CS 344 : Operating Systems Lab — Lab Assignment 2

Group Number: 28

Tanay Maheshwari	190101092
Shreyansh Meena	190101084
Vignesh Ravichandra Rao	190101109
Nikhil Kumar Pandey	190123040

Commands to reproduce changes:
The patch file **`G28_lab2b.patch`** is included in the submission. This patch file works on the entire (unedited) XV6 source directory. Assuming that the XV6 source directory on your machine is **`xv6-public`**, run the following command from the parent directory:

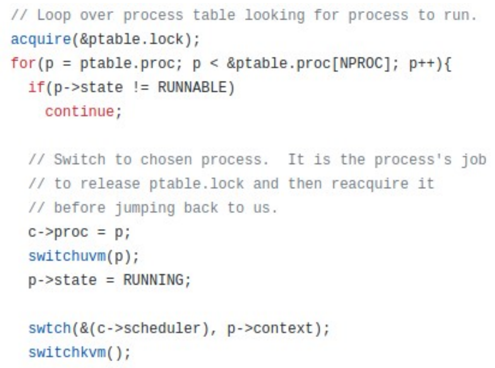
```
patch -uN -d xv6-public < G28_lab2b.patch
```

In case the name of the XV6 source code directory is different, replace **`xv6-public`** in the above command with the appropriate directory name. **Note**: This patch is for all tasks.

Task 1) Scheduling

Observations about default scheduling policy in XV6:

- Q1) Which process does the policy select for running?
  - The scheduler selects the first RUNNABLE process available in the process table (ptable.proc).

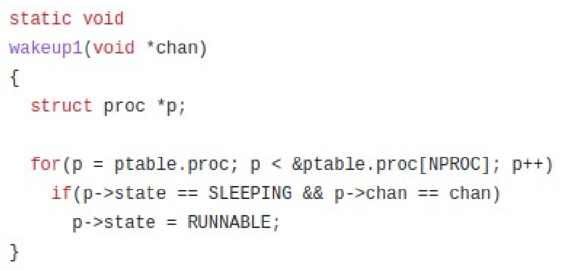


```
// Loop over process table looking for process to run.
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  // 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();
```

Figure 1: Q1) scheduler

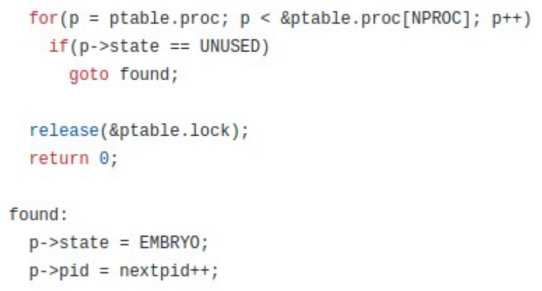


```
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}
```

Figure 2: Q2) wakeup1()

- Q2) What happens when a process returns from I/O?
  - When a process is in IO wait stage, its state is SLEEPING. Upon returning from IO wait, the wakeup1() function is called which changes its state to RUNNABLE.
- Q3) What happens when a new process is created?
  - The allocproc() function is called. This function searched for a process in UNUSED state and assigns the new process to it. It then initialises the PCB fields like pid, registers etc and also changes the state to EMBRYO
- Q4) When/how often does the scheduling take place?
  - Scheduling takes place every clock tick. This can be seen in trap.c where the scheduler is called after every tick which means a time QUANTA of 1.

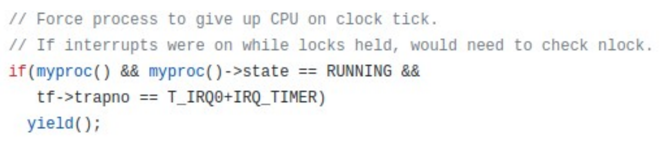


```
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;

  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;
  p->pid = nextpid++;
```

Figure 3: Q3) allocproc()



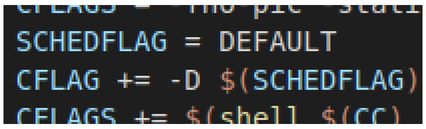
```
// Force process to give up CPU on clock tick.
// If interrupts were on while locks held, would need to check nlock.
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

Figure 4: Q4) yield() called after every TIMER interrupt i.e every tick

Therefore, the default XV6 scheduling is a form of Round Robin with time slice = 1 clock cycle.

Implementation details

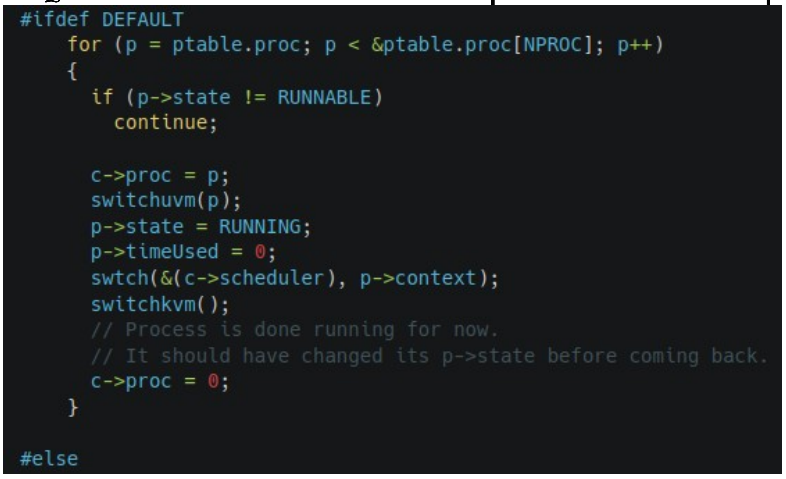
- QUANTA was defined as 5 in params.h . This implies that each time slice is 5 ticks long.
- Makefile was edited to include the SCHEDFLAG macro as follows:



```
SCHEDFLAG = DEFAULT
CFLAG += -D $(SCHEDFLAG)
CFLAGS += $(shell $(CC)
```

Policy 1: DEFAULT

- The proc structure used to store details about the process in the PCB was edited to include the "`timeUsed`" field. This indicates the number of clock cycles for which the process has been running in the current time QUANTA.
  - When the `scheduler()` selects a process, it resets the timeUsed to 0.
- In trap.c, the `yield()` is called to choose the next process only when `timeUsed == QUANTA` which means the process has completed its timeslice.



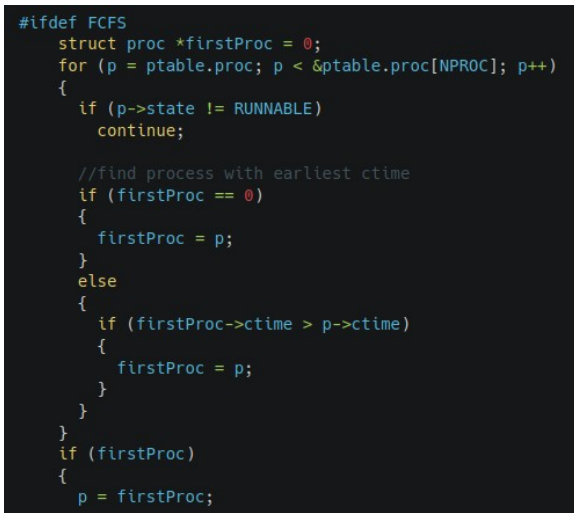
```
#ifdef DEFAULT
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state != RUNNABLE)
        continue;

      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;
      p->timeUsed = 0;
      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
```

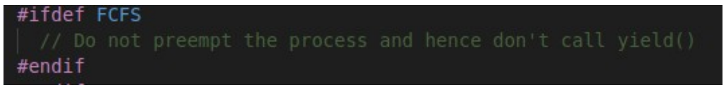
Policy 2: FCFS

- The scheduler selects a RUNNABLE process with minimum creation time (ctime field in proc structure) as the next process to run.
- No preemption takes place in trap.c as well.



```
#ifdef FCFS
    struct proc *firstProc = 0;
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->state != RUNNABLE)
        continue;

      //find process with earliest ctime
      if (firstProc == 0)
      {
        firstProc = p;
      }
      else
      {
        if (firstProc->ctime > p->ctime)
        {
          firstProc = p;
        }
      }
    }
    if (firstProc)
    {
      p = firstProc;
```

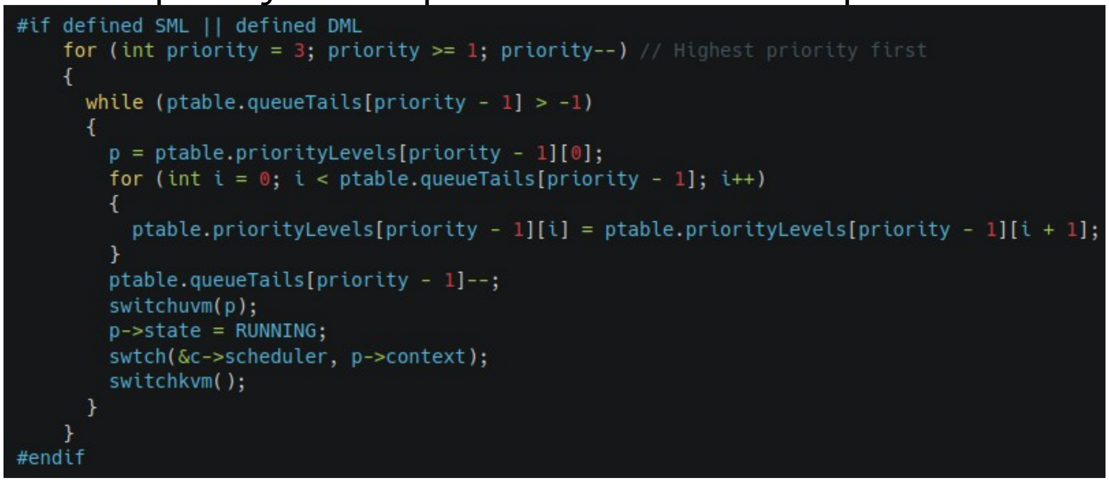


```
#ifdef FCFS
  // Do not preempt the process and hence don't call yield()
#endif
```

Policy 3: SML

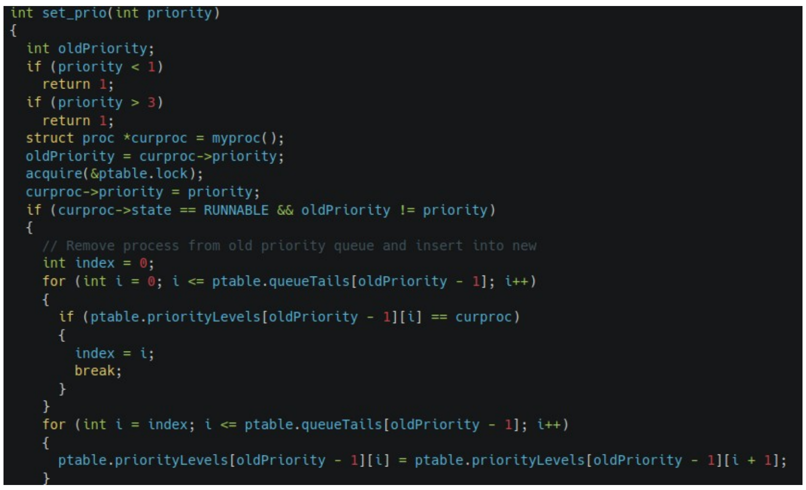
- The ptable structure was edited to include three priority level queues.

- The queue for priority = n is indexed at i = n-1 in the ptable.priorityLevels[i][] array
- The index of the last process present at each level is stored with ptable.queueTails[3] where ptable.queueTails[i] = x implies that the last process with priority = x+1 is present at index i in the queue.

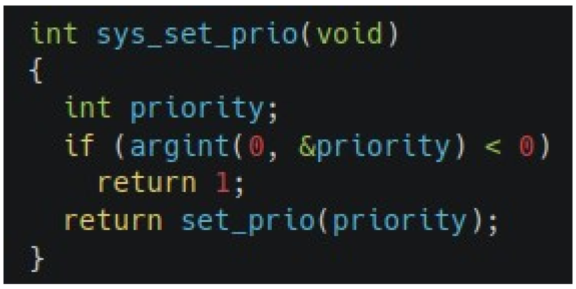


```
#if defined SML || defined DML
    for (int priority = 3; priority >= 1; priority--) // Highest priority first
    {
      while (ptable.queueTails[priority - 1] > -1)
      {
        p = ptable.priorityLevels[priority - 1][0];
        for (int i = 0; i < ptable.queueTails[priority - 1]; i++)
        {
          ptable.priorityLevels[priority - 1][i] = ptable.priorityLevels[priority - 1][i + 1];
        }
        ptable.queueTails[priority - 1]--;
        switchuvm(p);
        p->state = RUNNING;
        swtch(&c->scheduler, p->context);
        switchkvm();
      }
    }
#endif
```

- The scheduler shown above iterates through the queue starting from the highest priority = 3 to 1.
- The first element of the selected queue is chosen for RUNNING and the entire queue is shifted one position to the left and the queueTail is reduced by one to signify the removal of the process at queue head.
- A system call **`int set_prio(int)`** was created to allow a process to set its priority as follows:



```
int set_prio(int priority)
{
  int oldPriority;
  if (priority < 1)
    return 1;
  if (priority > 3)
    return 1;
  struct proc *curproc = myproc();
  oldPriority = curproc->priority;
  acquire(&ptable.lock);
  curproc->priority = priority;
  if (curproc->state == RUNNABLE && oldPriority != priority)
  {
    // Remove process from old priority queue and insert into new
    int index = 0;
    for (int i = 0; i <= ptable.queueTails[oldPriority - 1]; i++)
    {
      if (ptable.priorityLevels[oldPriority - 1][i] == curproc)
      {
        index = i;
        break;
      }
    }
    for (int i = index; i <= ptable.queueTails[oldPriority - 1]; i++)
    {
      ptable.priorityLevels[oldPriority - 1][i] = ptable.priorityLevels[oldPriority - 1][i + 1];
    }
```

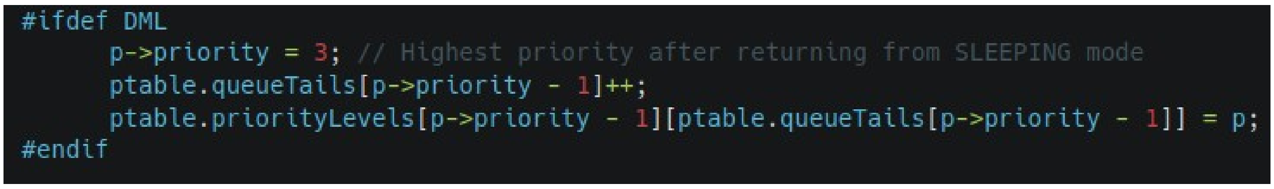


```
int sys_set_prio(void)
{
  int priority;
  if (argint(0, &priority) < 0)
    return 1;
  return set_prio(priority);
}
```

  - In set_prio the ptable lock was acquired. The priority of the process is then changed and the process is added into the new priority queue.
  - This system call is assigned number 25 in syscall.h and sys_set_prio is used as the kernel side system call.

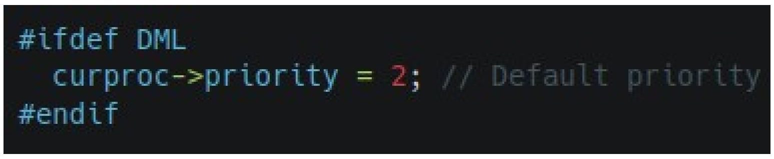
Policy 4: DML

- This follows the same procedure for selecting a process from the priority queues as SML.
- The dynamic process priority assignment rules were implemented as follows:
  - In the wakeup1() function which is called after a process returns from IO wait stage, it is assigned priority 3 (highest priority).



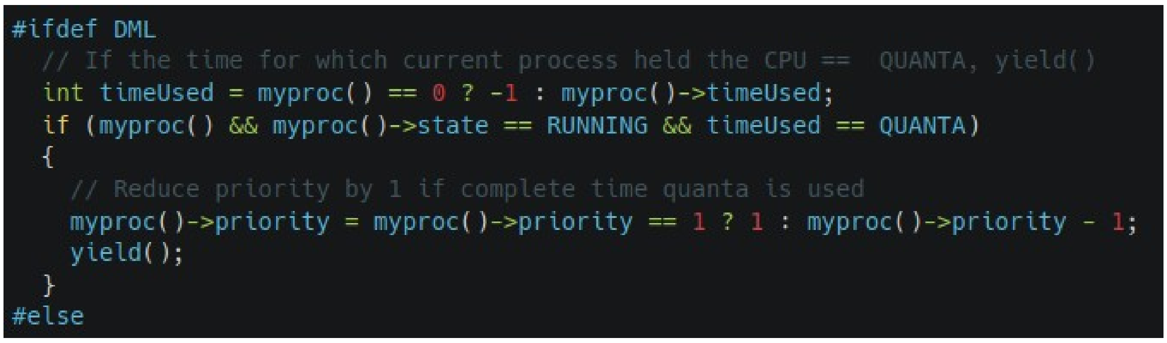
```
#ifdef DML
      p->priority = 3; // Highest priority after returning from SLEEPING mode
      ptable.queueTails[p->priority - 1]++;
      ptable.priorityLevels[p->priority - 1][ptable.queueTails[p->priority - 1]] = p;
#endif
```

- In exec.c the process is given priority = 2.



```
#ifdef DML
  curproc->priority = 2; // Default priority
#endif
```

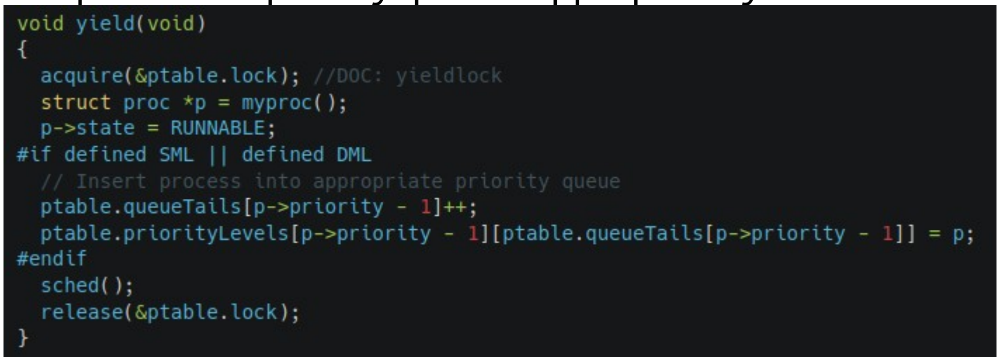
- In trap.c, if a process utilises its complete time QUANTA, it's priority is reduced by 1 until it reaches the lowest priority = 1.



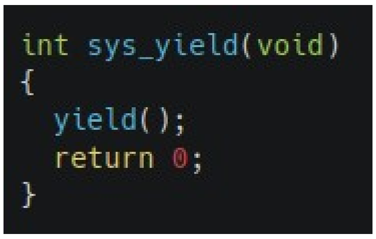
```
#ifdef DML
  // If the time for which current process held the CPU ==  QUANTA, yield()
  int timeUsed = myproc() == 0 ? -1 : myproc()->timeUsed;
  if (myproc() && myproc()->state == RUNNING && timeUsed == QUANTA)
  {
    // Reduce priority by 1 if complete time quanta is used
    myproc()->priority = myproc()->priority == 1 ? 1 : myproc()->priority - 1;
    yield();
  }
#else
```

Task 2) yield()

- The yield() function is present in proc.c and it lets the currently running process gracefully give up the processor for another process to run.
- In order to accomodate the SML and DML priorities, the yield() function updates the priority queues appropriately



```
void yield(void)
{
  acquire(&ptable.lock); //DOC: yieldlock
  struct proc *p = myproc();
  p->state = RUNNABLE;
#if defined SML || defined DML
  // Insert process into appropriate priority queue
  ptable.queueTails[p->priority - 1]++;
  ptable.priorityLevels[p->priority - 1][ptable.queueTails[p->priority - 1]] = p;
#endif
  sched();
  release(&ptable.lock);
}
```



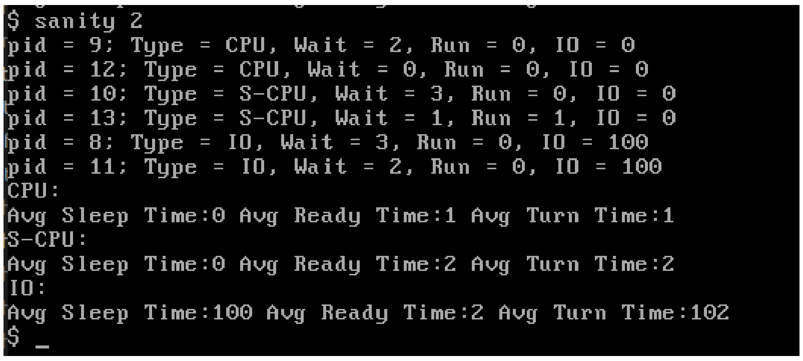
```
int sys_yield(void)
{
  yield();
  return 0;
}
```

- The system call was added by including the prototype in user.h and assigning system call number 26 to sys_yield in syscall.h
  - The yield() doesn't fail unless ptable lock can't be acquired.
  - This case is taken care of by XV6 panic

Task 3.1) General Sanity Test

Default Scheduling:

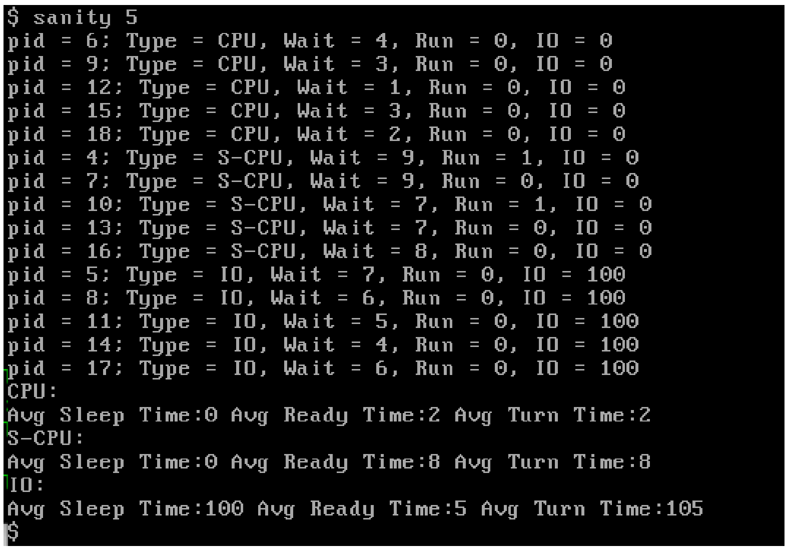
- We would expect the IO bound process to spend more time in I/O Wait since it sleeps before completing its entire time quanta.



```
$ sanity 2
pid = 9; Type = CPU, Wait = 2, Run = 0, IO = 0
pid = 12; Type = CPU, Wait = 0, Run = 0, IO = 0
pid = 10; Type = S-CPU, Wait = 3, Run = 0, IO = 0
pid = 13; Type = S-CPU, Wait = 1, Run = 1, IO = 0
pid = 8; Type = IO, Wait = 3, Run = 0, IO = 100
pid = 11; Type = IO, Wait = 2, Run = 0, IO = 100
CPU:
Avg Sleep Time:0 Avg Ready Time:1 Avg Turn Time:1
S-CPU:
Avg Sleep Time:0 Avg Ready Time:2 Avg Turn Time:2
IO:
Avg Sleep Time:100 Avg Ready Time:2 Avg Turn Time:102
$ _
```

FCFS Scheduling:

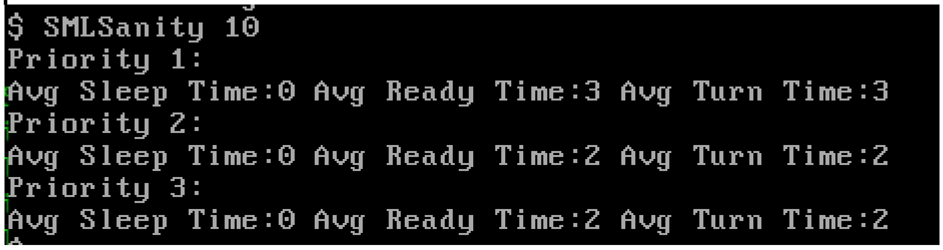
- Processes the were forked initially will get completed before later processes. Therefore, we would expect an increase in wait time as the number of processes grow.



```
$ sanity 5
pid = 6: Type = CPU, Wait = 4, Run = 0, IO = 0
pid = 9: Type = CPU, Wait = 3, Run = 0, IO = 0
pid = 12: Type = CPU, Wait = 1, Run = 0, IO = 0
pid = 15: Type = CPU, Wait = 3, Run = 0, IO = 0
pid = 18: Type = CPU, Wait = 2, Run = 0, IO = 0
pid = 4: Type = S-CPU, Wait = 9, Run = 1, IO = 0
pid = 7: Type = S-CPU, Wait = 9, Run = 0, IO = 0
pid = 10: Type = S-CPU, Wait = 7, Run = 1, IO = 0
pid = 13: Type = S-CPU, Wait = 7, Run = 0, IO = 0
pid = 16: Type = S-CPU, Wait = 8, Run = 0, IO = 0
pid = 5: Type = IO, Wait = 7, Run = 0, IO = 100
pid = 8: Type = IO, Wait = 6, Run = 0, IO = 100
pid = 11: Type = IO, Wait = 5, Run = 0, IO = 100
pid = 14: Type = IO, Wait = 4, Run = 0, IO = 100
pid = 17: Type = IO, Wait = 6, Run = 0, IO = 100
CPU:
Avg Sleep Time:0 Avg Ready Time:2 Avg Turn Time:2
S-CPU:
Avg Sleep Time:0 Avg Ready Time:8 Avg Turn Time:8
IO:
Avg Sleep Time:100 Avg Ready Time:5 Avg Turn Time:105
$
```

Task 3.2) SML Sanity Test

We would expect the process with lowest priority that is priority 1 to have a greater ready time since priority 1 processes are scheduled only after all priority 2 and 3 processes complete.



```
$ SMLSanity 10
Priority 1:
Avg Sleep Time:0 Avg Ready Time:3 Avg Turn Time:3
Priority 2:
Avg Sleep Time:0 Avg Ready Time:2 Avg Turn Time:2
Priority 3:
Avg Sleep Time:0 Avg Ready Time:2 Avg Turn Time:2
```