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

PART-1

1. What are **four cases** when a **preemptive** CPU scheduler can take control?

A: 1. A process switches from RUNNING to WAITING (Waiting for IO).

2. A process switches from RUNNING to READY (Timer-interrupt occurs).

3. A process switches from WAITING to READY (IO completes)

4. A process terminates.

1. What is the difference between a **scheduler** and a **dispatcher**?

A: The scheduler decides when to dispatch which process while the. dispatcher is the mechanism used to perform the context switch.

1. What are the **two cases** when a **non-preemptive** CPU scheduler can take control?

A: A non-preemptive CPU scheduler can take control in the cases 1 and 4 from question 1. Also, the process may explicitly yield which is case 5 (Not discussed in the textbook).

1. What kind of problem can be caused by preemptive scheduling that is not a problem for nonpreemptive?

A: It may cause a race condition. A race condition causes an inconsistent state of shared data because of context switching before data modification. is completed. Example -> One process may overwrite other processes changes.

1. Does a **kernel** have to be designed to be preemptive to support preemptive scheduling of **user** processes? Or can a non-preemptive kernel also support preemptive scheduling of user processes?

A: A preemptive kernel is required. In a non-preemptive kernel, when a task makes a syscall, that task cannot be preempted until that syscall ends, thus, preemptive scheduling for user processes is not possible (e.g. preemptive SJF ). However, multiprogramming is still allowed.

1. What is the definition of **CPU utilization**?

A: CPU utilization is the % of time is NOT idle. Or the % of time the CPU is doing work.

* 1. What is its range? – (0-100%)
  2. What is a practical utilization level? – 40% utilization
  3. What level is considered heavy utilization? - 90% utilization.

1. What is the definition of **throughput**?

A: Throughput is defined as the number of processes completed per time unit over a long period of time. An average.

1. What is the difference between **turnaround time** and **response time**?

A: Turnaround time is the time a process takes since it is submitted until it completes. However, processes that are more event-driven do not care much about completion time since they may be idle for a long time (e.g., text editing), thus response time, or the time from submission to the first response produced is a better measure for these kind of programs.

\*\*\*Response time: Waiting time before the process can run for the first time.

1. What is the definition of **waiting time**?

A: Waiting time is the total time in the **ready** queue.

1. A scheduling may have the objectives to maximize or minimize which of the following criteria?
   1. CPU utilization - Maximize
   2. turnaround time – Minimize
   3. throughput - Maximize
   4. response time - Minimize
   5. waiting time - Minimize
2. How does FCFS algorithm schedule processes?
   1. What is an advantage with FCFS?

A: It has a simple FIFO structure which is "fair" to some extent.

* 1. What are the two disadvantages?

A: First, CPU-bound and I/O-bound processes don't mix well. I/O bound processes (short) wait a long time for CPU-bound processes (long) to finish. This minimizes CPU utilization. Second, FCFS is effectively non-preemptive and thus processes must run until completion before other processes may use the CPU. This is bad for interactive systems.

1. Consider the Shortest-Job First (SJF) algorithm:
   1. What does "shortest job" refer to? Does it refer to the job's total length?

A: Shortest job refers to the job with the shortest next CPU burst, not the shortest total length.

* 1. SJF is optimal for what criterion?

A: SJF is optimal for the minimum waiting time criterion. However, this is assuming we exactly know the next CPU burst which in practice is not possible to know exactly.

* 1. Why can't **true SJF** be implemented? How can it be **approximated** in practice?

A: Because in practice we don't exactly know the length of the next CPU burst, it is an undecidable problem. Thus, in practice it is approximated using the history of previous CPU bursts.

* 1. What is the difference between **preemptive** and **non-preemptive** versions of SFJ?

A: The difference lies in the action to take when a new job arrives. **Preemptive** SFJ preempts the current process if the new job has a shorter remaining burst. **Non-preemptive** SFJ runs a process until completion regardless of if a new job comes in with a shorter remaining burst.

1. 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 non-preemptive SJF

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | (7) | | | | | | |  |  |  |  |  |  |  |  |  |
| P2 |  |  | 5 | | | | | 1 | (4) | | | |  |  |  |  |
| P3 |  |  |  |  | 3 | | | (1) | ) |  |  |  |  |  |  |  |
| P4 |  |  |  |  |  | 2 | | 1 | 4 | | | | (4) | | | |
| 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?

A: RtP1 = 0, RtP2 = 6, RtP3 = 3, RtP4 = 7

1. What is the **total waiting** time of the four processes?

A: WT = 16

1. What is the **average waiting time**?

A: AWT = WT/PN = 16/4

|  |  |  |
| --- | --- | --- |
| 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 | (2/7) | | 2 | | 1 | 2 | | 4 | | | | (7/7) | | | | |
| P2 |  |  | (2/4) | | 1 | (4/4) | |  |  |  |  |  |  |  |  |  |
| P3 |  |  |  |  | (1/1) |  |  |  |  |  |  |  |  |  |  |  |
| P4 |  |  |  |  |  | 2 | | (4/4) | | | |  |  |  |  |  |
| 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?

A: RtP1 = 0, RtP2 = 0, RtP3 = 0, RtP4 = 2

1. What is the **total waiting** time of the four processes?

A: WT = 12

1. What is the **average waiting time**?

A: AWT = 12/4 = 3

PART-2

1. Consider round robin (RR) scheduling
2. Does it assume preemption or no preemption?

A: RR assumes preemption. If we assume no preemption, then we have

FCFS.

1. How does RR algorithm schedule tasks?

A: The RR algorithm schedules tasks using the concept of time quantum aka time slice (10-100 ms). After a TQ has elapsed a timer interrupt occurs, preempts the process, and puts it back to ready queue. Finally, the next process gets to run.

1. What is the effect of a **long-time quantum**? It becomes like which other scheduling policy?

A: The effect of a long-time quantum is running the processes as per their order of arrival. It then behaves like FCFS.

1. What is the effect of a **short-time quantum**?

A: It increases context-switching overhead.

1. 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 | (4/6) | | | |  | |  | |  | |  | |  | | (2/6) | |  |
| P2 |  |  |  |  | 3/3 | | | | |  | |  | |  |  |  |  |
| P3 |  |  |  |  |  |  | |  | | 3/3 | | | | |  |  |  |
| 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?

A: RtP1 = 0, RtP2 = 3, RtP3 = 5

1. What is the total waiting time of the three processes?

A: WT = 13

1. 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 – No preemption. Priority scheme -> Equal priority.
   2. SJF - Preemption. Priority scheme -> Inverse of the length of the next CPU burst.
2. What is the meaning of **starvation**? What can cause starvation, and what is a possible solution?

A: Starvation is an indefinite blocking phenomenon where a process is ready but is not scheduled to execute for a long time. A possible solution is aging where a process increases its priority after a certain time frame has passed.

1. What is the difference between **multilevel queue** scheduling and **multilevel feedback queue** scheduling?

A: In multilevel feedback queue scheduling the processes can move between queues whereas in multilevel queues processes stay in each queue.

1. What is the scope of contention for
   1. user-level (many-to-one and many-to-many) thread scheduling?

A: Process-contention scope – User-level scheduling competition is within process.

* 1. (one-to-one) kernel thread scheduling?

A: System-contention scope – Competition among all kernel threads in the system.

1. For multiprocessor 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?

A: Asymmetric multiprocessing does scheduling centrally on one processor while symmetric multiprocessing lets each processor schedule its own processes.

* 1. What are two scheduling options in SMP, and which option needs additional mechanisms? Which one is more common?

A: First option – Each processor has its own ready queue.

Second option – All processors share ready queue. This is the most common option however some synchronization is required.

1. What are two ways two interpret "load balanced"?

A: The first way to interpret “load balanced” is when private queues have the same number of threads, however, threads can be different. (i.e., CPU-bound vs IO-bound) and thus in practice this definition may not hold. The second way to interpret "load balanced" is equal distribution of thread priorities across all queues. This means all queues have the same "mix" of priorities.

1. What are two kinds of migration during load balancing?

A: **Push migration** where an overloaded CPU pushes the load to a less busy or idle CPU and **Pull migration** where an idle CPU pulls waiting task from a busy CPU.

PART-3

1. What is **processor affinity** of a process?

A: Also known as “cache affinity, enables the binding and unbinding of a process or a thread to a CPU or range of CPU’s so that the process or thread will **ONLY** execute on the designated CPU or range of CPU’s.

Also defined as the “closeness” between a process and its processor.

1. What are two kinds of affinity policies an OS can set on process migration?

A: Soft affinity and hard affinity. In soft affinity, the processor tries. to keep the process on same processor (high affinity) but allows migration between processors (load balancing). In addition, in soft affinity all possible cases for migration are considered (e.g., far away processor).

In hard affinity, process migration is restricted only to a subset of processors. Example: An idle processor far away wouldn’t be considered for migration in hard affinity.

1. In a NUMA, what are the processors with **high processor affinity** for a process?

A: In a NUMA, processes with high affinity are those whose memory is allocated to the CPU.

1. What is the difference between **heterogeneous multiprocessing** (HMP) and asymmetric multiprocessing?

A: They are completely different concepts. Asymmetric multiprocessing is a scheduling mechanism. On the other hand, heterogeneous multiprocessing refers to the different power vs. performance classes of processors.

1. 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?

A: A soft real-time provides preferences but does not guarantee in scheduling real-time processes (e.g., multimedia streaming). A hard real-time system provides guarantees in meeting deadlines once the tasks are accepted.

1. What does **event latency** refer to? From the time an event occurs to \_\_\_\_?

A: Amount of time from the event occurring to the time the event is serviced.

1. The **interrupt latency**
   1. is defined to be the amount of time from an arrival of interrupt to the s**tart** of the ISR execution.
   2. for real-time systems, it is not enough to just minimize the interrupt latency, but it must also be **bounded**.
2. the **dispatch latency**
   1. is defined to be the amount of time for the dispatcher to switch process.
   2. it is best minimized through preemptive kernel.
   3. dispatch latency is further decomposed into **conflicts** and **dispatch** times. The conflicts time consists of time due to **preemption** and **release of resources**.
3. What is the difference between **online** scheduling and **offline** scheduling?
4. Do both need to do schedulability analysis?

A: Both may use it, however in online scheduling, schedulability analysis is done as the system runs. On the other hand, offline scheduling performs it before running,

1. What does **admission control** mean?

A: Admission control refers to the acceptance or rejection of tasks based on the ability of meeting time constraints.

1. 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?

A: The task is rejected as per admission control. Real-time constraints must be met.

1. How is a periodic task defined?

A: Periodic task means the task is recurring at a regular interval.

* 1. it can have up to four variables. What are they?

A: Execution time t, deadline d, period p and may have release time after the start of period.

1. What can an aperiodic task be? Can it repeat? if so, what would be some condition?

A: Task is nonrecurring. It may repeat after some minimum separation.

1. In **rate monotonic** (RM) scheduling,
   1. Are the tasks periodic or aperiodic? - Periodic
   2. What is the deadline defined to be? - d = end of period
   3. Is rate monotonic preemptive or not? – Preemptive
   4. Is rate monotonic fixed or dynamic priority? – Fixed priority
   5. How does rate monotonic define the priority of a task? – Based on period length.
   6. Does the rate-monotonic priority of a task depend on the task's execution time? – It depends on the length of the period.
2. In earliest deadline first (EDF) scheduling,
   1. Can the tasks be periodic? aperiodic? - Both
   2. Is EDF preemptive or nonpreemptive? - Preemptive
   3. Is EDF static priority or dynamic priority? – Dynamic priority
   4. How does EDF define the priority of a task? – Based on time of deadline
3. What is the schedulability condition for rate monotonic? Is it a necessary or a sufficient condition or both?

A: Sufficient condition only -> Utilization on uniprocessor <= Ln(2)

1. What is the schedulability condition for EDF? Is it a necessary or sufficient condition or both?

A: Utilization on uniprocessor <= 100%. – Both necessary and sufficient condition.

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