CS-344 Assignment-2B

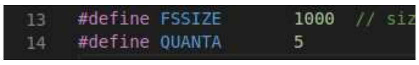
Group-9:

P Sai Ashrritth:190101061
Amancha Jagruth:190101010
Adduri Sri Sai Datta:190101003
Bhukya Bharath:190101024

Task-1:

After seeing the current scheduling policy,

● Which process does the policy select for running?
  Xv6 uses the Round Robin scheduler.
  selects the first process in the Process table(available to RUN).
● What happens when a process returns from I/O?
  Any process that is waiting for an I/O event its state is set to WAIT.
  When any process returns from I/O completion its state is set to RUN.
● What happens when a new process is created?
  Init process will create the first process after the bootup,it will fork shell process which will Fork other processes to run user commands.
  After every other process created by fork () function, which creates a new child process from the parent process and set it to RUN state.
  The allocproc() function is called during both init and fork, which allocates new process Data Structure and set up a new kernel stack for context switching.
● When/how often does the scheduling take place?
  Every clock cycle, scheduling takes place.



```
#define FSIZE       1000  // siz
#define QUANTA      5
```

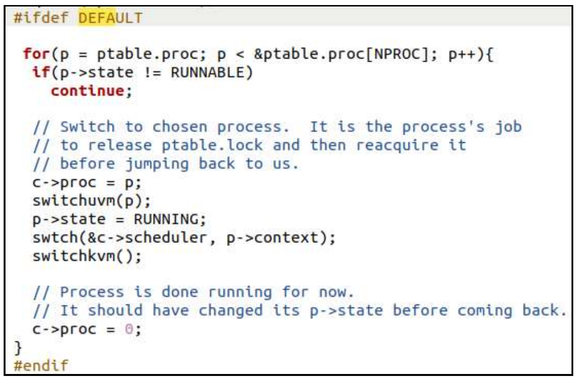
Firstly, we added a macro name QUANTA in param.h.

Now we implemented each scheduling policy using ifdef macros in proc.c

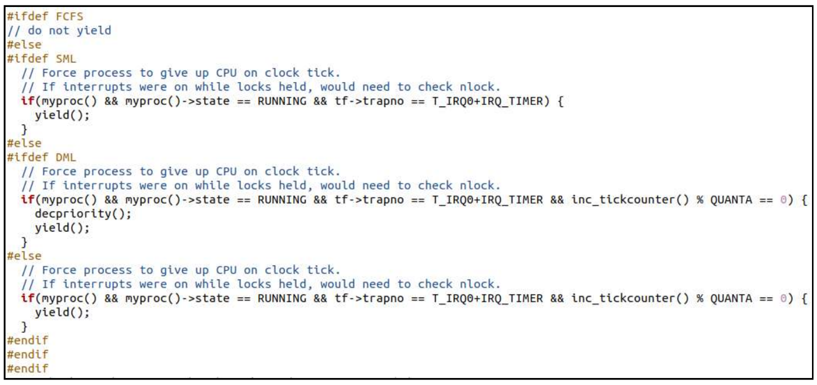
Adding a time quantum;

We have changed the current scheduling code such that process preemption will be done in every time quantum instead of every clock cycle.
In proc.c we inc_tickcounter return the tickcounter.

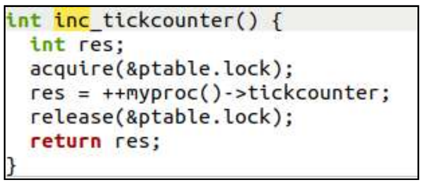
The basic logic we set this QUANTA is, we will call inc_tickcounter at every tick, this function actually increases the tickcounter parameter in the process struct and return that value. Now when this tickcounter is divisible by 5, we will yield the process. Then we will fetch the next process using the scheduler, So until 5 ticks, we are not yielding a process, so in this way, we implement round-robin scheduling.



```
#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;
}
#endif
```



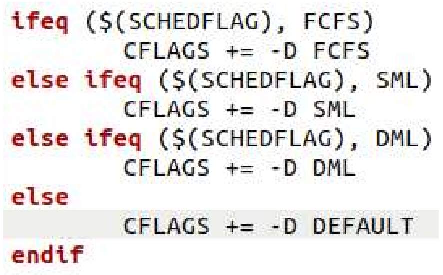
```
#ifdef FCFS
// do not yield
#else
#ifdef SML
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER) {
    yield();
  }
#else
#ifdef DML
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER && inc_tickcounter() % QUANTA == 0) {
    decpriority();
    yield();
  }
#else
  // Force process to give up CPU on clock tick.
  // If interrupts were on while locks held, would need to check nlock.
  if(myproc() && myproc()->state == RUNNING && tf->trapno == T_IRQ0+IRQ_TIMER && inc_tickcounter() % QUANTA == 0) {
    yield();
  }
#endif
#endif
#endif
```



```
int inc_tickcounter() {
  int res;
  acquire(&ptable.lock);
  res = ++myproc()->tickcounter;
  release(&ptable.lock);
  return res;
}
```

Here in the above image from trap.c, the last else statement refers to the DEFAULT mode.

Now, let’s modify makefile.

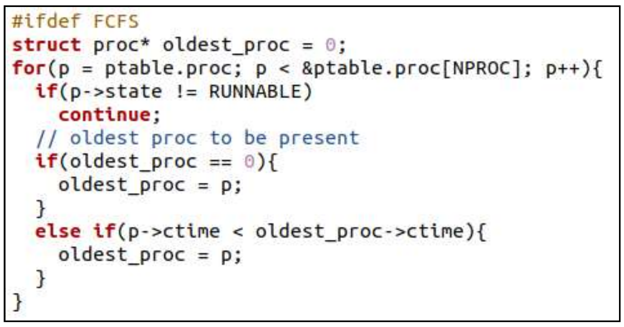


```
ifeq ($(SCHEDFLAG), FCFS)
        CFLAGS += -D FCFS
else ifeq ($(SCHEDFLAG), SML)
        CFLAGS += -D SML
else ifeq ($(SCHEDFLAG), DML)
        CFLAGS += -D DML
else
        CFLAGS += -D DEFAULT
endif
```

<u>FCFS:</u>

<u>FCFS function:</u>

1. In for loop first_proc variable finds the process which can be run and which has least
        creation_time in ptable.
2. If oldest_proc !=NULL then oldest_proc get cpu



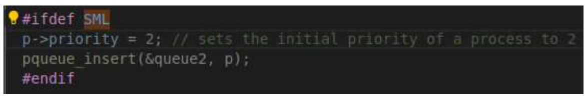
```
#ifdef FCFS
struct proc* oldest_proc = 0;
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;
  // oldest proc to be present
  if(oldest_proc == 0){
    oldest_proc = p;
  }
  else if(p->ctime < oldest_proc->ctime){
    oldest_proc = p;
  }
}
```

As shown before in trap.c, for FCFS, we do not yield a process in between.

<u>MULTILEVEL QUEUE SCHEDULING(SML):</u>

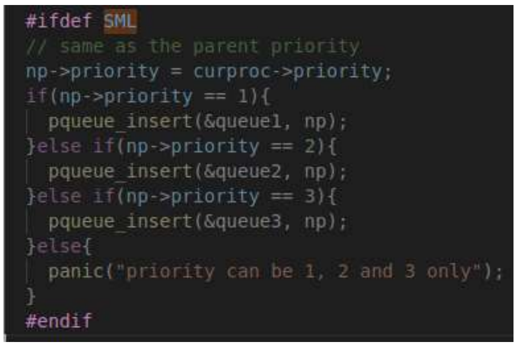
In this scheduling policy, we will first create a queue structure and implement its related functions. Then we declare 3 queues. Now for this, we will go through the queues in their priority order. If the higher priority queue has no process to run, only then the scheduler will move on to the next higher priority queue.

In userinit(), we set the initial priority of a process to 2. And insert it into the respective queue. (using pqueue_insert())



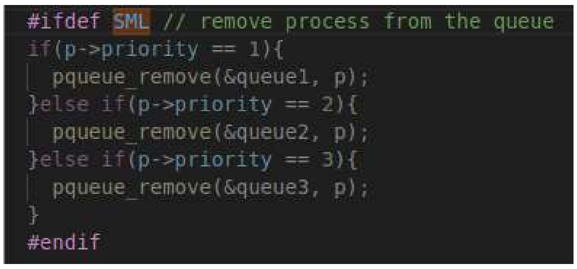
```
#ifdef SML
  p->priority = 2; // sets the initial priority of a process to 2
  pqueue_insert(&queue2, p);
  #endif
```

Now, when a process is created using fork(), we set the priority of the new process the same as the priority of the parent.



```
#ifdef SML
// same as the parent priority
np->priority = curproc->priority;
if(np->priority == 1){
  pqueue_insert(&queue1, np);
}else if(np->priority == 2){
  pqueue_insert(&queue2, np);
}else if(np->priority == 3){
  pqueue_insert(&queue3, np);
}else{
  panic("priority can be 1, 2 and 3 only");
}
#endif
```

Now, when a ZOMBIE process is being terminated using wait(), we should also remove that process from its respective priority queue. So we added the below part in wait().



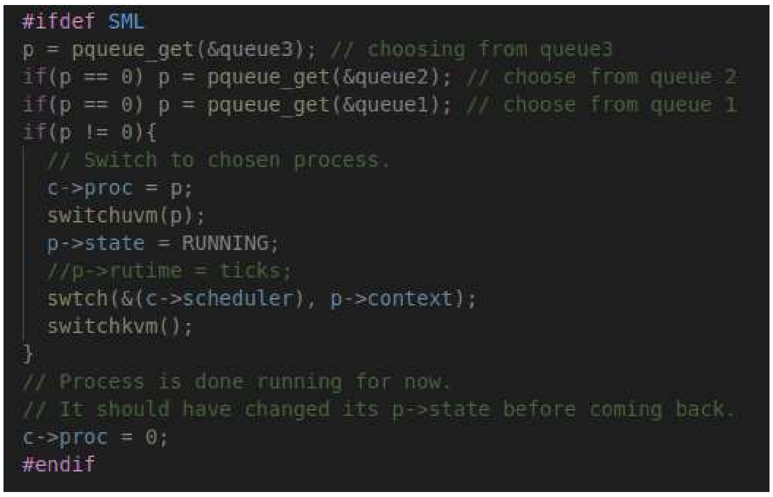
```
#ifdef SML // remove process from the queue
if(p->priority == 1){
  pqueue_remove(&queue1, p);
}else if(p->priority == 2){
  pqueue_remove(&queue2, p);
}else if(p->priority == 3){
  pqueue_remove(&queue3, p);
}
#endif
```

Now, her For SML scheduling, I assumed that QUANTA is not set for a process, it gets context switched at every cycle. I implemented this code in that way. Now if we want to set QUANTA in this process, we have to go to trap.c and this *inc_tickcounter() % QUANTA == 0* condition in the if statement in SML related code.

Now, in scheduler, we will select a process from queue3 i.e highest priority queue using pqueue_get() function.

Now if there are no processes left in the queue then the function will return a 0 value. In that case, we will go into the next queue i.e queue2, and approach similarly. We will do this every clock cycle.

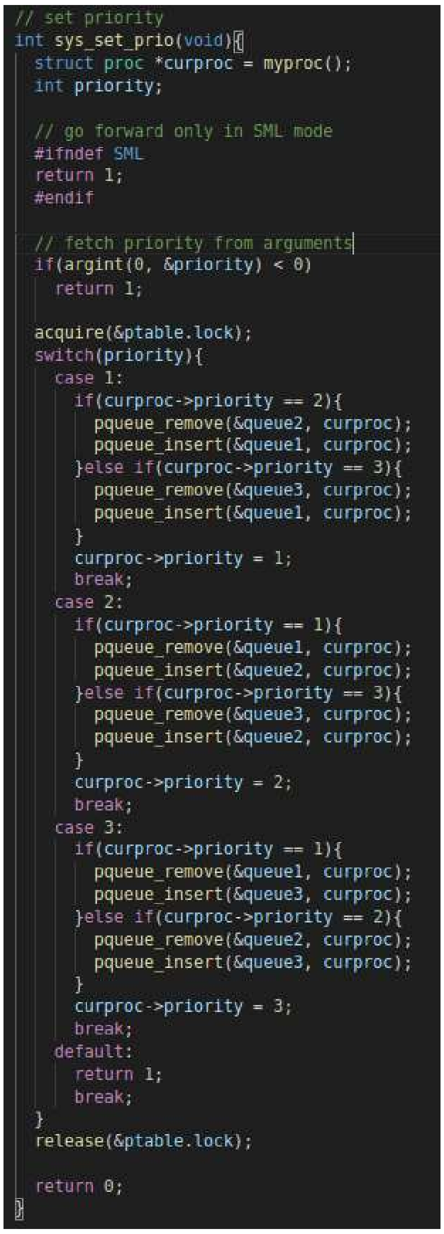
Then we will run the selected process and again select another process in the next cycle.



```
#ifdef SML
p = pqueue_get(&queue3); // choosing from queue3
if(p == 0) p = pqueue_get(&queue2); // choose from queue 2
if(p == 0) p = pqueue_get(&queue1); // choose from queue 1
if(p != 0){
  // Switch to chosen process.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
  //p->rutime = ticks;
  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;
#endif
```

Next, we will implement a system call that will change the priority of the calling process. So we will do all the steps we need to do to add a system call like defining a macro in syscall.h and declaring and making it available to the whole program and adding them to the vector space in syscall.c, then declaring a user-level call definition in usys.S and including it user.h. Now we will implement this function in proc.c.

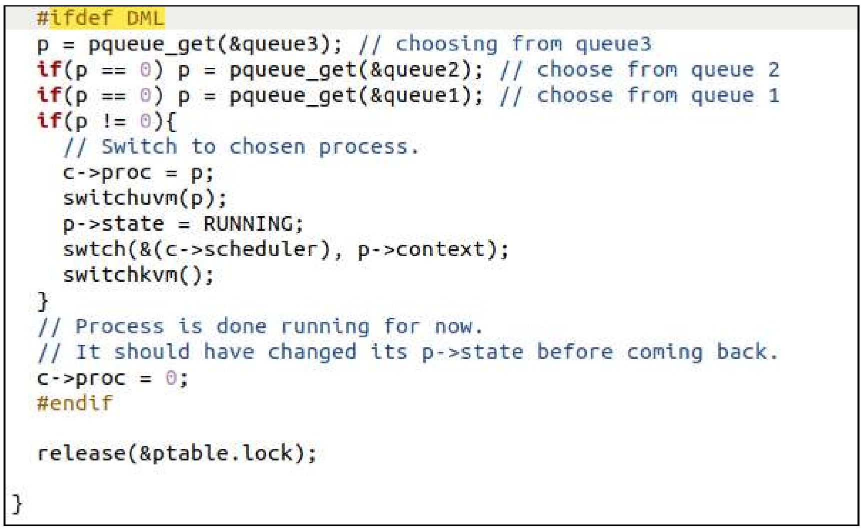
Now we will make this function only to work in SML mode by adding a ifndef statement. Now we will set the priority of the process by first moving into the new priority queue, then we will set the priority parameter of the process to the new priority value.



```
// set priority
int sys_set_prio(void){
  struct proc *curproc = myproc();
  int priority;

  // go forward only in SML mode
  #ifndef SML
  return 1;
  #endif

  // fetch priority from arguments
  if(argint(0, &priority) < 0)
    return 1;

  acquire(&ptable.lock);
  switch(priority){
    case 1:
      if(curproc->priority == 2){
        pqueue_remove(&queue2, curproc);
        pqueue_insert(&queue1, curproc);
      }else if(curproc->priority == 3){
        pqueue_remove(&queue3, curproc);
        pqueue_insert(&queue1, curproc);
      }
      curproc->priority = 1;
      break;
    case 2:
      if(curproc->priority == 1){
        pqueue_remove(&queue1, curproc);
        pqueue_insert(&queue2, curproc);
      }else if(curproc->priority == 3){
        pqueue_remove(&queue3, curproc);
        pqueue_insert(&queue2, curproc);
      }
      curproc->priority = 2;
      break;
    case 3:
      if(curproc->priority == 1){
        pqueue_remove(&queue1, curproc);
        pqueue_insert(&queue3, curproc);
      }else if(curproc->priority == 2){
        pqueue_remove(&queue2, curproc);
        pqueue_insert(&queue3, curproc);
      }
      curproc->priority = 3;
      break;
    default:
      return 1;
      break;
  }
  release(&ptable.lock);

  return 0;
}
```

DML:

Now, first we will set priority of a new process to 2 just like we did in SML, we will also set the priority of a child process the same as the parent process. And we will also remove a child process from the queue after it is terminated. (The queue structure and implementation is same as in SML)



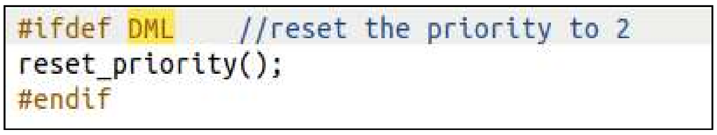
```
  #ifdef DML
  p = pqueue_get(&queue3); // choosing from queue3
  if(p == 0) p = pqueue_get(&queue2); // choose from queue 2
  if(p == 0) p = pqueue_get(&queue1); // choose from queue 1
  if(p != 0){
    // Switch to chosen process.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    switchkvm();
  }
  // Process is done running for now.
  // It should have changed its p->state before coming back.
  c->proc = 0;
  #endif

  release(&ptable.lock);

}
```

<u>DML in Scheduler Function:</u>

1. pqueue_get return the first process that can be run in priority queue
2. First it will choose from queue3 if it is == null it will choose from queue2, if that is also ==null it will choose from q1.
3. if chosen process is != NULL then it will get cpu.
4. For every QUANTUM priority is decremented by one by decpriority() func in trap.c

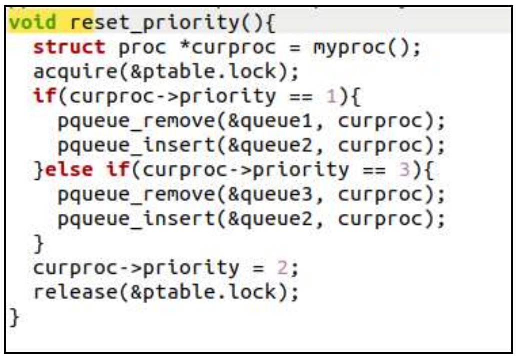
Now, Exec system call will reset the priority to two(2).

This function is called by exec() function in exec.c, so every time a process calls this function it's priority is changed to 2 and we will move the process to its respective queue.



```
#ifdef DML    //reset the priority to 2
reset_priority();
#endif
```

The above code is added in exec.c

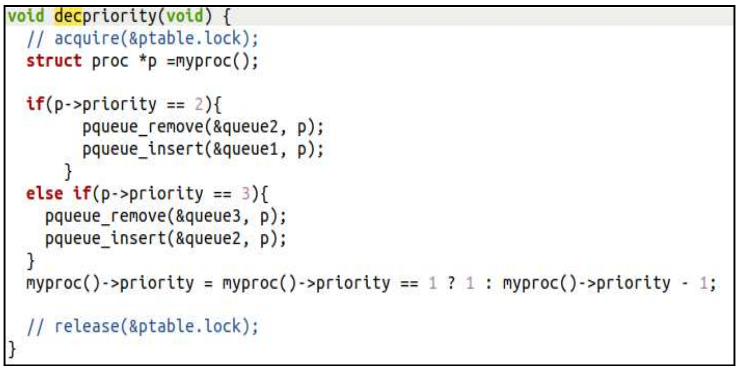


```
void reset_priority(){
  struct proc *curproc = myproc();
  acquire(&ptable.lock);
  if(curproc->priority == 1){
    pqueue_remove(&queue1, curproc);
    pqueue_insert(&queue2, curproc);
  }else if(curproc->priority == 3){
    pqueue_remove(&queue3, curproc);
    pqueue_insert(&queue2, curproc);
  }
  curproc->priority = 2;
  release(&ptable.lock);
}
```

Now, when a process returns from the SLEEPING mode will increase priority of process to highest priority.

A process's state can be changed from SLEEPING to RUNNABLE by using wakeup1(). So in DML mode, we will set the priority of the waking up process to 3 and move it to queue3.

Now, let’s decrease the priority for every quantum:
This function is called in trap.c in DML mode, every time a process runs for a QUANTA.
This decreases the priority by 1 of the process and if the process priority is already 1, then it doesn’t decrease the priority.



```
void decpriority(void) {
  // acquire(&ptable.lock);
  struct proc *p =myproc();

  if(p->priority == 2){
        pqueue_remove(&queue2, p);
        pqueue_insert(&queue1, p);
      }
  else if(p->priority == 3){
    pqueue_remove(&queue3, p);
    pqueue_insert(&queue2, p);
  }
  myproc()->priority = myproc()->priority == 1 ? 1 : myproc()->priority - 1;

  // release(&ptable.lock);
}
```

Task-2

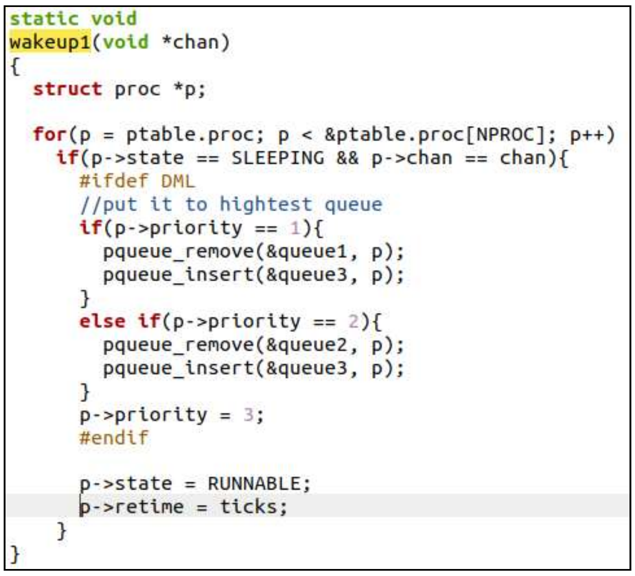
Scheduling:

To add a system call we have to make the changes in some files.

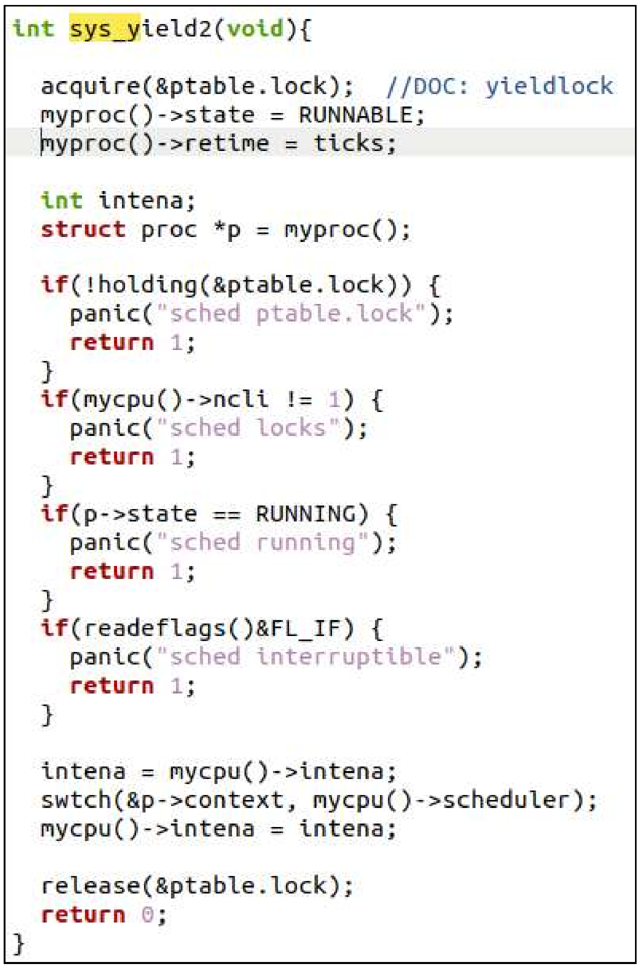
- syscall.h -Defines the system_call .
- syscall.c -Creates pointer sys_yield2 that points to sys_yield2 in syscall.h and Declares
  func prototype “extern int sys_yield2();”.
- user.h- The func for yield2 has been declared in user.h
- Usys.s- Creates interface “SYSCALL(yield2)” for user_program inorder to call system_call
- proc.c -implements “int sys_yield2()” func.

int sys_yield2() in proc.c:

1. Current process that is running is being set to run and will give up the cpu.
2. It will hold only ptable.lock else it returns 1 or if the cpu is taken or if the cpus ncli is not ==
   1, it will return 1.
3. In case everything is correct then it will return 0.



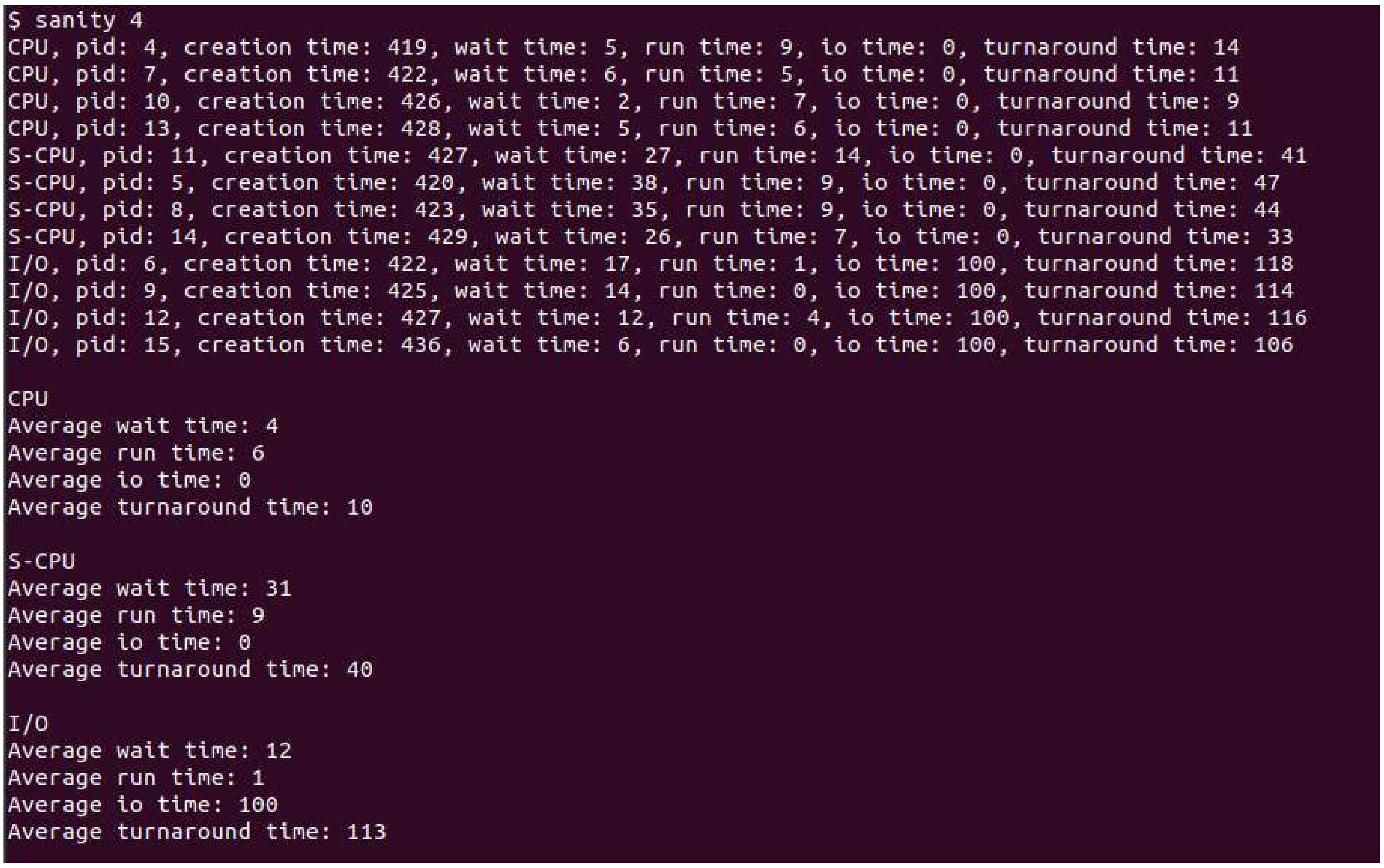
```
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan){
      #ifdef DML
      //put it to hightest queue
      if(p->priority == 1){
        pqueue_remove(&queue1, p);
        pqueue_insert(&queue3, p);
      }
      else if(p->priority == 2){
        pqueue_remove(&queue2, p);
        pqueue_insert(&queue3, p);
      }
      p->priority = 3;
      #endif

      p->state = RUNNABLE;
      p->retime = ticks;
    }
}
```



```
int sys_yield2(void){

  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  myproc()->retime = ticks;

  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock)) {
    panic("sched ptable.lock");
    return 1;
  }
  if(mycpu()->ncli != 1) {
    panic("sched locks");
    return 1;
  }
  if(p->state == RUNNING) {
    panic("sched running");
    return 1;
  }
  if(readeflags()&FL_IF) {
    panic("sched interruptible");
    return 1;
  }

  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;

  release(&ptable.lock);
  return 0;
}
```

TASK-03:

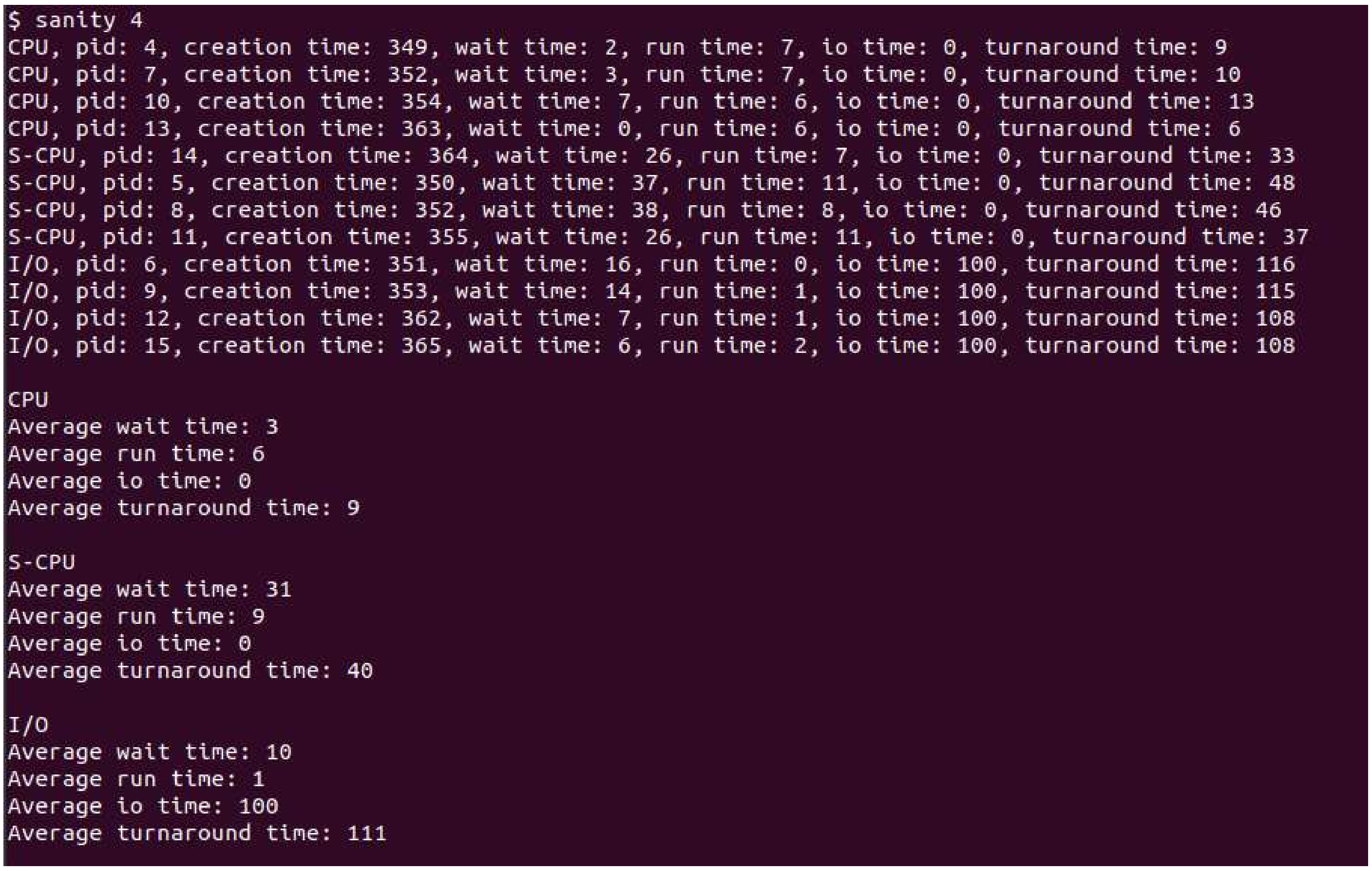
DEFAULT:



```
$ sanity 4
CPU, pid: 4, creation time: 419, wait time: 5, run time: 9, io time: 0, turnaround time: 14
CPU, pid: 7, creation time: 422, wait time: 6, run time: 5, io time: 0, turnaround time: 11
CPU, pid: 10, creation time: 426, wait time: 2, run time: 7, io time: 0, turnaround time: 9
CPU, pid: 13, creation time: 428, wait time: 5, run time: 6, io time: 0, turnaround time: 11
S-CPU, pid: 11, creation time: 427, wait time: 27, run time: 14, io time: 0, turnaround time: 41
S-CPU, pid: 5, creation time: 420, wait time: 38, run time: 9, io time: 0, turnaround time: 47
S-CPU, pid: 8, creation time: 423, wait time: 35, run time: 9, io time: 0, turnaround time: 44
S-CPU, pid: 14, creation time: 429, wait time: 26, run time: 7, io time: 0, turnaround time: 33
I/O, pid: 6, creation time: 422, wait time: 17, run time: 1, io time: 100, turnaround time: 118
I/O, pid: 9, creation time: 425, wait time: 14, run time: 0, io time: 100, turnaround time: 114
I/O, pid: 12, creation time: 427, wait time: 12, run time: 4, io time: 100, turnaround time: 116
I/O, pid: 15, creation time: 436, wait time: 6, run time: 0, io time: 100, turnaround time: 106

CPU
Average wait time: 4
Average run time: 6
Average io time: 0
Average turnaround time: 10

S-CPU
Average wait time: 31
Average run time: 9
Average io time: 0
Average turnaround time: 40

I/O
Average wait time: 12
Average run time: 1
Average io time: 100
Average turnaround time: 113
```

Round Robin scheduling benefits shorter jobs and longer jobs have to wait even more because of context switching . Every process gets a fair amount of CPU time . which benefits recently arrived processes and longer jobs starve much more because of context switching.

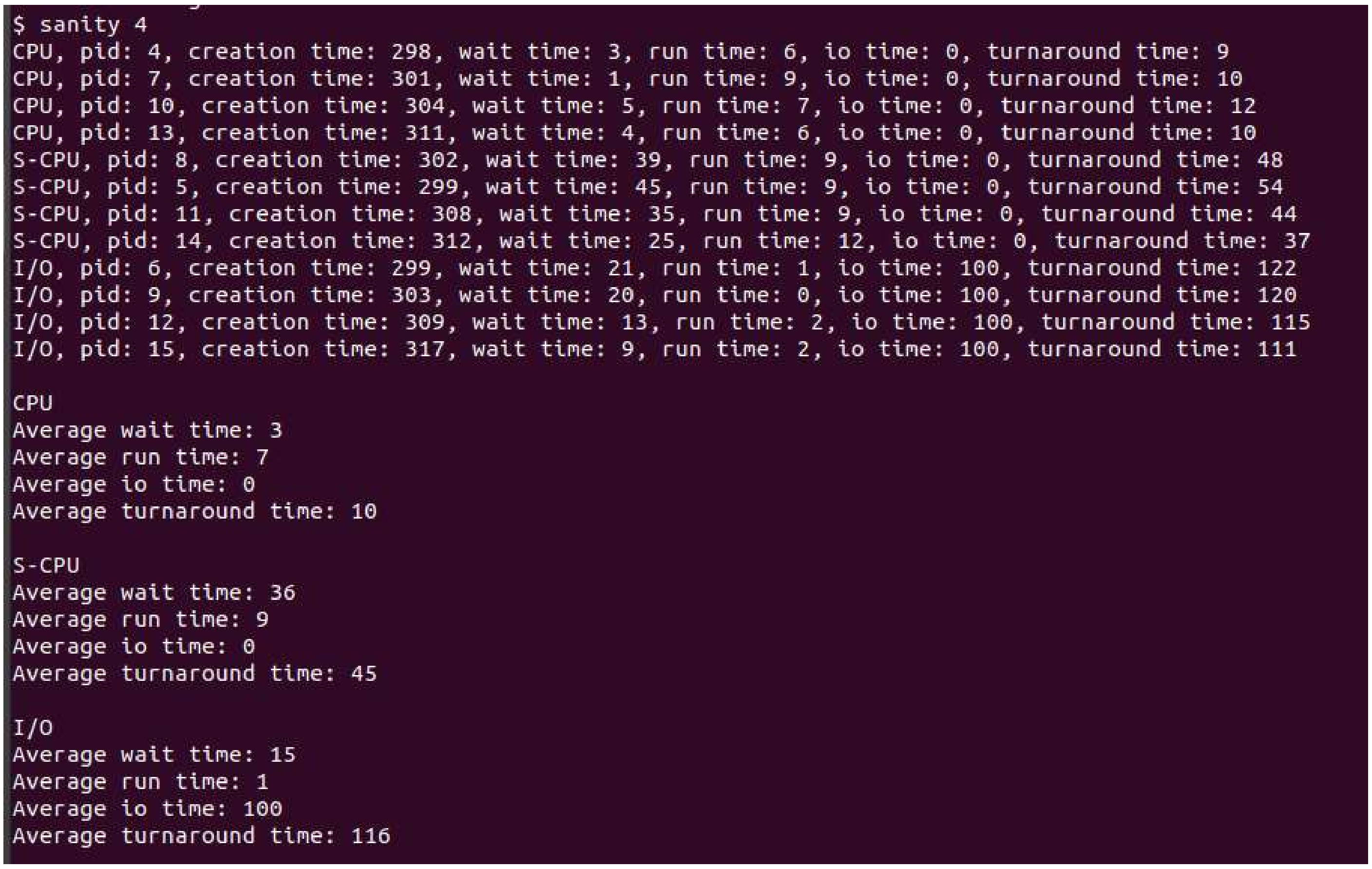
FCFS:



```
$ sanity 4
CPU, pid: 4, creation time: 349, wait time: 2, run time: 7, io time: 0, turnaround time: 9
CPU, pid: 7, creation time: 352, wait time: 3, run time: 7, io time: 0, turnaround time: 10
CPU, pid: 10, creation time: 354, wait time: 7, run time: 6, io time: 0, turnaround time: 13
CPU, pid: 13, creation time: 363, wait time: 0, run time: 6, io time: 0, turnaround time: 6
S-CPU, pid: 14, creation time: 364, wait time: 26, run time: 7, io time: 0, turnaround time: 33
S-CPU, pid: 5, creation time: 350, wait time: 37, run time: 11, io time: 0, turnaround time: 48
S-CPU, pid: 8, creation time: 352, wait time: 38, run time: 8, io time: 0, turnaround time: 46
S-CPU, pid: 11, creation time: 355, wait time: 26, run time: 11, io time: 0, turnaround time: 37
I/O, pid: 6, creation time: 351, wait time: 16, run time: 0, io time: 100, turnaround time: 116
I/O, pid: 9, creation time: 353, wait time: 14, run time: 1, io time: 100, turnaround time: 115
I/O, pid: 12, creation time: 362, wait time: 7, run time: 1, io time: 100, turnaround time: 108
I/O, pid: 15, creation time: 365, wait time: 6, run time: 2, io time: 100, turnaround time: 108

CPU
Average wait time: 3
Average run time: 6
Average io time: 0
Average turnaround time: 9

S-CPU
Average wait time: 31
Average run time: 9
Average io time: 0
Average turnaround time: 40

I/O
Average wait time: 10
Average run time: 1
Average io time: 100
Average turnaround time: 111
```

Processes which arrive first get the cpu and processes which are short and arrive late have to wait till all it before processes are completed.Shorter jobs suffer if longer processes come first.

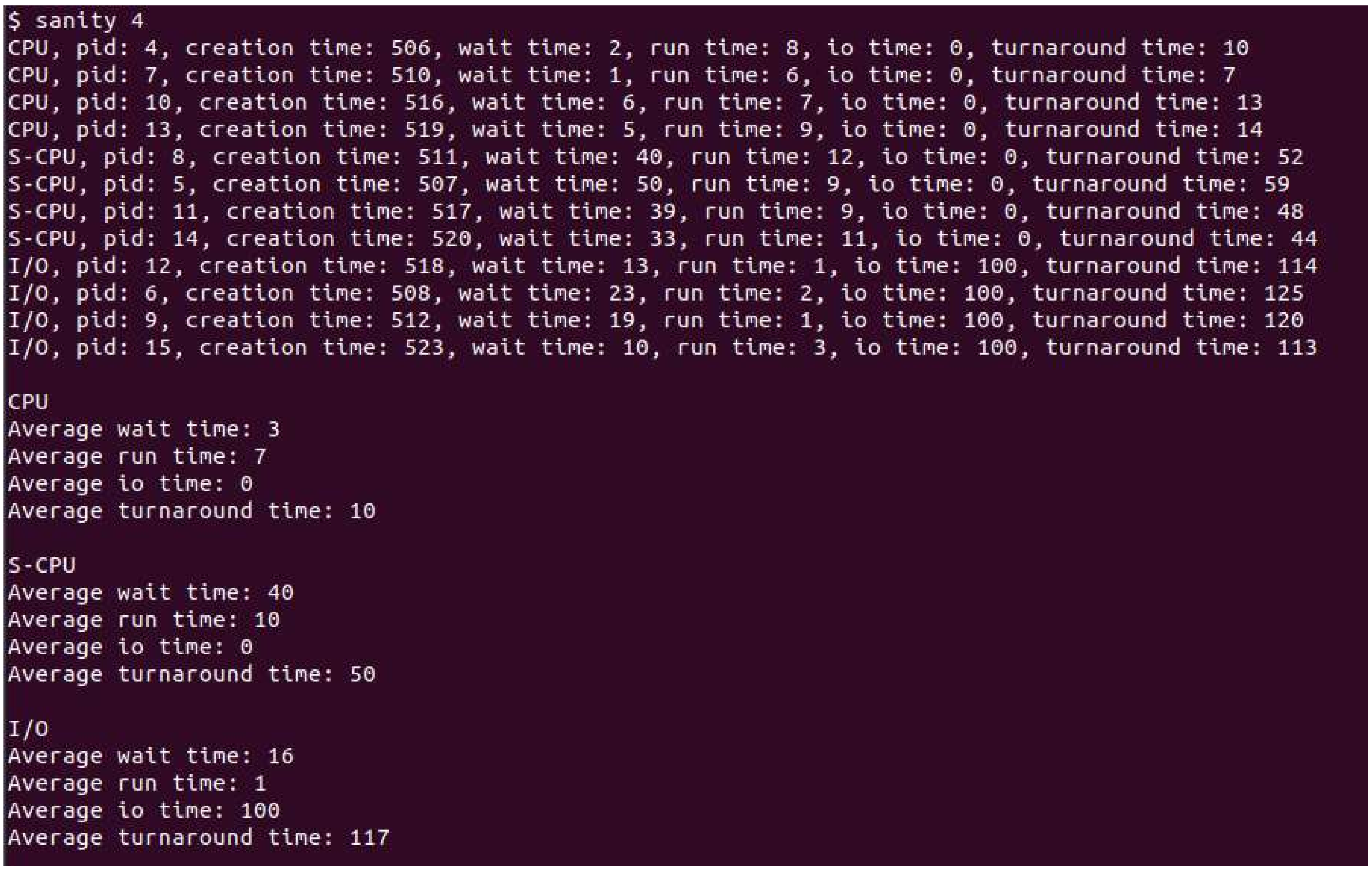
SML



```
$ sanity 4
CPU, pid: 4, creation time: 298, wait time: 3, run time: 6, io time: 0, turnaround time: 9
CPU, pid: 7, creation time: 301, wait time: 1, run time: 9, io time: 0, turnaround time: 10
CPU, pid: 10, creation time: 304, wait time: 5, run time: 7, io time: 0, turnaround time: 12
CPU, pid: 13, creation time: 311, wait time: 4, run time: 6, io time: 0, turnaround time: 10
S-CPU, pid: 8, creation time: 302, wait time: 39, run time: 9, io time: 0, turnaround time: 48
S-CPU, pid: 5, creation time: 299, wait time: 45, run time: 9, io time: 0, turnaround time: 54
S-CPU, pid: 11, creation time: 308, wait time: 35, run time: 9, io time: 0, turnaround time: 44
S-CPU, pid: 14, creation time: 312, wait time: 25, run time: 12, io time: 0, turnaround time: 37
I/O, pid: 6, creation time: 299, wait time: 21, run time: 1, io time: 100, turnaround time: 122
I/O, pid: 9, creation time: 303, wait time: 20, run time: 0, io time: 100, turnaround time: 120
I/O, pid: 12, creation time: 309, wait time: 13, run time: 2, io time: 100, turnaround time: 115
I/O, pid: 15, creation time: 317, wait time: 9, run time: 2, io time: 100, turnaround time: 111

CPU
Average wait time: 3
Average run time: 7
Average io time: 0
Average turnaround time: 10

S-CPU
Average wait time: 36
Average run time: 9
Average io time: 0
Average turnaround time: 45

I/O
Average wait time: 15
Average run time: 1
Average io time: 100
Average turnaround time: 116
```

Higher priority processes gets the cpu faster and the lower priority processes have to wait until all the processes with higher priority are completed .Lower priority queues suffer if higher priority processes keeps coming

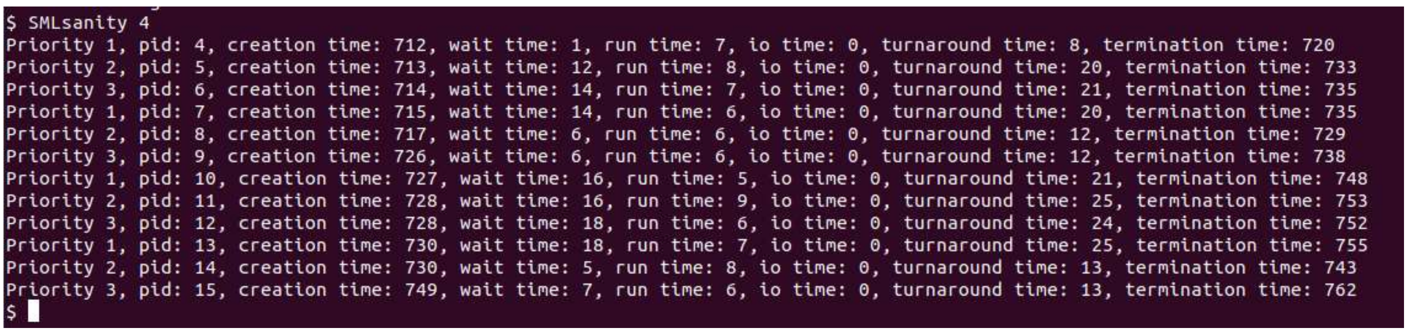
DML



```
$ sanity 4
CPU, pid: 4, creation time: 506, wait time: 2, run time: 8, io time: 0, turnaround time: 10
CPU, pid: 7, creation time: 510, wait time: 1, run time: 6, io time: 0, turnaround time: 7
CPU, pid: 10, creation time: 516, wait time: 6, run time: 7, io time: 0, turnaround time: 13
CPU, pid: 13, creation time: 519, wait time: 5, run time: 9, io time: 0, turnaround time: 14
S-CPU, pid: 8, creation time: 511, wait time: 40, run time: 12, io time: 0, turnaround time: 52
S-CPU, pid: 5, creation time: 507, wait time: 50, run time: 9, io time: 0, turnaround time: 59
S-CPU, pid: 11, creation time: 517, wait time: 39, run time: 9, io time: 0, turnaround time: 48
S-CPU, pid: 14, creation time: 520, wait time: 33, run time: 11, io time: 0, turnaround time: 44
I/O, pid: 12, creation time: 518, wait time: 13, run time: 1, io time: 100, turnaround time: 114
I/O, pid: 6, creation time: 508, wait time: 23, run time: 2, io time: 100, turnaround time: 125
I/O, pid: 9, creation time: 512, wait time: 19, run time: 1, io time: 100, turnaround time: 120
I/O, pid: 15, creation time: 523, wait time: 10, run time: 3, io time: 100, turnaround time: 113

CPU
Average wait time: 3
Average run time: 7
Average io time: 0
Average turnaround time: 10

S-CPU
Average wait time: 40
Average run time: 10
Average io time: 0
Average turnaround time: 50

I/O
Average wait time: 16
Average run time: 1
Average io time: 100
Average turnaround time: 117
```

Higher priority processes gets the cpu faster and the processes with same priority get the equal chances of getting cpu irrespective of their arrival time.Longer priority jobs suffer because of context switching .

SMLsanity:



```
$ SMLsanity 4
Priority 1, pid: 4, creation time: 712, wait time: 1, run time: 7, io time: 0, turnaround time: 8, termination time: 720
Priority 2, pid: 5, creation time: 713, wait time: 12, run time: 8, io time: 0, turnaround time: 20, termination time: 733
Priority 3, pid: 6, creation time: 714, wait time: 14, run time: 7, io time: 0, turnaround time: 21, termination time: 735
Priority 1, pid: 7, creation time: 715, wait time: 14, run time: 6, io time: 0, turnaround time: 20, termination time: 735
Priority 2, pid: 8, creation time: 717, wait time: 6, run time: 6, io time: 0, turnaround time: 12, termination time: 729
Priority 3, pid: 9, creation time: 726, wait time: 6, run time: 6, io time: 0, turnaround time: 12, termination time: 738
Priority 1, pid: 10, creation time: 727, wait time: 16, run time: 5, io time: 0, turnaround time: 21, termination time: 748
Priority 2, pid: 11, creation time: 728, wait time: 16, run time: 9, io time: 0, turnaround time: 25, termination time: 753
Priority 3, pid: 12, creation time: 728, wait time: 18, run time: 6, io time: 0, turnaround time: 24, termination time: 752
Priority 1, pid: 13, creation time: 730, wait time: 18, run time: 7, io time: 0, turnaround time: 25, termination time: 755
Priority 2, pid: 14, creation time: 730, wait time: 5, run time: 8, io time: 0, turnaround time: 13, termination time: 743
Priority 3, pid: 15, creation time: 749, wait time: 7, run time: 6, io time: 0, turnaround time: 13, termination time: 762
$ 
```

Higher priority queues get terminated faster than lower priority queues.
In the above screenshot , pid 9,10 are created almost at the same time and pid 10 has lower priority than pid 9. Also the run time of pid 9 is more than of pid 10 but pid 9 terminated before pid 10.
So, we can see that SMLsanity works. Termination time also depends on creation time.