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

Assignment 7

Due Date: Sunday, November 5, 2017, 11:59pm
Up to one-day late submission without penalty
Up to one-week late submission with 20% penalty
Submit electronically on iLMS

What to submit: One zip file named <studentID>-hw7.zip (replace <studentID> with your own student ID). It should contain four files:

- one PDF file named **hw7.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 2.1: **graph.py**, **typescript1**
  - Section 2.2, 2.3, 2.4: **banker.py**, **typescript2** (showing output of TestUtility(), TestConstructor(), TestSafety(), and TestRequest())
  - Section 2.5: **detect.py**, **typescript5**

1. [40 points] Problem Set

1. [20 points] 7.4 In Section 7.4.4, we describe a situation in which we prevent deadlock by ensuring that all locks are acquired in a certain order. However, we also point out that deadlock is possible in this situation if two threads simultaneously invoke the `transaction()` function. Fix the `transaction()` function to prevent deadlocks.

```
void transaction(Account from, Account to, double amount) {
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

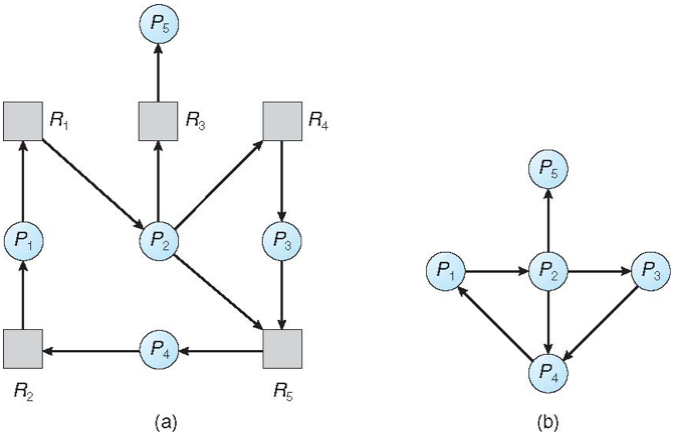
2. [20 points] 7.7 Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock free.

2. [60 points] Programming Exercise

In this programming exercise, you are to implement a number of deadlock detection and avoidance algorithms.

2.1 [10 points] Cycle Detection in Graphs

Cycle detection can be used to detect deadlocks. Cycles are found in graphs. A (system) resource-allocation graph (RAG) is a directed graph for capturing the dependencies of processes on resources. An example of a RAG is shown in the following figure (a):



For a special case of a RAG where there is exactly one instance of resource per type, it can be transformed into a wait-for graph (WFG), which can use a conventional directed graph to capture just the processes but not resources. It is shown in figure (b) above.

A conventional graph $G(V, E)$ can be represented in adjacency-list format, which has space complexity closer to $O(V+E)$ for sparse graphs. In Python, a (directed) graph can be represented more conveniently using a dictionary, where a dictionary keeps track of key-value pairs often in the form of a hash table. For example, consider the graph from Fig. (b) above. It can be represented in Python as

```
G = {'P1': ['P2'], 'P2': ['P4', 'P5', 'P3'], 'P3': ['P4'], 'P4': ['P1'], 'P5': []}
```

To find the list of neighbors of a vertex `v`, simply do `G[v]`. For example, `G['P2']` gives the value `['P4', 'P5', 'P3']`. However, it is more convenient to wrap the adjacency list inside a class so that more attributes can be associated with the graph. One way to do this is

```
class Graph:
   def __init__(self, G):
      self.G = G
      self.vertices = list(G.keys())
   def Adj(self, v):
      # return the adjacency list
      for i in self.G[v]:
         yield i
```

```
    def V(self):
        for i in self.vertices:
            yield i
```

Cycle detection can be done by depth-first search (DFS), among many other algorithms. A generic version of DFS based on the CLRS textbook (Cormen, Leiserson, Rivest, and Stein) is given below (assuming you have the Graph data structure above). You may download the graph-template.py file and rename it `graph.py`. It contains the `Graph` class and the following DFS code.

```
WHITE = 'white'
GRAY =  'gray'
BLACK = 'black'

def DFS(G):
    G.color = {} # color, which is WHITE, GRAY, or BLACK
    G.pred = {}  # the predecessor
    for u in G.V():
        G.color[u] = WHITE
        G.pred[u] = None
    for u in G.V():
        if G.color[u] == WHITE:
            DFSVisit(G, u)

def DFSVisit(G, u):
    G.color[u] = GRAY
    for v in G.Adj(u):
        if G.color[v] == WHITE:
            G.pred[v] = u
            DFSVisit(G, v)
    G.color[u] = BLACK
```

DFS can be used for cycle detection, but it does not do it automatically. You will need to know the right place to make the modification to detect a cycle. The two graphs in the above figures (a) and (b) have been input to the test case of the .py file.

What to turn in: `graph.py`, typescript1 showing the cycle has been detected or printed, or the empty list if there is no cycle.

2.2 Banker's Algorithm

The Banker's Algorithm by Dijkstra is a deadlock avoidance algorithm during resource allocation. To implement this in Python, it is easier to package things in a class and call a set of utility functions. You can download the banker-template.py file and rename it `banker.py`. There are two parts: the constructor and utility functions, Safety core algorithm, and the request processing.

2.1 [10 points] Construtor and Utility Functions

The helper functions are

```
def sumColumn(M, col): #  M is a row major matrix; col is the column index.
#       returns the scalar sum of the values in the column.
   return sum(list(map(lambda x: x[col], M)))
#  it is the same as
#   tot = 0
#   for row in M:
#     tot += row[col]
#   return tot

def IncrVec(A, B):
#   helper function to do A += B as vector, assuming len(A) == len(B)

def DecrVec(A, B):
#  vector A -= B, assuming len(A) == len(B)

def GtVec(A, B):
#  vector A[i]>B[i]. true if one or more pairs are true.

def LeVec(A, B):
# vector A[i] <= B[i]. true if ALL pairs are true.
```

The code for `sumColumn()` is given to you, but you need to write the other four utility functions. `GtVec()` and `LeVec()` are the “greater-than” and “less-than-or-equal-to” functions comparing two vectors (represented as lists), respectively. Unlike the built-in > and <= operators on lists and tuples, which perform lexicographical comparison, what is required here is the pairwise comparison. Note the subtle point that `GtVec()` is *disjunctive* while `LeVec()` is *conjunctive*.

```
class Banker:
  def __init__(self, alloc, max, totalRsrc):
    '''
       constructor for Banker class.
       alloc is a vector of number of instances of m resource types.
       max is a  matrix for max #of instances that the process may request.
       totalRsrc is vector of total #of instances of each type of resource
    '''
    self.Allocation = alloc
    self.TotalResources = totalRsrc
    self.n = len(alloc) # number of processes
    self.m = len(totalRsrc)  # number of resources

    # the following if-max allows the deadlock detection algorithm
    # to be able to subclass without max (since it doesn't need it)
    if max is not None:
```

```
        self.Max = max
        self.Need = []# your code here to initialize the Need matrix.

    self.Available = [] # your code here to compute Available vector.
        # hint: involves TotalResources and sumColumn() function,

    # a boolean flag to indicate whether in Safety() function you want to
    # print the traced output. by default False, but can be set to True.
    self.traceSafety = False
```

Modify the testbench to call just the `TestUtility()` and `TestConstructor()` functions (provided) and make sure your code behaves correctly before moving on to the next subsections. You should output that looks like this:

```
Testing Utility Functions:
A = [1, 2, 1], B = [1, 0, 2],
A += B is [2, 2, 3], expect [2, 2, 3]
A -= B is [1, 2, 1], expect [1, 2, 1]
A > B is True, expect True; A <= B is False, expect False
A = [1, 2, 3], B = [2, 2, 4],
A += B is [3, 4, 7], expect [3, 4, 7]
A -= B is [1, 2, 3], expect [1, 2, 3]
A > B is False, expect False; A <= B is True, expect True
A = [2, 3, 3], B = [2, 3, 3],
A += B is [4, 6, 6], expect [4, 6, 6]
A -= B is [2, 3, 3], expect [2, 3, 3]
A > B is False, expect False; A <= B is True, expect True
b.Available=[3, 3, 2], expect ([3, 3, 2],)b.Need=[[7, 4, 3], [1, 2, 2], [6,
0, 0], [0, 1, 1], [4, 3, 1]], expect [[7, 4, 3], [1, 2, 2], [6, 0, 0], [0, 1,
1], [4, 3, 1]]
```

2.2.2 [10 points] Safety Algorithm (week 8 slide 37)

The Safety algorithm finds a safe sequence of executing a set of processes such that the system never enters an unsafe state, or else it reports that such a safe sequence does not exist. It is implemented as a method in the Banker class.

```
   def Safety(self):
      if self.traceSafety: print('Need=%s, Available=%s' % (self.Need,
self.Available))
      # step 1
      Sequence = []  # use this list to save the safe sequence
      Finish = [False for i in range(self.n)]
      Work =  [] # your code to initialize Work vector
      # step 2
      for _ in range(self.n):
          for i in range(self.n):
              if self.traceSafety: print('i=%d, ' % i, end="")
              # follow the pseudocode on slide 37
```

```
            # may need to print
            #
            # compare Need[i] with Work.
            # - hint: you may use LecVec(A, B) for A <= B:
            #
            # step 3
            # update bookkeeping: Work, Finish, and add to sequence
            # Hint: you may want to use IncrVec() for Work += Allocation
            #
    # step 4.  return the sequence if there is one, or None if not.
```

Run the `TestSafety()` function (provided) in the testbench. Note that we include a `traceSafety` flag, which will print the intermediate values as the code runs. You can expect to get output like the following:

```
i=0, (Need[0]=[7, 4, 3]) <= (Work=[3, 3, 2]) False, P0 must wait
i=1, (Need[1]=[1, 2, 2]) <= (Work=[3, 3, 2]) True, append P1
i=2, (Need[2]=[6, 0, 0]) <= (Work=[5, 3, 2]) False, P2 must wait
i=3, (Need[3]=[0, 1, 1]) <= (Work=[5, 3, 2]) True, append P3
i=4, (Need[4]=[4, 3, 1]) <= (Work=[7, 4, 3]) True, append P4
i=0, (Need[0]=[7, 4, 3]) <= (Work=[7, 4, 5]) True, append P0
i=1, Finish[1] True, skipping
i=2, (Need[2]=[6, 0, 0]) <= (Work=[7, 5, 5]) True, append P2
s is [1, 3, 4, 0, 2]
```

2.2.2 [10 points] Resource-Request Algorithm (week 8 slide 47)

The Resource-Request Algorithm is the outer code of the Banker's algorithm that calls the Safety algorithm above to decide how to respond to the request by the process. Add the following method named Request() and a utility method named Release() to your Banker class:

```
    def Request(self, i, rqst):  # slide 47
        '''
           called with the requesting process i and the resource vector
           for how many instances of each resource to request.
           the rqst is a vector of m length.
        '''
        # step 1
        # hint: use GtVec of LeVec to compare request vector with Need[i]
        # raise an exception if overclaimed
        #
        # step 2
        # in case of wait, simply return None
        #
        # step 3
        # pretend to allocate requested resource:
        # save snapshot of Available, Allocation, and Need
        # update Available, Allocation, and Need
        # call Safety()
        # if a safe sequence exists, return it.
```

```
        # otherwise, restore saved snapshot and return None
    def Release(self, i):
        '''
            need this function to release the resources allocated to P_i
            after it has finished execution.
        '''
        # hint: update self.Available, self.Allocation, and self.Need.
        # hint: you may want to call utility functions IncrVec
        # hint: but in which order? who goes first, last, or don't care?
```

Run the TestRequest() code using the return values of the TestSafety() as provided in the template code. You can expect to get the output like this for this part:

```
Found safe sequence [1, 3, 4, 0, 2]
P1 allocated [2, 0, 0], requesting [1, 0, 2],
 P1 releasing, available=[5, 3, 2]
P3 allocated [2, 1, 1], requesting [0, 1, 1],
 P3 releasing, available=[7, 4, 3]
P4 allocated [0, 0, 2], requesting [3, 3, 0],
 P4 releasing, available=[7, 4, 5]
P0 allocated [0, 1, 0], requesting [0, 2, 0],
 P0 releasing, available=[7, 5, 5]
P2 allocated [3, 0, 2], requesting [3, 0, 0],
 P2 releasing, available=[10, 5, 7]
```

2.3 [20 points] Deadlock Detection Algorithm (week 8 slide 53)

Write the deadlock detection algorithm. It is similar to the Banker's algorithm, and code reuse including the utility functions and most of the constructor is possible, if you make minor adjustments. The differences are

- there is no Max and Need; instead, it has requests. => we pass None to the superclass's constructor, and it will skip capturing Max and computing Need.
- it detects deadlock from the current allocation and request matrix, rather than checking existence of a safe sequence.

Download detect-template.py and rename it `detect.py`. It looks like the following:

```
# Deadlock Detection, similar to Banker's

from banker import Banker, sumColumn, IncrVec, DecrVec, GtVec

class DeadlockDetector(Banker):
    def __init__(self, alloc, totalRsrc):
        Banker.__init__(self, alloc, None, totalRsrc)

    def detect(self, Request):  # see week 8 slide 53
        '''detect deadlock with the request matrix'''
        # 1(a) initialize Work = a copy of Available
        # 1(b) Finish[i] = (Allocation[i] == [0, ...0])
```

```
        # optionally, you can keep a Sequence list
        for _ in range(self.n):
            for i in range(self.n):
                # Step 2: similar to safety algorithm
                #   if there is an i such that (Finish[i] == False)
                #   and Request_i <= Work, (hint: LeVec() could help) then
                #   Step 3:
                #     Work += Allocation[i]
                #     Finish[i] = True
                #        continue Step 2
        # Step 4: either done iterating or (no such i exists)
        #    Finish vector indicates deadlocked processes.
        #    if all True then no deadlock.
```

The testbench is included in the template file. There are two cases: one without deadlock and one with deadlock, both taken from the textbook. You can expect to see the following output:

```
Finish=[False, False, False, False, False]
i=0, (Request[0]=[0, 0, 0]) <= (Work=[0, 0, 0]) True, append P0
  (+Allocation[0]=[0, 1, 0])=> Work=[0, 1, 0], Finish=[True, False, False,
False, False]
i=1, (Request[1]=[2, 0, 2]) <= (Work=[0, 1, 0]) False, P1 must wait
i=2, (Request[2]=[0, 0, 0]) <= (Work=[0, 1, 0]) True, append P2
  (+Allocation[2]=[3, 0, 3])=> Work=[3, 1, 3], Finish=[True, False, True,
False, False]
i=3, (Request[3]=[1, 0, 0]) <= (Work=[3, 1, 3]) True, append P3
  (+Allocation[3]=[2, 1, 1])=> Work=[5, 2, 4], Finish=[True, False, True,
True, False]
i=4, (Request[4]=[0, 0, 2]) <= (Work=[5, 2, 4]) True, append P4
  (+Allocation[4]=[0, 0, 2])=> Work=[5, 2, 6], Finish=[True, False, True,
True, True]
i=0, Finish[0] is True, skipping
i=1, (Request[1]=[2, 0, 2]) <= (Work=[5, 2, 6]) True, append P1
  (+Allocation[1]=[2, 0, 0])=> Work=[7, 2, 6], Finish=[True, True, True,
True, True]
sequence = [0, 2, 3, 4, 1]
Finish=[False, False, False, False, False]
i=0, (Request[0]=[0, 0, 0]) <= (Work=[0, 0, 0]) True, append P0
  (+Allocation[0]=[0, 1, 0])=> Work=[0, 1, 0], Finish=[True, False, False,
False, False]
i=1, (Request[1]=[2, 0, 2]) <= (Work=[0, 1, 0]) False, P1 must wait
i=2, (Request[2]=[0, 0, 1]) <= (Work=[0, 1, 0]) False, P2 must wait
i=3, (Request[3]=[1, 0, 0]) <= (Work=[0, 1, 0]) False, P3 must wait
i=4, (Request[4]=[0, 0, 2]) <= (Work=[0, 1, 0]) False, P4 must wait
i=0, Finish[0] is True, skipping
i=1, (Request[1]=[2, 0, 2]) <= (Work=[0, 1, 0]) False, P1 must wait
i=2, (Request[2]=[0, 0, 1]) <= (Work=[0, 1, 0]) False, P2 must wait
i=3, (Request[3]=[1, 0, 0]) <= (Work=[0, 1, 0]) False, P3 must wait
i=4, (Request[4]=[0, 0, 2]) <= (Work=[0, 1, 0]) False, P4 must wait
deadlock
```