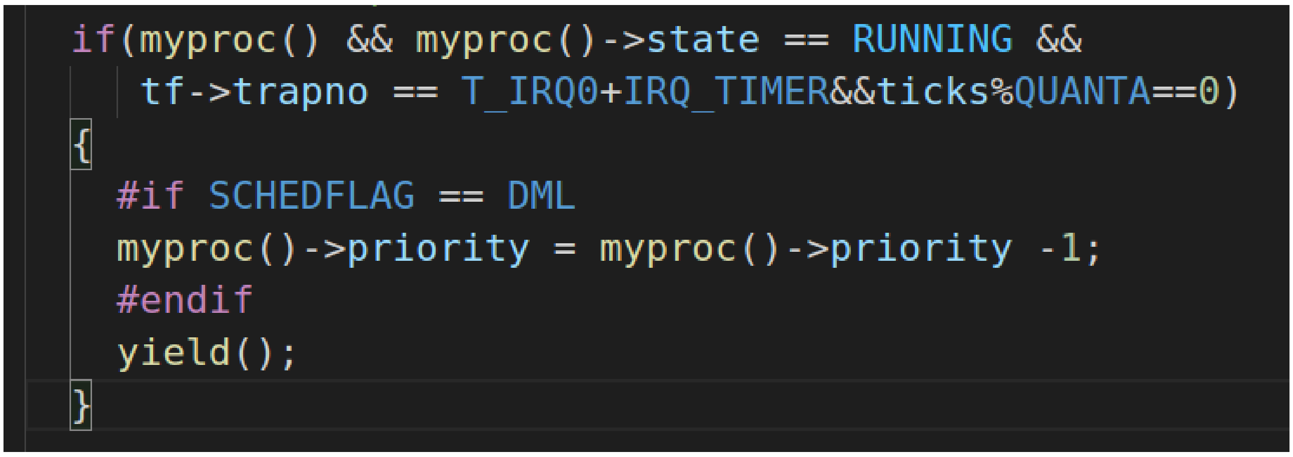
1.

a) For changing the timer interrupt frequency we have to chage the frequency of calling the yield(def in proc.c) function from trap(innterrupt handler in trap.c) so we added the condition ticks%QUANTA == 0 in the if block and we can see that the condition is satisfied once for every QUANTA . so, we call yield once in a QUANTA. QUANTA is defined in pram.h file.

Below is the code

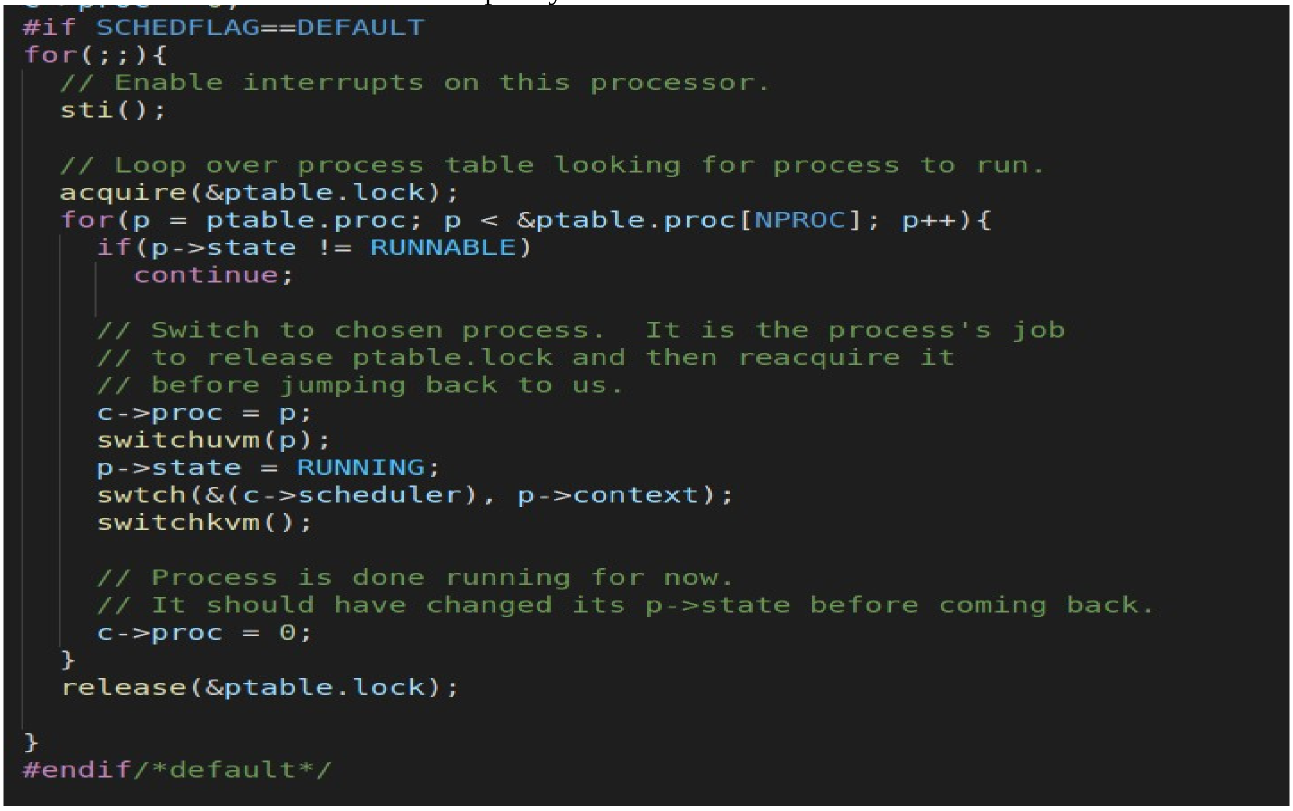


```
if(myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0+IRQ_TIMER&&ticks%QUANTA==0)
{
  #if SCHEDFLAG == DML
  myproc()->priority = myproc()->priority -1;
  #endif
  yield();
}
```

Policy 1: Default policy

In the above(a) we changed the frequency of invoking yield function so, now the scheduling takes place every QUANTA.

Below is the code for Default policy:

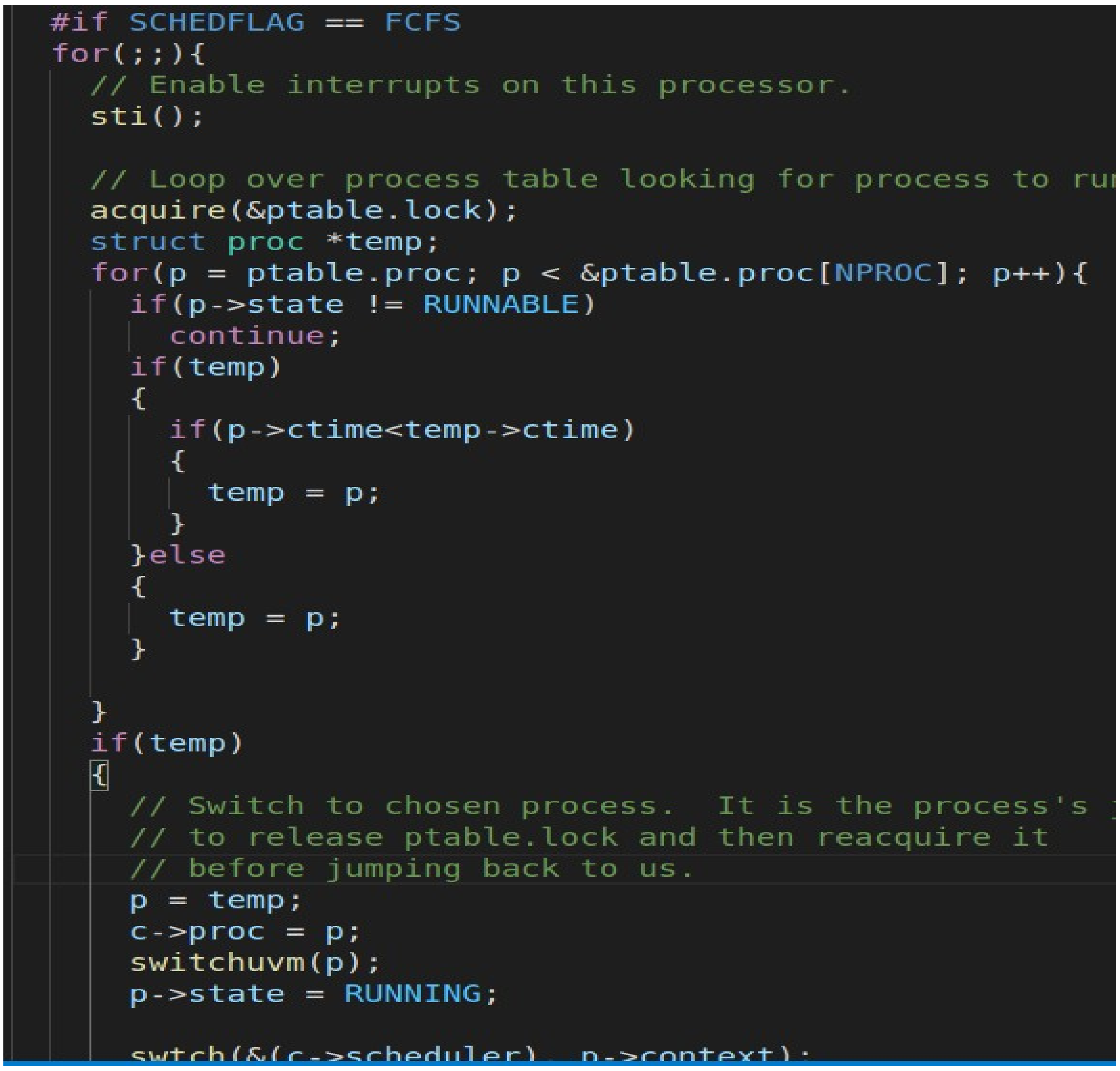


```
#if SCHEDFLAG==DEFAULT
for(;;){
  // Enable interrupts on this processor.
  sti();

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);
  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;
  }
  release(&ptable.lock);

}
#endif/*default*/
```

Policy 2: First come – First Served

In the FCFS we have to make sure that the process is not preempted for the timers interrupt and it will made to the waiting/sleeping state in case of a I/O and the scheduler is invoked scheduling will be based on the creation time of the process(ctime in struct proc represents this).

Below is the code for FCFS :



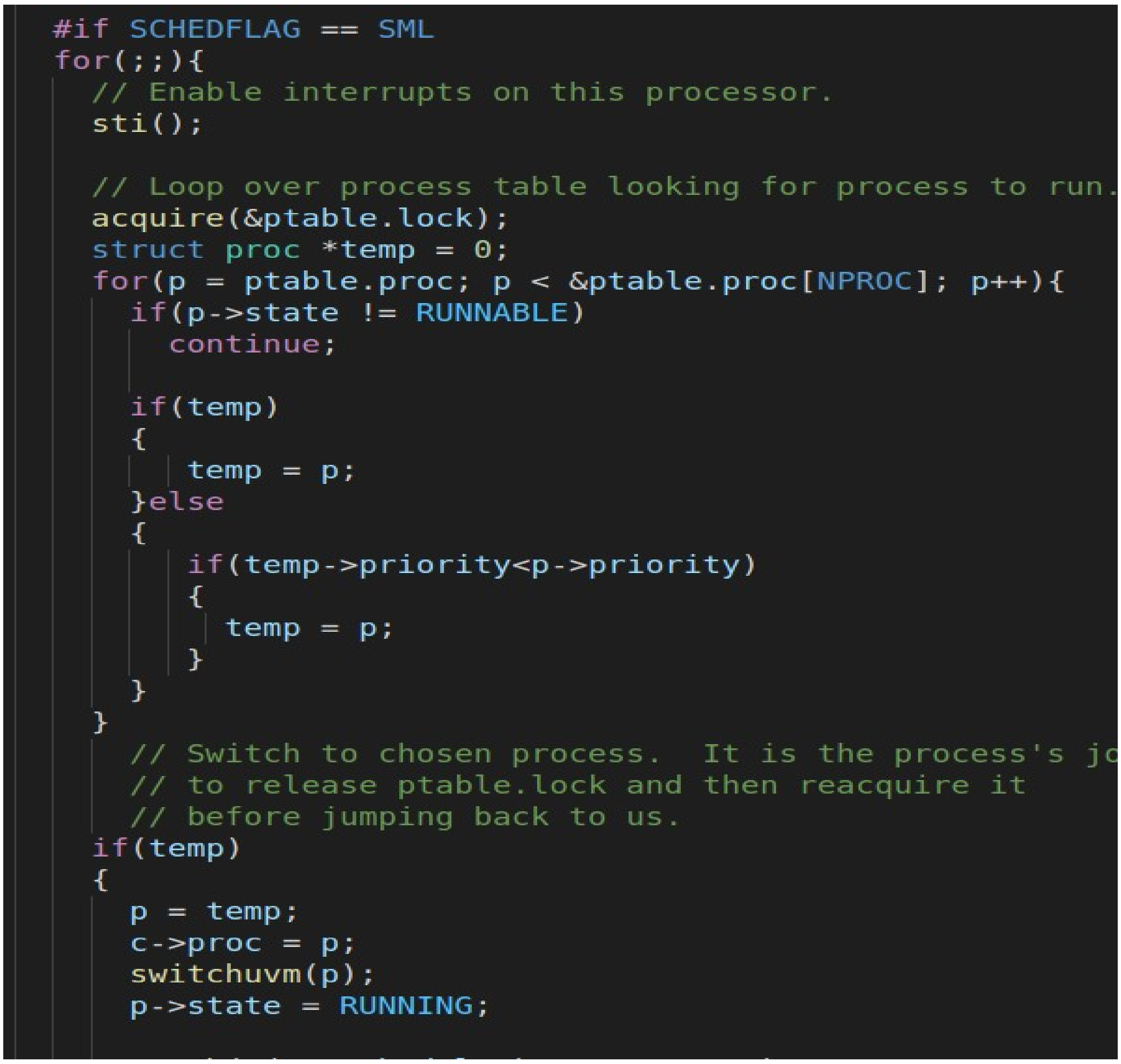
```
#if SCHEDFLAG == FCFS
for(;;){
  // Enable interrupts on this processor.
  sti();

  // Loop over process table looking for process to ru
  acquire(&ptable.lock);
  struct proc *temp;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;
    if(temp)
    {
      if(p->ctime<temp->ctime)
      {
        temp = p;
      }
    }else
    {
      temp = p;
    }

  }
  if(temp)
  {
    // Switch to chosen process.  It is the process's
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
    p = temp;
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

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

-> In case of a timer interrupt(per QUANTA) first trap() in trap.c will be called which inturn calls the yield() which changes the state of the current proces to RUNNABLE. Now when the scheduler runs we can see that the process ran before the interrupt had the lowest ctime(we find it using temp) so it will selected again .

-> In case of a IO the state of the process will be changed to SLEEPING and when the scheduler runs a new process will be selected because the if block at the start of the nested for loop makes sures that only process in RUNNABLE state will be selected.

Policy 3: Multi-level queue scheduling

In Multi-level queue scheduling, the ready is splited into some no of queues and there exists a priority between the queues. Based on the process the queue is assigned and here in this task the promotion and demotion of a process is done by the set_prio system call (we use this sys call for task 3.2).In this task the intial process is given a priority 2 and the priority will be copied upon fork. Which means that for this task all processes have priority 2.
we have to make sure that a process from a lower queue is selected only if no process is ready to run at a higher queue. We are taking that each queue uses DEFAULT scheduling of xv6.
set_prio is discussed in TASK – 3.2
Below is the code for Multi-level queue scheduling :



```
#if SCHEDFLAG == SML
for(;;){
  // Enable interrupts on this processor.
  sti();

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);
  struct proc *temp = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;

    if(temp)
    {
      temp = p;
    }else
    {
      if(temp->priority<p->priority)
      {
        temp = p;
      }
    }
  }
    // Switch to chosen process.  It is the process's jo
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
  if(temp)
  {
    p = temp;
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
```

-> when a process of priority i is selected it implies that there are no process in the queues of higher priorites this taken care by using temp in above implementation.

->A process is preempted when the time quanta is expired or it has to wait for a IO event then the scheduler will select a process from the highest priority which has non- terminated process.

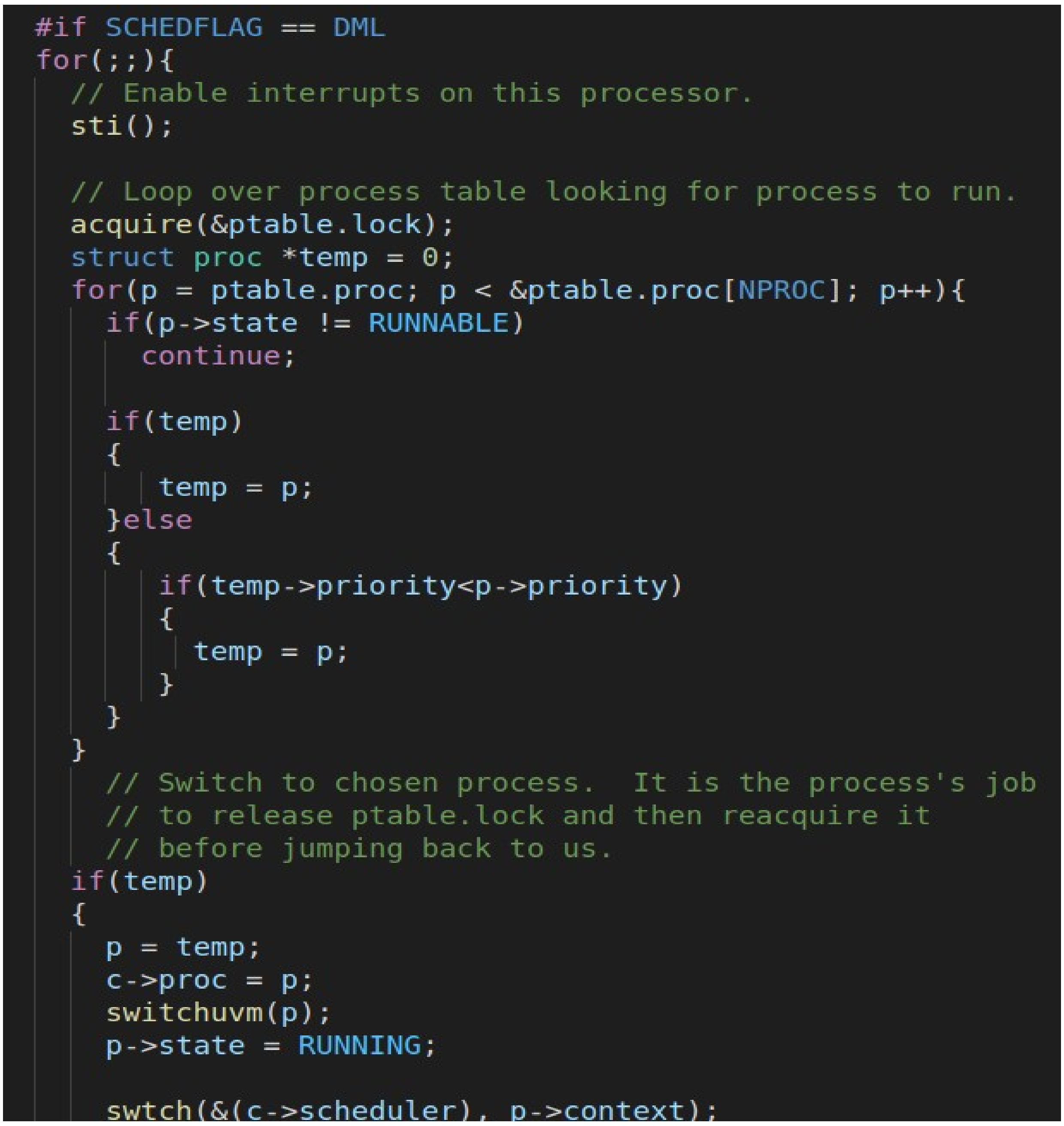
Policy 4: Dyanamic Multi-level queue scheduling

The only difference from the above policy is that there are some rules for changing the priority of the process and we implemented these rules by

i) RULE – 1 : In the exec.c we have added the line currproc->priority = 2 in the function exec().
ii)RULE – 2 : In the proc.c we added the line p->priority = 3 in the wakeup1(which wakes the process from SLEEPING mode) function.

iii)RULE – 4 : In trap.c before calling the yield we decremented the priority of the current process(because it ran for a full time QUANTA).

Below is the implementation for Dynamic Multi-level queue scheduling :



```
#if SCHEDFLAG == DML
for(;;){
  // Enable interrupts on this processor.
  sti();

  // Loop over process table looking for process to run.
  acquire(&ptable.lock);
  struct proc *temp = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE)
      continue;

    if(temp)
    {
      temp = p;
    }else
    {
      if(temp->priority<p->priority)
      {
        temp = p;
      }
    }
  }
    // Switch to chosen process.  It is the process's job
    // to release ptable.lock and then reacquire it
    // before jumping back to us.
  if(temp)
  {
    p = temp;
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;

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

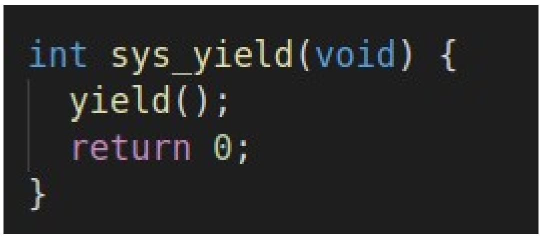
Note that every thing is in the conditional marcos.

Which process does the policy select for running? - the above screensots
What happens when a process returns from I/O? - depends upon the policy
What happens when a new process is created? - it will be added to the ready queue
When/how often does the scheduling take place? - per time quanta(if i have to say a number)

2.

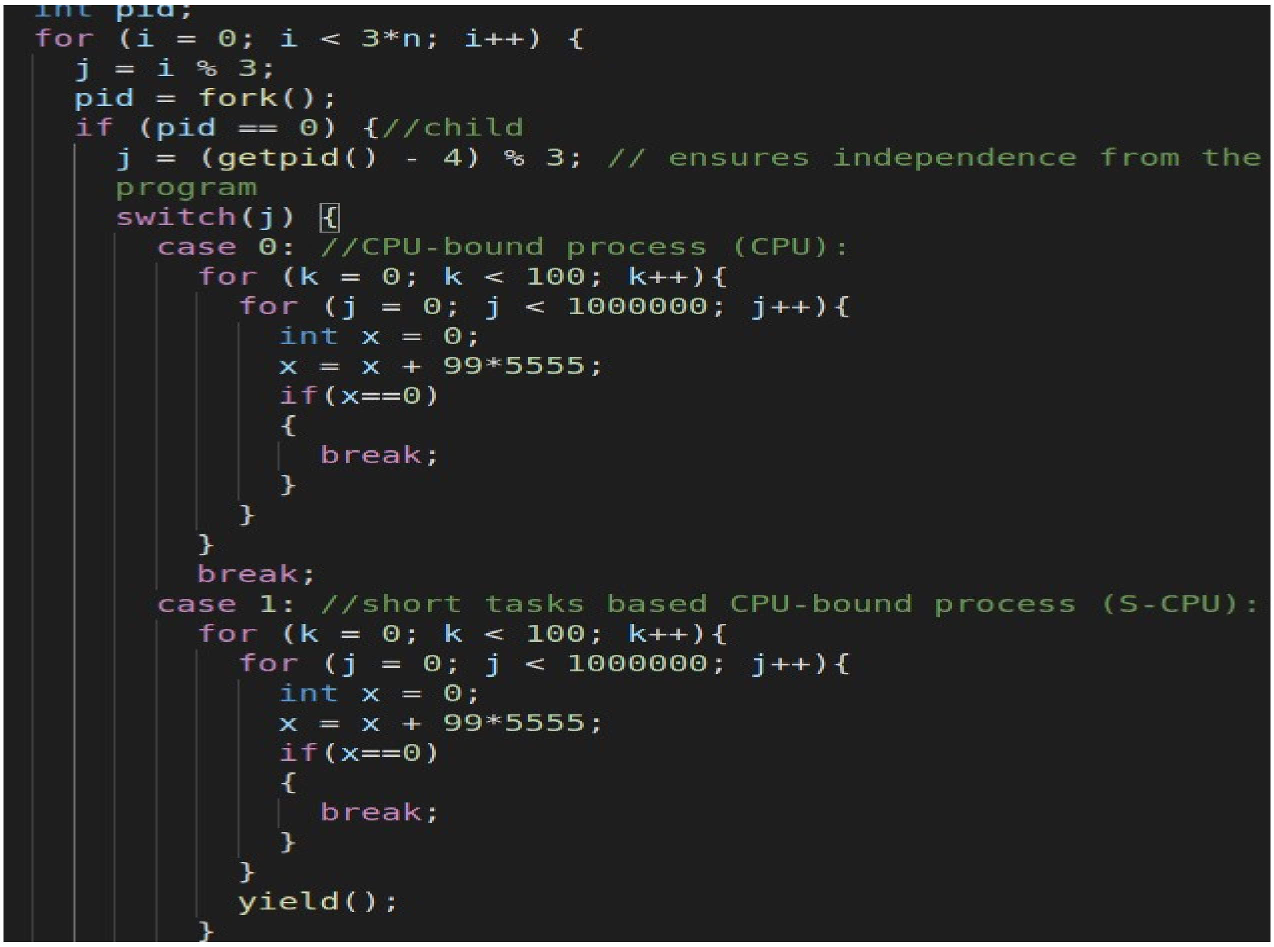
The yield system call is implemented using the yield function in the proc.c file. The yield function in the proc.c changes the state of the current process and calls the sched.
We changed the files for adding the system call like we have done in previous labs.

Below is code for sys_yield in sys_proc.c :



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

3.1
The testing code is in the file sanity.c . whose screenshot is attached below

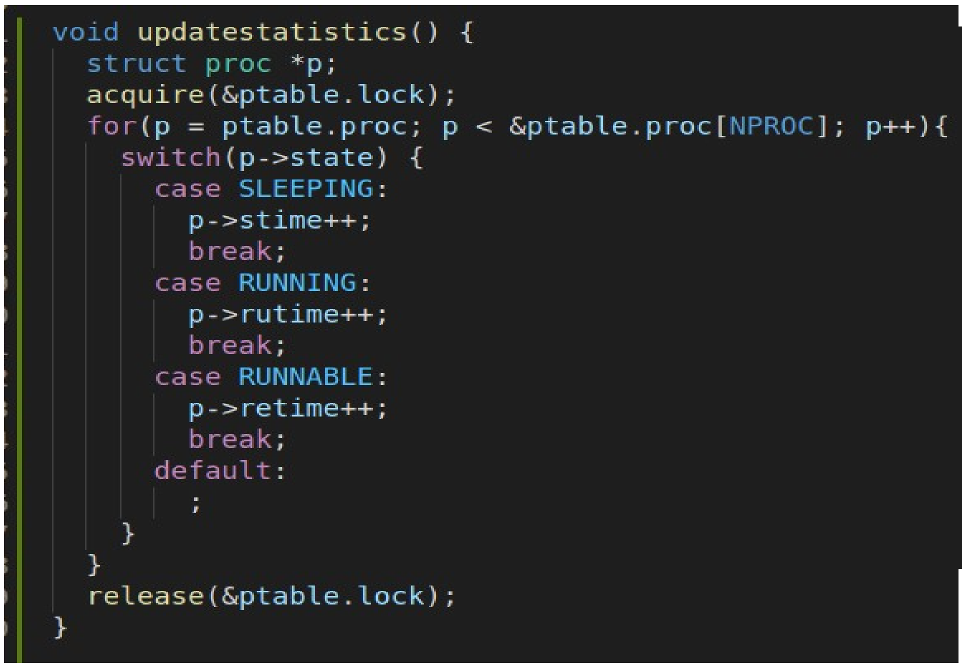


```
int pid;
for (i = 0; i < 3*n; i++) {
  j = i % 3;
  pid = fork();
  if (pid == 0) {//child
    j = (getpid() - 4) % 3; // ensures independence from the
    program
    switch(j) {
      case 0: //CPU-bound process (CPU):
        for (k = 0; k < 100; k++){
          for (j = 0; j < 1000000; j++){
            int x = 0;
            x = x + 99*5555;
            if(x==0)
            {
              break;
            }
          }
        }
        break;
      case 1: //short tasks based CPU-bound process (S-CPU):
        for (k = 0; k < 100; k++){
          for (j = 0; j < 1000000; j++){
            int x = 0;
            x = x + 99*5555;
            if(x==0)
            {
              break;
            }
          }
          yield();
        }
```

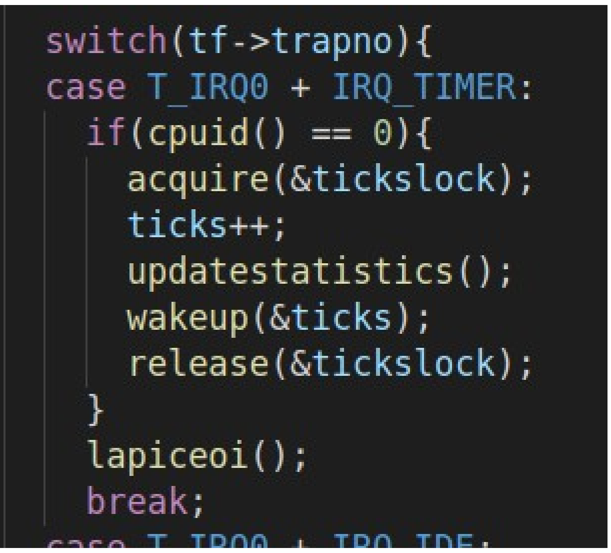
Some computations are added in the CPU – bound and Short CPU – bound process for non –   zero running time.

Calculations of ctime, rtime, rutime, stime :
-> ctime is set at the time of process creation i.e, ctime = ticks
-> we wrote a function(updatestatistics) which will update the rtime,rutime,stime for all the processes based on their states and we invoke this function for every tick/timer interrupt(it is in trap).so, in this way we calculate the rtime,rutime,stime.

Below is the code for updatestatistis and change in trap.c(right) :

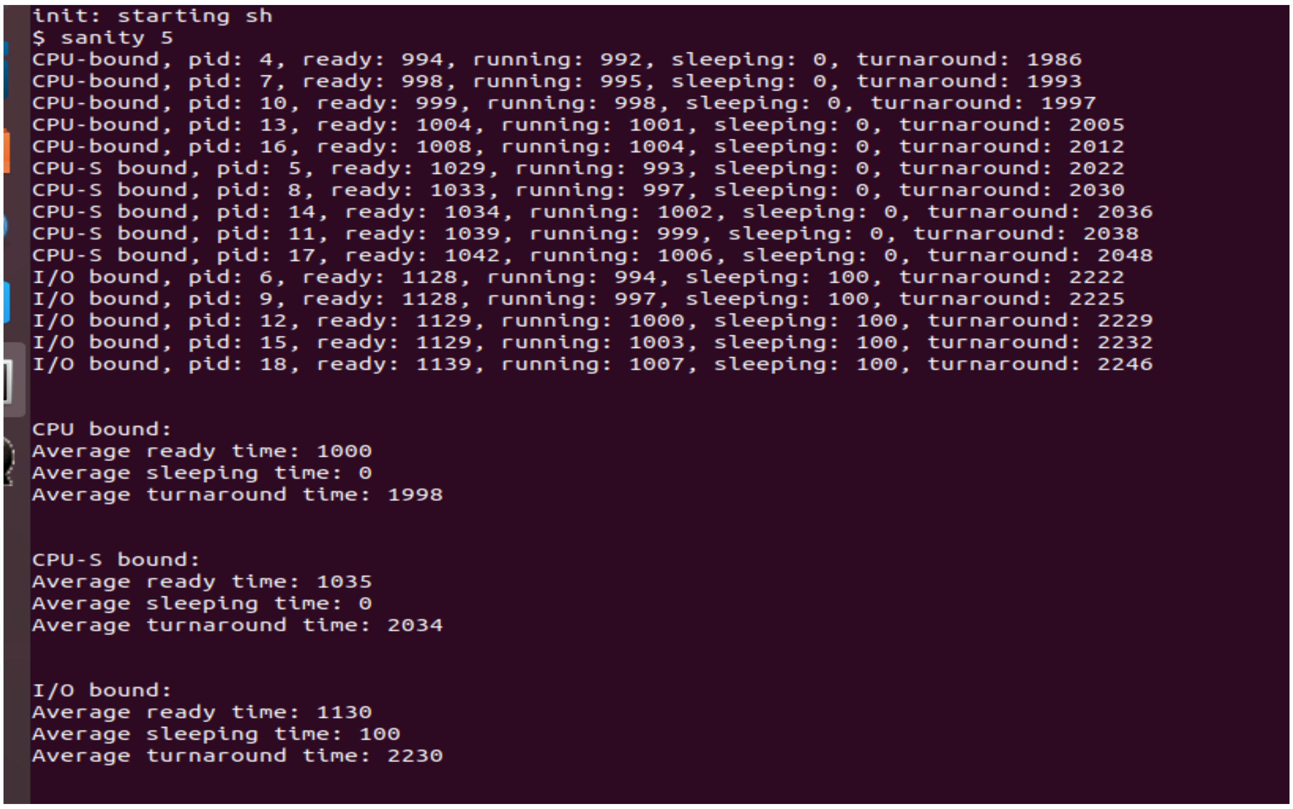


```
void updatestatistics() {
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    switch(p->state) {
      case SLEEPING:
        p->stime++;
        break;
      case RUNNING:
        p->rutime++;
        break;
      case RUNNABLE:
        p->retime++;
        break;
      default:
        ;
    }
  }
  release(&ptable.lock);
}
```



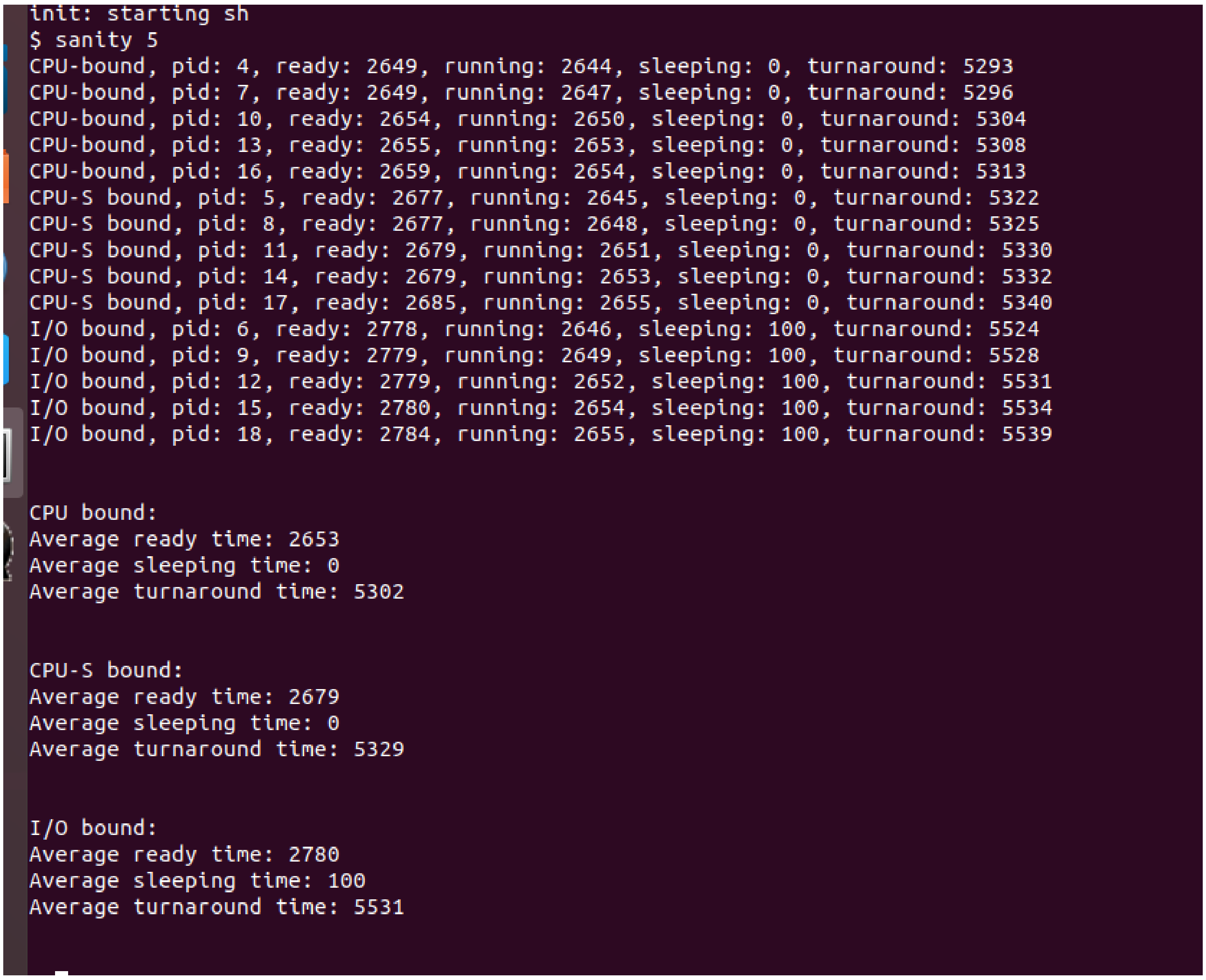
```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpuid() == 0){
    acquire(&tickslock);
    ticks++;
    updatestatistics();
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

Stats for each policy is shown below :
i)DEFAULT :



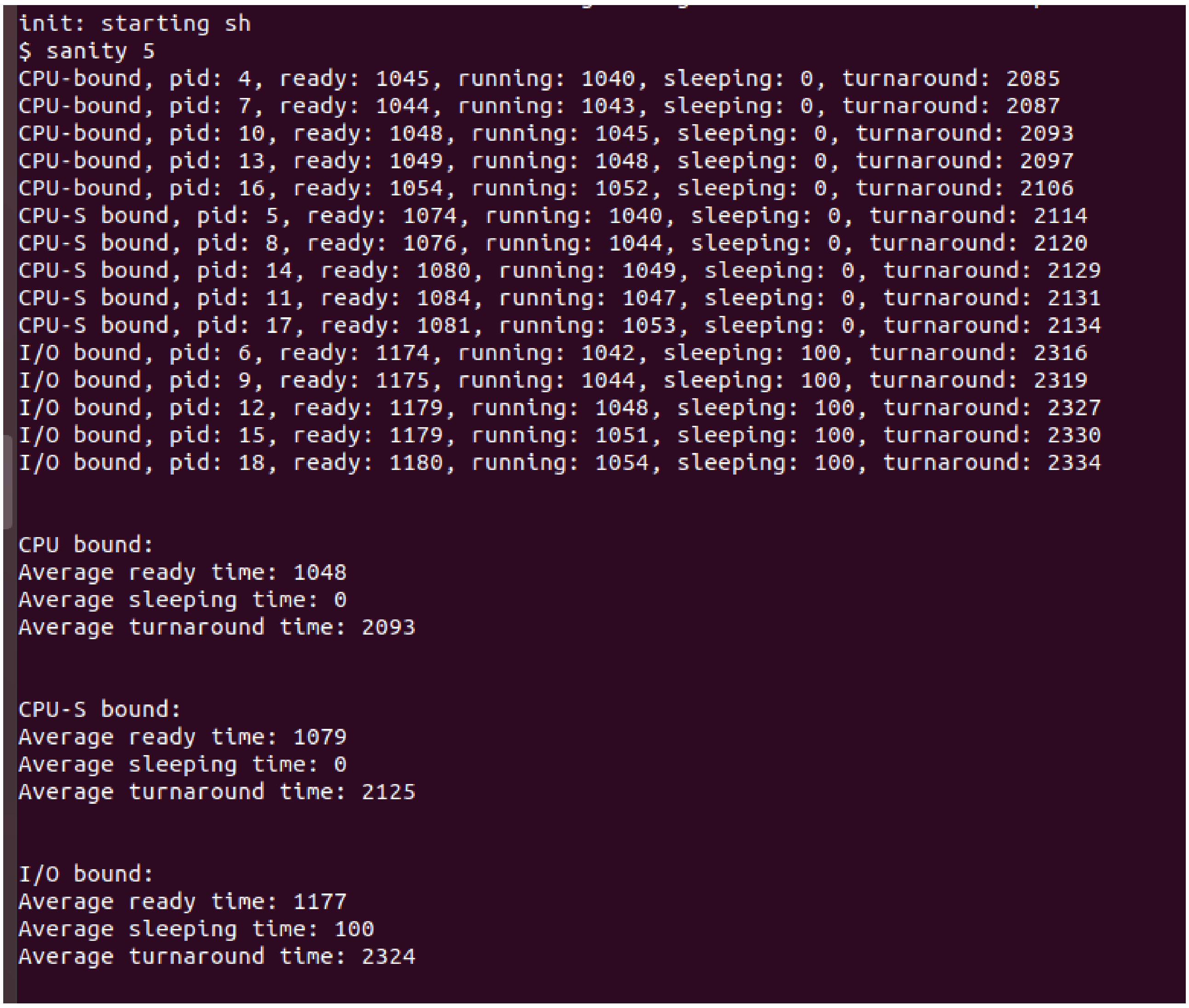
```
init: starting sh
$ sanity 5
CPU-bound, pid: 4, ready: 994, running: 992, sleeping: 0, turnaround: 1986
CPU-bound, pid: 7, ready: 998, running: 995, sleeping: 0, turnaround: 1993
CPU-bound, pid: 10, ready: 999, running: 998, sleeping: 0, turnaround: 1997
CPU-bound, pid: 13, ready: 1004, running: 1001, sleeping: 0, turnaround: 2005
CPU-bound, pid: 16, ready: 1008, running: 1004, sleeping: 0, turnaround: 2012
CPU-S bound, pid: 5, ready: 1029, running: 993, sleeping: 0, turnaround: 2022
CPU-S bound, pid: 8, ready: 1033, running: 997, sleeping: 0, turnaround: 2030
CPU-S bound, pid: 14, ready: 1034, running: 1002, sleeping: 0, turnaround: 2036
CPU-S bound, pid: 11, ready: 1039, running: 999, sleeping: 0, turnaround: 2038
CPU-S bound, pid: 17, ready: 1042, running: 1006, sleeping: 0, turnaround: 2048
I/O bound, pid: 6, ready: 1128, running: 994, sleeping: 100, turnaround: 2222
I/O bound, pid: 9, ready: 1128, running: 997, sleeping: 100, turnaround: 2225
I/O bound, pid: 12, ready: 1129, running: 1000, sleeping: 100, turnaround: 2229
I/O bound, pid: 15, ready: 1129, running: 1003, sleeping: 100, turnaround: 2232
I/O bound, pid: 18, ready: 1139, running: 1007, sleeping: 100, turnaround: 2246


CPU bound:
Average ready time: 1000
Average sleeping time: 0
Average turnaround time: 1998


CPU-S bound:
Average ready time: 1035
Average sleeping time: 0
Average turnaround time: 2034


I/O bound:
Average ready time: 1130
Average sleeping time: 100
Average turnaround time: 2230
```

ii)FCFS :



```
init: starting sh
$ sanity 5
CPU-bound, pid: 4, ready: 2649, running: 2644, sleeping: 0, turnaround: 5293
CPU-bound, pid: 7, ready: 2649, running: 2647, sleeping: 0, turnaround: 5296
CPU-bound, pid: 10, ready: 2654, running: 2650, sleeping: 0, turnaround: 5304
CPU-bound, pid: 13, ready: 2655, running: 2653, sleeping: 0, turnaround: 5308
CPU-bound, pid: 16, ready: 2659, running: 2654, sleeping: 0, turnaround: 5313
CPU-S bound, pid: 5, ready: 2677, running: 2645, sleeping: 0, turnaround: 5322
CPU-S bound, pid: 8, ready: 2677, running: 2648, sleeping: 0, turnaround: 5325
CPU-S bound, pid: 11, ready: 2679, running: 2651, sleeping: 0, turnaround: 5330
CPU-S bound, pid: 14, ready: 2679, running: 2653, sleeping: 0, turnaround: 5332
CPU-S bound, pid: 17, ready: 2685, running: 2655, sleeping: 0, turnaround: 5340
I/O bound, pid: 6, ready: 2778, running: 2646, sleeping: 100, turnaround: 5524
I/O bound, pid: 9, ready: 2779, running: 2649, sleeping: 100, turnaround: 5528
I/O bound, pid: 12, ready: 2779, running: 2652, sleeping: 100, turnaround: 5531
I/O bound, pid: 15, ready: 2780, running: 2654, sleeping: 100, turnaround: 5534
I/O bound, pid: 18, ready: 2784, running: 2655, sleeping: 100, turnaround: 5539


CPU bound:
Average ready time: 2653
Average sleeping time: 0
Average turnaround time: 5302


CPU-S bound:
Average ready time: 2679
Average sleeping time: 0
Average turnaround time: 5329


I/O bound:
Average ready time: 2780
Average sleeping time: 100
Average turnaround time: 5531
```

iii) Dynamic Multi level queue scheduling :

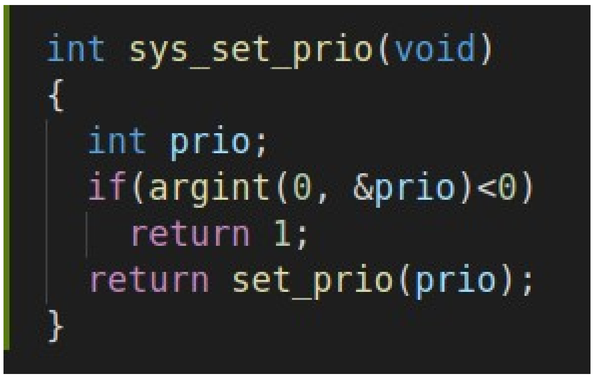


```
init: starting sh
$ sanity 5
CPU-bound, pid: 4, ready: 1045, running: 1040, sleeping: 0, turnaround: 2085
CPU-bound, pid: 7, ready: 1044, running: 1043, sleeping: 0, turnaround: 2087
CPU-bound, pid: 10, ready: 1048, running: 1045, sleeping: 0, turnaround: 2093
CPU-bound, pid: 13, ready: 1049, running: 1048, sleeping: 0, turnaround: 2097
CPU-bound, pid: 16, ready: 1054, running: 1052, sleeping: 0, turnaround: 2106
CPU-S bound, pid: 5, ready: 1074, running: 1040, sleeping: 0, turnaround: 2114
CPU-S bound, pid: 8, ready: 1076, running: 1044, sleeping: 0, turnaround: 2120
CPU-S bound, pid: 14, ready: 1080, running: 1049, sleeping: 0, turnaround: 2129
CPU-S bound, pid: 11, ready: 1084, running: 1047, sleeping: 0, turnaround: 2131
CPU-S bound, pid: 17, ready: 1081, running: 1053, sleeping: 0, turnaround: 2134
I/O bound, pid: 6, ready: 1174, running: 1042, sleeping: 100, turnaround: 2316
I/O bound, pid: 9, ready: 1175, running: 1044, sleeping: 100, turnaround: 2319
I/O bound, pid: 12, ready: 1179, running: 1048, sleeping: 100, turnaround: 2327
I/O bound, pid: 15, ready: 1179, running: 1051, sleeping: 100, turnaround: 2330
I/O bound, pid: 18, ready: 1180, running: 1054, sleeping: 100, turnaround: 2334


CPU bound:
Average ready time: 1048
Average sleeping time: 0
Average turnaround time: 2093


CPU-S bound:
Average ready time: 1079
Average sleeping time: 0
Average turnaround time: 2125


I/O bound:
Average ready time: 1177
Average sleeping time: 100
Average turnaround time: 2324
```

From the above screenshots we can see that the avg turnaround time and avg ready time is leass for DML when compared to the above three because in DML based on the process(CPU bound/IO bound) we changed the priority of the process which decreases the starvation.

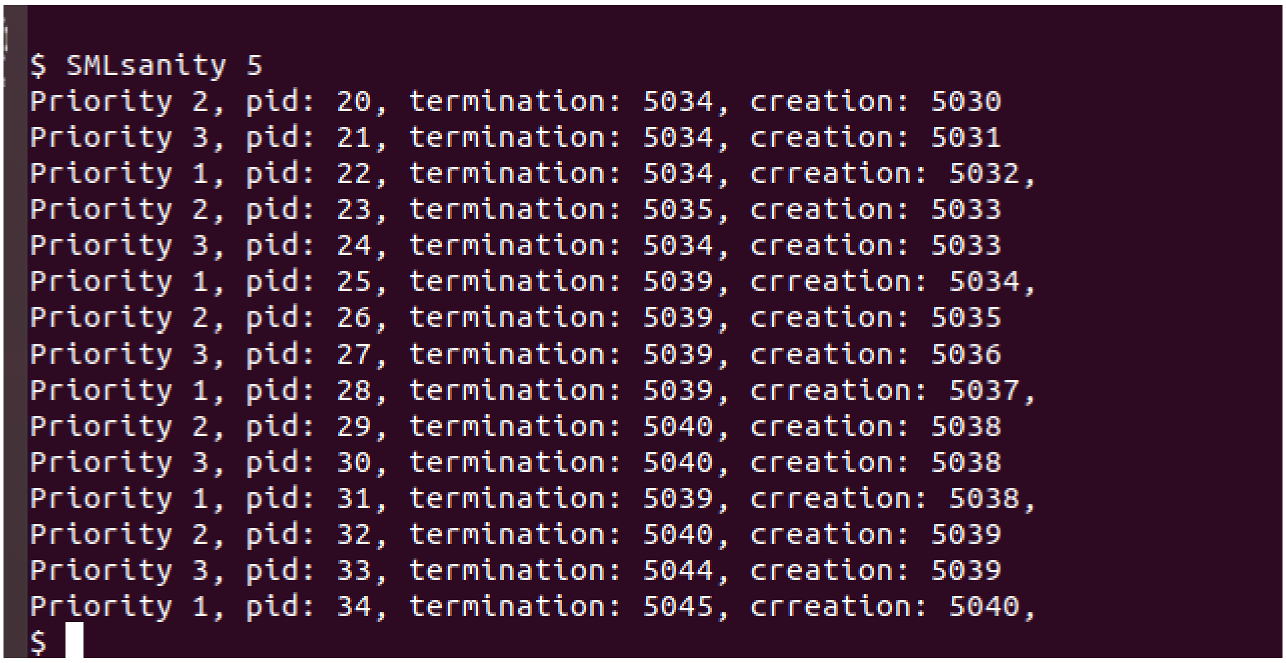
3.2

set_prio system call is made to promote or demote the process based on its behaviour.The implemantation of set_prio is shown below , it will use the set_prio function in proc.c file in the implemantaion



```c
int sys_set_prio(void)
{
  int prio;
  if(argint(0, &prio)<0)
    return 1;
  return set_prio(prio);
}
```

SMLsanity test for Multi level queue scheduling :



```
$ SMLsanity 5
Priority 2, pid: 20, termination: 5034, creation: 5030
Priority 3, pid: 21, termination: 5034, creation: 5031
Priority 1, pid: 22, termination: 5034, crreation: 5032,
Priority 2, pid: 23, termination: 5035, creation: 5033
Priority 3, pid: 24, termination: 5034, creation: 5033
Priority 1, pid: 25, termination: 5039, crreation: 5034,
Priority 2, pid: 26, termination: 5039, creation: 5035
Priority 3, pid: 27, termination: 5039, creation: 5036
Priority 1, pid: 28, termination: 5039, crreation: 5037,
Priority 2, pid: 29, termination: 5040, creation: 5038
Priority 3, pid: 30, termination: 5040, creation: 5038
Priority 1, pid: 31, termination: 5039, crreation: 5038,
Priority 2, pid: 32, termination: 5040, creation: 5039
Priority 3, pid: 33, termination: 5044, creation: 5039
Priority 1, pid: 34, termination: 5045, crreation: 5040,
$ 
```

from the above screenshot we can see that priority 3 process(second one) is created at 5031 after the priority 2 process(created at 5030) but they both terminated at the same time (5034) because we used SML scheduling.