CS 3423 Operating Systems
Fall Semester 2017
Prof. Pai H. Chou

Assignment 6

Due Date: Sunday, October 29, 2017, 11:59pm
One day late submission without penalty
One week late submssion with 20% penalty
Submit electronically on iLMS

What to submit: One zip file named <studentID>`-hw6.zip` (replace <studentID> with your own student ID). It should contain four files:

- one PDF file named **hw6.pdf** for Section 1 and Section 2. Write your answers in **English**. Check your spelling and grammar. Include your name and student ID!
- Section 2: Python source files. Include your name and student ID in the program comments on top.
    - Section 2.2: **task.py**
    - Section 2.3: **npsched.py** (for non-preemptive scheduler) and **typescript3**
    - Section 2.4 **psched.py** (for preemptive scheduler) and **typescript4**
    - Section 2.5 **typescript5**

1. [40 points] Problem Set

1. [20 points] 5.10 Which of the following scheduling algorithms could result in starvation? Explain.
    a. First-come, first-served
    b. Shortest job first
    c. Round robin
    d. Priority
2. [20 points] 5.12 Consider a system running ten I/O-bound tasks and one CPU-bound task. Assume that the I/O-bound tasks issue an I/O operation once for every millisecond of CPU computing and that each I/O operation takes 10 milliseconds to complete. Also assume that the context-switching overhead is 0.1 millisecond and that all processes are long-running tasks. Describe the CPU utilization for a round-robin scheduler when:
    a. The time quantum is 1 millisecond
    b. The time quantum is 10 milliseconds

2. [60 points] Programming Exercise

In this programming exercise, you are to build a CPU scheduler that can compute the schedule for a variety of policies and calculate the various cost functions.

2.1 FIFO and Priority Queue

A fundamental data structure in any CPU scheduler is a queue. Here, it can refer to a FIFO (first-in first-out) queue, but it may also refer to a priority queue, a LIFO (last-in first-out, also known as a stack), etc. Unlike random-access memory, where the reader or writer provides the memory address explicitly, a queue keeps track of its own addresses and provides only `.get()` and `.put()` methods for reading and writing one element at a time. The following class is provided as an example:

---------- file "[fifo.py]" ----------

```
class FIFO:
   def __init__(self, initList=[]):
      self.A = list(initList)
   def get(self):        # remove element and return itse value
      return self.A.pop(0) # throws underflow exception if empty
   def put(self, val):  # add element
      self.A.append(val)
   def head(self): # A[0] if not empty, None instead of underflow exception
      return len(self.A) and self.A[0] or None
   def __iter__(self):  # iterator over its elements
      for i in self.A:  # convertable to tuple, list, for-in loop, etc
         yield i
   def __len__(self):  # allows caller to call len(f) where f is FIFO
      return len(self.A)
   def __repr__(self): # shows a representation; we just show it as list
      return repr(self.A)
```

This will handle any data type. An example is (assume you save it in fifo.py)

```
>>> from fifo import *
>>> f = FIFO(range(3))
>>> f
[0, 1, 2]
>>> f.put(6)
>>> f.get()
0
>>> f.head()
1
>>> len(f)
3
```

In addition, we also provide an implementation of a priority queue based on min-heap. You don't get the source code but you can import it from the minheap.pyc file provided: (updated 10/27 17:30)

minheap.pyc (for python 2.7.13)
minheap.pyc (for python 3.5.2)
minheap.pyc (for python 3.5.3)
minheap.pyc (for python 3.6.2)

It has the following API:

--------- file "minheap.py" -----------

```
class MinHeap:
   def __init__(self):
   def __len__(self):
   def __iter__(self):
   def __repr__(self):
   def get(self):
   def put(self, value):
   def head(self):
   def buildheap(self):  # reinitialize content to be heap again
```

One difference is that your minheap data structure typecasts its elements to tuples before comparison, and Python will compare tuples in lexicographical order, and we will exploit this characteristic later when prioritizing tasks to run.

```
>>> from minheap import MinHeap
>>> h = MinHeap()
>>> for i in [(2,3), (3,4), (2,4), (4,5), (5, 6)]: h.put(i)
...
>>> h
[(2, 3), (3, 4), (2, 4), (4, 5), (5, 6)]
>>> h.get()
(2, 3)
>>> h
[(2, 4), (3, 4), (5, 6), (4, 5)]
>>> h.get()
(2, 4)
>>> h
[(3, 4), (4, 5), (5, 6)]
>>> h.put((6,7))
>>> h.get()
(3, 4)
>>> h
[(5, 6), (4, 5), (6, 7)]
```

2.2 Task class [10 points]

You need to declare a `Task` class for representing the properties of a task to be scheduled, including properties given by the user and additional data for bookkeeping purpose. Here, we use the term Task to mean the workload to be performed, with or without having a process or a thread attached to it. A thread or process may be recycled to run different tasks over time. But sometimes tasks and processes are used interchangeably when the task is attached to a process. The given data are passed as arguments to the constructors. You may use the following template to define your task. Look for the italicized comments to add your own code.

--------- file "task-template.py" : save and rename it as "task.py" ----------

```
class Task:
   def __init__(self, name, release, cpuBurst):
      # the task has a string name, release time and cpuBurst.
      # the constructor may also need to initialize other fields,
      # for statistics purpose.  Examples include
```

```
        # waiting time, remaining time, last dispatched time, and
        # completion time

    def __repr__(self):
        # returns a string that looks like constructor syntax
      return self.__class__.__name__ + '(%s, %d, %d)' % (repr(self.name),\
                    self.release, self.cpuBurst)
    def __str__(self):
      return self.name
    _KNOWN_SCHEMES = ["FCFS", "SJF", "RR"]
    def setPriorityScheme(self, scheme="SJF"):
      """
         the scheme can be "FCFS", "SJF", "RR", etc
      """
      self.scheme = scheme
      if not scheme in _KNOWN_SCHEMES:
         raise ValueError("unknown scheme %s: must be FCFS, SJF, RR" % scheme)
    def decrRemaining(self):
        # call this method to decrement the remaining time of this task
    def remainingTime(self):
        # returns the remaining time of this task
    def done(self):
        # returns a boolean for if this task has remaining work to do
    def setCompletionTime(self, time):
        # record the clock value when the task is completed
    def turnaroundTime(self):
        # returns the turnaround time of this task, as defined on
        # slide 10 of week 7 lecture
    def incrWaitTime(self):
        # increment the amount of waiting time for this task
    def releaseTime(self):
        # returns the release time of this task
    def __iter__(self):
        # this enables converting the task into a tuple() type so that
        # the priority queue can just cast it to tuple before comparison.
        #  it depends on the policy
        if (self.scheme == 'FCFS'):
           t = # your tuple that defines the priority
        elif (self.scheme == 'SJF'): # shortest job first
           t = # your tuple that defines the priority
        elif (self.scheme == 'RR'): # round robin
           t = # your tuple that defines the priority
        else:
           raise ValueError("Unknown scheme %s" % self.scheme)
        for i in t:
           yield i
```

2.3 Nonpreemptive Scheduler (20 points)

The NPScheduler class is instantiated with a policy and up to N time steps. Then the caller may add tasks to be scheduled, either as the scheduler runs or all at the beginning. The scheduler runs one time step at a time to fill in the Gantt chart with scheduled tasks. It also provides methods for the statistics. Use the following template (npsched-template.py, rename it as `npsched.py`) to make your scheduler

```
from fifo import FIFO
from minheap import MinHeap
from task import Task
class NPScheduler: # nonpreemptive scheduler
   def __init__(self, N, policy='SJF'):
      self.N = N  # number of timesteps to schedule
      self.running = None
      self.clock = 0    # the current timestep being scheduled
      self.policy = policy
      # instantiate the readyQueue, which may be a FIFO or MinHeap
      # you may need additional queues for
      # - tasks that have been added but not released yet
      # - tasks that have been completed
      # - the Gantt chart

   def addTask(self, task):
      # if the release time of the new task is not in the future, then
      # put it in ready queue; otherwise, put into not-ready queue.
      # you may need to copy the scheduler policy into the task

   def dispatch(self, task):
      # dispatch here means assign the chosen task as the one to run
      # in the current time step.
      # the task should be removed from ready-queue by caller;
      # The task may be empty (None).
      # This method will make an entry into the Gantt chart and perform
      # bookkeeping, including
      # - recording the last dispatched time of this task,
      # - increment the wait times of those tasks not scheduled
      #   but in the ready queue

   def releaseTasks(self):
     '''
        this is called at the beginning of scheduling each time step to see
        if new tasks became ready to be released to ready queue, when their
        release time is no later than the current clock.
     '''
     while True:
        r = self.notReadyQueue.head()
        # assuming the not-Ready Queue outputs by release time
        if r is None or r.releaseTime() > self.clock:
           break
        r = self.notReadyQueue.get()
        r.setPriorityScheme(self.policy)
        self.readyQueue.put(r)
```

```python
    def checkTaskCompletion(self):
        # if there is a current running task, check if it has just finished.
        # (i.e., decrement remaining time and see if it has more work to do.
        # If so, perform bookkeeping for completing the task,
        # - move task to done-queue, set its completion time and lastrun time
        # set the scheduler running task to None, and return True
        # (so that a new task may be picked.)
        # but if not completed, return False.
        # If there is no current running task, also return True.
        if self.running is None:
            return True
        # your code here

    def schedule(self):
        # scheduler that handles nonpreemptive scheduling.
        # the policy such as RR, SJF, or FCFS is handled by the task as it
        # defines the attribute to compare (in its __iter__() method)
        # first, check if added but unreleased tasks may now be released
        # (i.e., added to ready queue)
        self.releaseTasks()
        if self.checkTaskCompletion() == False:
            # There is a current running task and it is not done yet!
            # the same task will continue running to its completion.
            # simply redispatch the current running task.
        else:
            # task completed or no running task.
            # get the next task from priority queue and dispatch it.

    def clockGen(self):
        # this method runs the scheduler one time step at a time.
        for self.clock in range(self.N):
            # now run scheduler here
            self.schedule()
            yield self.clock

    def getSchedule(self):
        return '-'.join(map(str, self.ganttChart))


def testNPScheduler(tasks, policy):
    nClocks = 20
    scheduler = NPScheduler(nClocks, policy)

    for t in tasks:
        scheduler.addTask(t)

    for clock in scheduler.clockGen():
        pass
```

```
    print('nonpreemptive %s: %s' % (scheduler.policy,
scheduler.getSchedule()))

if __name__ == '__main__':
    tasks = [Task(*i) for i in [('A', 0, 7), ('B', 2, 4), ('C', 4, 1), ('D',
5, 4)]]
    print('tasks = %s' % tasks)
    for policy in ['SJF', 'FCFS', 'RR']:
        tasks = [Task(*i) for i in [('A', 0, 7), ('B', 2, 4), ('C', 4, 1),
('D', 5, 4)]]
        testNPScheduler(tasks, policy)

--------- Your output would look like this:

$ python3 npscheduler.py
tasks = [Task('A', 0, 7), Task('B', 2, 4), Task('C', 4, 1), Task('D', 5, 4)]
nonpreemptive SJF: A-A-A-A-A-A-A-C-B-B-B-B-D-D-D-D-None-None-None-None
nonpreemptive FCFS: A-A-A-A-A-A-A-B-B-B-B-C-D-D-D-D-None-None-None-None
nonpreemptive RR: A-A-A-A-A-A-A-B-B-B-B-C-D-D-D-D-None-None-None-None
```

2.4 Preemptive Scheduler (20 points)

For this part, make a copy of your nonpreemptive scheduler and make it a preemptive one.

The overall structure is the same as the Nonpreemptive scheduler.
-------- file “psched-template.py”, rename and save as “psched.py”

```
class PScheduler(NPScheduler): # subclass from nonpreemptive scheduler
    # this means it can inherit
    # __init__(), addTask(), dispatch(), releaseTasks()
    # clockGen(), getSchedule()

    def preempt(self):
        # this is the new method to add to put the running task
        # back into ready queue, plus any bookkeeping if necessary.

    def schedule(self):
        self.releaseTasks() # same as before
        if self.checkTaskCompletion() == False:
            # still have operation to do.
            # see if running task or next ready task has higher priority
            # hint: compare by first typecasting the tasks to tuple() first
            #   and compare them as tuples.  The tuples are defined in
            #   the __iter__() method of the Task class based on policy.
            # if next ready is not higher priority, redispatch current task.
            # otherwise,
            # - swap out current running (by calling preempt method)
        # task completed or swapped out
        # pick next task from ready queue to dispatch, if one exists.
```

```
def testPScheduler(tasks, policy):
   # this is same as before, but instantiate the preemptive scheduler.
   nClocks = 20
   scheduler = PScheduler(nClocks, policy)
   # the rest is the same as before
   for t in tasks:
      scheduler.addTask(t)
   for clock in scheduler.clockGen():
      pass
   print('preemptive %s: %s' % (scheduler.policy, scheduler.getSchedule()))

if __name__ == '__main__':
   tasks = [Task(*i) for i in [('A', 0, 7), ('B', 2, 4), ('C', 4, 1), ('D',
5, 4)]]
   print('tasks = %s' % tasks)
   for policy in ['SJF', 'FCFS', 'RR']:
         tasks = [Task(*i) for i in [('A', 0, 7), ('B', 2, 4), ('C', 4, 1),
('D', 5, 4)]]
         testPScheduler(tasks, policy)
```

Your output would look like

```
tasks = [Task('A', 0, 7), Task('B', 2, 4), Task('C', 4, 1), Task('D', 5, 4)]
preemptive SJF: A-A-B-B-C-B-B-D-D-D-D-A-A-A-A-A-None-None-None-None
preemptive FCFS: A-A-A-A-A-A-A-B-B-B-B-C-D-D-D-D-None-None-None-None
preemptie RR: A-A-B-A-B-C-A-D-B-A-D-B-A-D-A-D-None-None-None-None
```

2.5 Add Statistics (10 points)

Implement the following methods to the nonpreemptive scheduler code (and the preemptive one will automatically get the same code due to inheritance).

```
   def getThroughput(self):
      # throughput is the number of processes completed per unit time.
      # returns a tuple for (number of done processes, number of clocks)

   def getWaitTime(self):
      # returns a tuple for (total wait time of processes, #processes)

   def getTurnaroundTime(self):
      # returns a tuple for (total turnaround times, #processes)
```

Combine the nonpreemptive and preemptive schedulers into the same test bench and print out the statistics, so that the output looks like

```
$ python3 hw6both.py
tasks = [Task('A', 0, 7), Task('B', 2, 4), Task('C', 4, 1), Task('D', 5, 4)]
nonpreemptive SJF: A-A-A-A-A-A-A-C-B-B-B-B-D-D-D-D-None-None-None-None
```

```
  thruput = (4, 16) = 0.25, waittimes = (16, 4) = 4.00, turnaroundtime = (32,
4) = 8.00
preemptive SJF: A-A-B-B-C-B-B-D-D-D-D-A-A-A-A-A-None-None-None-None
  thruput = (4, 16) = 0.25, waittimes = (12, 4) = 3.00, turnaroundtime = (28,
4) = 7.00
nonpreemptive FCFS: A-A-A-A-A-A-A-B-B-B-B-C-D-D-D-D-None-None-None-None
  thruput = (4, 16) = 0.25, waittimes = (19, 4) = 4.75, turnaroundtime = (35,
4) = 8.75
preemptive FCFS: A-A-A-A-A-A-A-B-B-B-B-C-D-D-D-D-None-None-None-None
  thruput = (4, 16) = 0.25, waittimes = (19, 4) = 4.75, turnaroundtime = (35,
4) = 8.75
nonpreemptive RR: A-A-A-A-A-A-A-B-B-B-B-C-D-D-D-D-None-None-None-None
  thruput = (4, 16) = 0.25, waittimes = (19, 4) = 4.75, turnaroundtime = (35,
4) = 8.75
preemptive RR: A-A-B-A-B-C-A-D-B-A-D-B-A-D-A-D-None-None-None-None
  thruput = (4, 16) = 0.25, waittimes = (22, 4) = 5.50, turnaroundtime = (38,
4) = 9.50
```