CS 342302 Operating Systems

Fall Semester 2021

Prof. Pai H. Chou

Weekly Review 7

The questions here serve the purpose of reviewing concepts from the lecture, and expect the concepts to be tested on the midterm and final. However, they are by no means exhaustive. Anything covered in the lecture and projects can be tested.

1. Definitions and Short Answers - CPU Scheduling

1. What are **four cases** when a **preemptive** CPU scheduler can take control?
2. What is the difference between a **scheduler** and a **dispatcher**?
3. What are the **two cases** when a **nonpreemptive** CPU scheduler can take control?
4. What kind of problem can be caused by preemptive scheduling that is not a problem for nonpreemptive?
5. Does a **kernel** have to be designed to be preemptive in order to support preemptive scheduling of **user** processes? Or can a nonpreemptive kernel also support preemptive scheduling of user processes?
6. What is the definition of **CPU utilization**?
	1. What is its range?
	2. What is a practical utilization level?
	3. What level is considered heavy utilization?
7. What is the definition of **throughput**?
8. What is the difference between **turnaround time** and **response time**?
9. What is the definition of **waiting time**?
10. A scheduling may have the objectives to maximize or minimize which of the following criteria?
	1. CPU utilization
	2. turnaround time
	3. throughput
	4. response time
	5. waiting time
11. How does FCFS algorithm schedule processes?
	1. What is an advantage with FCFS?
	2. What are the two disadvantages?
12. Consider the Shortest-Job First (SJF) algorithm:
	1. What does "shortest job" refer to? Does it refer to the job's total length?
	2. SJF is optimal for what criterion?
	3. Why can't **true SJF** be implemented? How can it be **approximated** in practice?
	4. What is the difference between **preemptive** and **nonpreemptive** versions of SFJ?
13. Given a job mix:

| Process | burst time | arrival time |
| --- | --- | --- |
| P1 | 7 | 0 |
| P2 | 4 | 2 |
| P3 | 1 | 4 |
| P4 | 4 | 5 |

1. Draw the Gantt chart for preemptive SJF

| P1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| P3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| P4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| t | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

1. What are the **response times** of P1, P2, P3, and P4?
2. What is the **total waiting** time of the four processes?
3. What is the **average waiting time**?
	1. Draw the Gantt chart for nonpreemptive SJF

| P1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| P3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| P4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| t | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

1. What are the **response times** of P1, P2, P3, and P4?
2. What is the **total waiting** time of the four processes?
3. What is the **average waiting time**?
4. Consider round robin (RR) scheduling
5. Does it assume preemption or no preemption?
6. How does RR algorithm schedule tasks?
7. What is the effect of a **long time quantum**? It becomes similar to which other scheduling policy?
8. What is the effect of a **short time quantum**?
9. Given the job mix and a time quantum of 4,

|  | burst time | arrival time |
| --- | --- | --- |
| P1 | 6 | 0 |
| P2 | 3 | 1 |
| P3 | 3 | 2 |

Draw the Gantt chart for Round Robin scheduling

| P1 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P2 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| P3 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| t | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

1. What are the response times of P1, P2, and P3?
2. What is the total waiting time of the three processes?
3. Most scheduling algorithms can be expressed as a combination of priority-scheduling and preemption (or not). What is the priority scheme and preemption option for the following algorithms?
	1. FCFS
	2. SJF
4. What is the meaning of **starvation**? What can cause starvation, and what is a possible solution?
5. \What is the difference between **multilevel queue** scheduling and **multilevel feedback queue** scheduling?
6. What is the scope of contention for
	1. user-level (many-to-one and many-to-many) thread scheduling?
	2. (one-to-one) kernel thread scheduling?
7. For multiiprocessor scheduling, what is **asymmetric** vs. **symmetric** multiprocessing?
	1. Which one does scheduling centrally on one processor, and which one lets each processor schedule its own processes?
	2. What are two scheduling options in SMP, and which option needs additional mechanisms? Which one is more common?
8. What are two ways two interpret "load balanced"?
9. What are two kinds of migration during load balancing?
10. What is **processor affinity** of a process?
11. What are two kinds of affinity policies an OS can set on process migration?
12. In a NUMA, what are the processors with **high processor affinity** for a process?
13. What is the difference between **heterogeneous multiprocessing** (HMP) and asymmetric multiprocessing?
14. What is a difference between how a **soft real time** system and a **hard real time** system in terms of treatment of real-time tasks?
15. What does **event latency** refer to? From the time an event occurs to \_\_\_\_?
16. the **interrupt latency**
	1. is defined to be the amount of time from an arrival of interrupt to \_\_\_?
	2. for real-time systems, it is not enough to just minimize the interrupt latency but it must also be \_\_\_?
17. the **dispatch latency**
	1. is defined to be the amount of time for the dispatcher to \_\_\_
	2. it is best minimized through \_\_\_
	3. dispatch latency is further decomposed into **conflicts** and **dispatch** times. The conflicts time consists of time due to \_\_\_ and \_\_\_
18. What is the difference between **online** scheduling and **offline** scheduling?
19. Do both need to do schedulability analysis?
20. What does **admission control** mean?
21. What happens when an online scheduler is asked to schedule another real-time task but won't be able to guarantee meeting its timing constraints?
22. How is a periodic task defined?
	1. it can have up to four variables. What are they?
23. What can an aperiodic task be? Can it repeat? if so, what would be some condition?
24. In **rate monotonic** (RM) scheduling,
	1. Are the tasks periodic or aperiodic?
	2. What is the deadline defined to be?
	3. Is rate monotonic preemptive or not?
	4. Is rate monotonic fixed or dynamic priority?
	5. How does rate monotonic define the priority of a task?
	6. Does the rate-monotonic priority of a task depend on the task's execution time?
25. In earliest deadline first (EDF) scheduling,
	1. Can the tasks be periodic? aperiodic?
	2. Is EDF preemptive or nonpreemptive?
	3. Is EDF static priority or dynamic priority?
	4. How does EDF define the priority of a task?
26. What is the schedulability condition for rate monotonic? Is it a necessary or a sufficient condition or both?
27. What is the schedulability condition for EDF? Is it a necessary or sufficient condition or both?

2. 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](https://drive.google.com/open?id=1VUds9s1P835jePbK_HDIdl0hED45hfb4)” ----------

class FIFO:

 def \_\_init\_\_(self, initList=[]):

 self.A = list(initList)

 def get(self): # remove element and return itse value

 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

 return iter(self.A) # use list's standard iterator

 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, you need an implementation of a priority queue based on min-heap. It has the following API. You are urged to try implementing minheap.py yourself, but a [reference](https://drive.google.com/open?id=1eIOef4jM_781tzfYOoN4Ww5P8xMW0G99) version is also available.

--------- template for 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

[(4, 5), (6, 7), (5, 6)]

2.2 Task class

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. Again, you are urged to try implementing task.py yourself, but a [reference](https://drive.google.com/open?id=1dfo2FaDrDIBS9dSGesw5VjtXRCk3NLI8) version is also available.

--------- file “[task-template.py](https://drive.google.com/open?id=15QzOFOMxxOCyNiHXE7p5KDj6nal8i1H7)” : save and rename it as “task.py” ----------

classTask:

 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\_\_str\_\_(self):

 returnself.name

 def\_\_repr\_\_(self):

 *# note: the field names here are just examples.*

 *# if you name them differently, please change them accordingly.*

 returnself.\_\_class\_\_.\_\_name\_\_ + repr((self.name, self.release, self.cpuBurst))

 defsetPriorityScheme(self, scheme="SJF"):

 """

 the scheme can be "FCFS", "SJF", "RR", etc

 """

 \_KNOWN\_SCHEMES = ["FCFS", "SJF", "RR"]

 ifnotscheme in\_KNOWN\_SCHEMES:

 raise ValueError("unknown priority scheme %s: must be FCFS, SJF, RR")

 self.scheme = scheme

 def\_\_str\_\_(self):

 return self.name

 defdecrRemaining(self):

 *# call this method to decrement the remaining CPU burst time*

 defremainingTime(self):

 *# return the remaining CPU burst time*

 defdone(self):

 *# returns a boolean for if this task has remaining work to do*

 defsetCompletionTime(self, time):

 *# records the clock value when the task is completed*

 defturnaroundTime(self):

 *# returns the turnaround time of this task*

 defincrWaitTime(self):

 *# increments the amount of waiting time*

 defreleaseTime(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 = (self.release, ) *# example, but you may want a secondary*

 *# priority for tie-breaker. if so, just add them to the tuple.*

 elif(self.scheme == 'SJF'): *# shortest job first*

 t = *# tuple that defines priority in terms of "job length"*

 *# or is it really job length?*

 elif(self.scheme == 'RR'): *# round robin*

 t = *# define round robin priority if you use a MinHeap;*

 *# or you could just use a FIFO.*

 else:

 raiseValueError("Unknown scheme %s" % self.scheme)

 returniter(t)

2.3 Nonpreemptive Scheduler

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](https://drive.google.com/open?id=1VCG698hsgqVoUsAlcWidC6P18k-ZA_jN), 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)

 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

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](https://drive.google.com/open?id=18h3EgZSSFCoX8uM5VqR2N2fzVLxdW_-b)”, 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

preemptive 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

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. Download the schedstat.py to run, and the output looks like

$ python3 schedstat.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