CS 342302 Operating Systems

Fall Semester 2021

Prof. Pai H. Chou

Weekly Review 9

(Scope: Ch. 8)

* + Section 3.1: **graph.py**, **typescript1**
	+ Section 3.2: **banker.py**, **typescript2** (showing output of TestUtility(), TestConstructor(), TestSafety(), and TestRequest())
	+ Section 3.3: **detect.py**, **typescript3**

## 1. Definitions and Short Answers

1. Given two processes

| process1:  lock(mutex1)  lock(mutex2)  work()  unlock(mutex2)  unlock(mutex1) | process2:  lock(mutex2)  lock(mutex1)  work()  unlock(mutex1)  unlock(mutex2) |
| --- | --- |

* 1. Do they always deadlock?
	2. Do they sometimes deadlock? If so, describe one such condition
1. If two processes are not blocked but busy trying and keep failing to make progress, is it a deadlock?
2. What are the **four necessary conditions** of a deadlock?
3. If **mutual exclusion** is a necessary condition for a deadlock, then does it mean that
	1. If you use a mutex then you **always** have the possibility of getting a deadlock?
	2. If you use a **counting semaphore** of n > 1 then you won't get into a deadlock?
4. If hold-and-wait is a necessary condition for a deadlock, then does it mean that
	1. If you acquire multiple resources, then you always have the possibility of getting into a deadlock?
	2. If you acquire multiple resources but at most one at a time, then you won't get into a deadlock?
5. If no-preemption is a necessary condition for a deadlock, then does it mean that
	1. All preemptive process (or thread) schedulers can get into deadlocks?
	2. If a process P1 holds a mutex but is now blocked while waiting to acquire another resource, **preemption** in this case means to temporarily allow another process P1 to acquire that mutex, finish, and give the mutex back to P1?
6. If circular-wait is a necessary condition for a deadlock, then does it mean that
	1. Whenever you have a process P1 that is waiting to acquire an instance of a resource type R1 that is currently assigned to process P2 and P2 is waiting to acquire an instance of a resource type R2 that is currently assigned to process P1, then you have a deadlock?
7. A resource-allocation graph (RAG) can be used to model a system for deadlock analysis.
	1. Is a RAG a bipartite graph? If so, what are the two sets of vertices and what do they represent?
	2. Is a RAG a directed or undirected graph?
	3. How is a resource request represented in the RAG?
	4. How is a resource assignment represented in the RAG?
8. Give a RAG that contains a **cycle but does not have deadlock**.
9. In some RAG, having a cycle means a deadlock exists. Why would this be the case?
10. What is the meaning of deadlock **prevention**? What is its general approach?
11. What is difficult about **denying mutual exclusion** as a way of achieving deadlock prevention?
12. What are two ways of eliminating **hold-and-wait**?
13. What are disadvantages with the two ways of eliminating hold and wait above?
14. What are difficulties with deadlock prevention by **allowing resource preemption**?
15. What is a way of breaking **circular wait** as the 4th way of deadlock prevention?
16. For **deadlock avoidance** to work, what does each process have to declare?
17. Does deadlock avoidance ensure that the system …
	1. never gets a deadlock?
	2. never enters an unsafe state?
	3. always stays in a safe state?
18. What is a **claim** in a deadlock avoidance algorithm?
19. In the **resource allocation graph** (RAG) scheme of deadlock avoidance,
	1. Why does the RAG use only a claim edge but does NOT use an edge weight to indicate the max **number of instances** of a resource that the process may claim?
	2. What is the difference between a **claim edge** and a **request edge** in the resource allocation graph scheme of deadlock avoidance?
	3. When a request is granted, what happens to the **request edge**?
	4. When the resource is released, what happens to the **assignment edge**?
	5. What does a cycle mean? Under what condition is a request granted?
20. To use the Banker's algorithm for deadlock avoidance,
	1. Banker's algorithm uses the Safety Algorithm to find a safety sequence. If such a sequence is found, is it a **necessary** condition or a **sufficient** condition that the system is in a safe state?
	2. What are variables *m* and *n*?
	3. What do *Available*[*j*] and *Work*[*j*] represent?
	4. What does the variable int *Max*[*n*][*m*] represent? Where does this matrix get its values from?
	5. What does the variable int *Allocation*[*n*][*m*] represent?
	6. What is the meaning of variable int *Need*[*n*][*m*] and how does it get its values from?
	7. In Step 3, Process Pi is chosen because its worst-case requests can be fulfilled, so why is *Allocation*[*i*][*j*] added back into *Work*[*j*] instead of being subtracted from *Work*[*j*]?
21. In the Resource-Request Algorithm, which decides whether a request *Requesti* (by process Pi, of different resource types),
	1. If *Requesti*[:] ≤ *Needi*[:], does it mean that the request can be fulfilled? In other words, is it a **necessary** condition or a **sufficient** for granting the request?
	2. If the previous condition is a necessary condition, then what additional condition is needed in order to grant the request?
	3. If the request cannot be granted, what happens?
22. A **wait-for** graph is used for deadlock detection of single-instance resource types.
	1. What is the wait-for graph that corresponds to the RAG below?
	
	2. if the wait-for graph contains a cycle, does it mean there is definitely a deadlock or just possibility of a deadlock?
	3. if the processes are in a deadlock, does it mean there is definitely a cycle or possibility of a cycle?
	4. What is the complexity of cycle detection?
23. For deadlock detection of multi-instance resource types, an algorithm essentially the same as the Safety Algorithm is used to detect cycles.
	1. How can you tell if the processes are deadlocked after running the algorithm?
	2. How do you find all the processes that have the circular dependency in the deadlock?
24. In practice, is the deadlock detection algorithm invoked on every request? Why or why not?
25. What can an OS do after it detects a deadlock?
	1. Does it abort a process? If so, what are possible considerations?
	2. What does **roll back** mean? Can every process be rolled back? What needs to happen first before a roll back?
	3. Even if a system can recover from a deadlock, what problem may still happen to some unlucky process?

## 2. Problem Set

1. 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. 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.

## 3. Programming Exercise

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

### 3.1 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

 **return** iter(self.G[v])

 **def** V(self):

 **return** iter(self.vertices)

In case you did not know, iter() is a built-in function in Python that returns an iterator object that lets you pull out one element at a time. Several iterators may simultaneously exist on the same list, for example.

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](https://drive.google.com/file/d/1qU-gpJUmSRU93MLU9duWhLScT_1XBOmt/view) 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

 # you may add your own field for tracking cycles

 **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)

 # add your own code for cycle detection here!!

 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.

Deliverable: graph.py, typescript1 showing the cycle has been detected or printed, or the empty list if there is no cycle.

### 3.2 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](https://drive.google.com/file/d/1NNMEqsVOLFXaIPQ-aALCIjFIYjYpT-Ht/view) file and rename it banker.py. There are two parts: the constructor and utility functions, Safety core algorithm, and the request processing.

#### 3.2.1 Constructor 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.

 tot = 0

 for row in M:

 tot += row[col]

 return tot

def IncrVec(A, B):

# helper function for A += B as vector, assuming len(A) == len(B)

# your code here

def DecrVec(A, B):

# vector A -= B, assuming len(A) == len(B)

# your code here

def GtVec(A, B):

# vector A[i]>B[i]. true if one or more pairs true. (disjunctive)

# your code here

def LeVec(A, B):

# vector A[i] <= B[i]. true if ALL pairs are true. (conjunctive)

# your code here

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*** (i.e., true if ANY A[i] is > B[i]) while LeVec() is ***conjunctive*** (i.e., ALL of A[i] must be <= B[i]).

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 #instances the process may request.

 totalRsrc is vector of total #instances of ea. type of rsrc.

 '''

 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.

 # hint: involves TotalResources and sumColumn() function,

 # a boolean flag to indicate whether in Safety() you want to

 # print the traced output. by default False but can be = 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]]

#### 3.2.2 Safety Algorithm (week 9 (Chapter 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 LeVec(A, B) for A <= B:

 #

 # step 3

 # update Work, Finish, and add to sequence

 # Hint: use IncrVec() for Work += Allocation

 #

 # step 4. return the sequence if there is one, else None

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]

#### 3.2.3 Resource-Request Algorithm (week 9 (Chapter 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 w/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 w/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 rsrc 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: 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 Deadlock Detection Algorithm (week 9 slides 53-54)

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](https://drive.google.com/file/d/10fH5M-nsZu8t6pgyYbdpTVrArAi6rmJg/view) 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 9 **slides 53-54**

 '''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)

 # 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