CS4411 Operating Systems Homework 1 (Deadlock) Solutions Spring 2019

In this solution set, we always scan the *Need* and *Request* arrays from top to bottom to find the first process whose *Need/Request* can be met. After finding that process, instead of filling 0's to its entries to indicate it can take the needed resource and run, we will replace the entries with blanks. In this way, the intension could be clearer.

1. **[10 points]** Consider a three process system in which processes may request any of 12 drives. Suppose the allocation state given below. Show that the allocation state is unsafe. Will this system deadlock?

	Allocation	*Max*	Need	Available
P_0	5	10		12
P_1	2	4		
P_2	3	9		

Always provide your detailed calculation. Or you risk a lower grade.

Answer: First, we calculate the *Need* array:

	Allocation	*Max*	Need	Available
P_0	5	10	5	12
P_1	2	4	2	
P_2	3	9	6	

Keep in mind that we always use a top-down scan. Because P_0's $Need = 5 \leq Available = 12$, P_0 can run and returns its allocated 5, making the new $Available = 12 + 5 = 17$.

	Allocation	*Max*	Need	Available
P_0		10		17
P_1	2	4	2	
P_2	3	9	6	

Then, P_1 can run because its $Need = 2 \leq Available = 17$. After P_1 finishes its work, its $Allocation = 2$ is returned to $Available$, and the new $Available = 17 + 2 = 19$.

	Allocation	*Max*	Need	Available
P_0		10		19
P_1		4		
P_2	3	9	6	

Finally, because P_2's $Need = 3 \leq Available = 17$, P_2 can run. As a result, this system is in a safe state and $< P_0, P_1, P_2 >$ is a safe sequence. ■

2. **[10 points]** Consider the following snapshot of a system in which four resources A, B, C and D area available. The system contains a total of 6 instances of A, 4 of resource B, 4 of resource C, 2 resource D.

	Allocation				*Max*				*Need*				*Available*			
	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*
P_0	2	0	1	1	3	2	1	1					6	4	4	2
P_1	1	1	0	0	1	2	0	2								
P_2	1	0	1	0	3	2	1	0								
P_3	0	1	0	1	2	1	0	1								

Do the following problems using the banker's algorithm:

- Compute what each process might still request and fill this in under the column *Need*.
- Is the system in a safe state? Why or why not?
- Is the system deadlocked? Why or why not?
- If a request from P_3 arrives for $(2,1,0,0)$, can the request be granted immediately?

Always provide your detailed calculation. Or you risk a lower grade.

Answer: We first calculate the *Need* array as follows:

	Allocation				*Max*				*Need*				*Available*			
	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*
P_0	2	0	1	1	3	2	1	1	1	2	0	0	6	4	4	2
P_1	1	1	0	0	1	2	0	2	0	1	0	2				
P_2	1	0	1	0	3	2	1	0	2	2	0	0				
P_3	0	1	0	1	2	1	0	1	2	0	0	0				

Because P_0's $Need = [1,2,0,0] \le Available = [6,4,4,2]$, P_0 can run and return its $Allocation = [2,0,1,1]$ to *Available*. The new $Available = [6,4,4,2]+[2,0,1,1] = [8,4,5,3]$:

	Allocation				*Max*				*Need*				*Available*			
	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*
P_0					3	2	1	1					8	4	5	3
P_1	1	1	0	0	1	2	0	2	0	1	0	2				
P_2	1	0	1	0	3	2	1	0	2	2	0	0				
P_3	0	1	0	1	2	1	0	1	2	0	0	0				

Now we run P_1. After that P_1 returns its $Allocation = [1,1,0,0]$ to *Available*, and, hence $Available =$ old $Availavle = [8,4,5,3] + P_1's\ Allocation = [1,1,0,0] = [9,5,5,3]$.

	Allocation				*Max*				*Need*				*Available*			
	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*
P_0					3	2	1	1					9	5	5	3
P_1					1	2	0	2								
P_2	1	0	1	0	3	2	1	0	2	2	0	0				
P_3	0	1	0	1	2	1	0	1	2	0	0	0				

Now we can run P_2 followed by P_3. Consequently, the system is in a safe state. Because the deadlock state is in the unsafe area and because this system is in a safe state, this system is not deadlocked.

Now get back to the original situation:

	Allocation				*Max*				*Need*				*Available*			
	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*
P_0	2	0	1	1	3	2	1	1	1	2	0	0	6	4	4	2
P_1	1	1	0	0	1	2	0	2	0	1	0	2				
P_2	1	0	1	0	3	2	1	0	2	2	0	0				
P_3	0	1	0	1	2	1	0	1	2	0	0	0				

If a new request from P_3 arrives for $[2,1,0,0]$, this is a mistake because $P_3's\ Request = [2,1,0,0] \not\le P_3's\ Need = [2,0,0,0]$ and this request cannot be granted. See the Resource-Request algorithm for the details. ∎

3. **[10 points]** Let us revisit the dining philosophers problem again using banker's algorithm. Suppose there are only 3 philosophers P_0, P_1 and P_2 and 3 chopsticks C_0, C_1 and C_2. Initially, the system starts with the following:

	Allocation			*Max*			Need			*Available*		
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2
P_0	0	0	0	1	1	0				1	1	1
P_1	0	0	0	0	1	1						
P_2	0	0	0	1	0	1						

Again, each philosopher will request his left chopstick followed by his right chopstick. Therefore, philosopher P_i will do the following:

```
while (true) {
      Request(C_i);
      Request(C_{(i+1)%3});
            Eat;
      Release(C_{(i+1)%3});
      Release(C_i);
}
```

Note that philosophers may make their requests/releases concurrently; however, you may assume each request and release will be handled in a mutually exclusive way. Use banker's algorithm (*i.e.*, resource-request and safety) to allocate chopsticks so that each philosopher will east at least twice.

Always provide your detailed calculation. Or you risk a lower grade.

Answer: This is a problem for you to practice a sequence of requests and releases. Because of a simple deadlock due to all philosophers picking up their left chopsticks at the same time, we shall follow this execution sequence:

- P_0 requests left chopstick C_0
- P_1 requests left chopstick C_1
- P_2 requests left chopstick C_2
- P_0 requests right chopstick C_1
- P_1 requests right chopstick C_2
- P_2 requests right chopstick C_0

If banker's algorithm determines that a particular request cannot be made if doing so will make the system into an unsafe state, we need a queue to store these unsatisfied requests. Then, once a release is made, after updating the *Allocation*, *Need* and *Available*, we will try to satisfy the queued request.

(a) **P_0 requests C_0**: Because P_0's *Request* $= [1,0,0] \leq$ *Available* $= [1,1,1]$, this request can be satisfied. Please find a safe sequence by yourself. After this allocation, the matrix representation becomes:

	Allocation			*Max*			Need			*Available*		
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2
P_0	1	0	0	1	1	0	0	1	0	0	1	1
P_1	0	0	0	0	1	1	0	1	1			
P_2	0	0	0	1	0	1	1	0	1			

(b) P_1 **requests** C_1: Because P_1's $Request = [0,1,0] \leq Available = [0,1,1]$, this request can be satisfied. Please find a safe sequence by yourself. After this allocation, the matrix representation becomes:

	Allocation			*Max*			Need			*Available*		
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2
P_0	1	0	0	1	1	0	0	1	0	0	0	1
P_1	0	1	0	0	1	1	0	0	1			
P_2	0	0	0	1	0	1	1	0	1			

(c) P_2 **requests** C_2: P_2's $Request = [0,0,1] \leq Available = [0,0,1]$. Can we satisfy this request? According to the Resource-Request algorithm, pretending this request can be allocated and then updating the tables. In other words, P_2's $Allocation = [0,0,0] + Request = [0,0,1] = [0,0,1]$, $Need = [1,0,1] - Request = [0,0,1] = [1,0,0]$, and $Available = [0,0,1] - Request = [0,0,1] = [0,0,0]$. Therefore, the new matrices are:

	Allocation			*Max*			Need			*Available*		
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2
P_0	1	0	0	1	1	0	0	1	0	0	0	0
P_1	0	1	0	0	1	1	0	0	1			
P_2	0	0	1	1	0	1	1	0	0			

It is easily seen that the system is not in a safe state because none of processes can run as their *Request*s are all $\not\leq Available$. As a result, we have to return to the original tables and queue the request "P_2 *requests* C_2.

Now the actual tables are shown below, with the queue of requests:

	Allocation			*Max*			Need			*Available*			
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	*Queue*
P_0	1	0	0	1	1	0	0	1	0	0	0	1	P_2 request C_2
P_1	0	1	0	0	1	1	0	0	1				
P_2	0	0	0	1	0	1	1	0	1				

(d) P_0 **requests** C_1: P_0's $Request = [0,1,0] \not\leq Available = [0,0,1]$. This cannot be satisfied, and has to be queued. The matrix representation is shown below:

	Allocation			*Max*			Need			*Available*			
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	*Queue*
P_0	1	0	0	1	1	0	0	1	0	0	0	1	P_2 requests C_2
P_1	0	1	0	0	1	1	0	0	1				P_0 requests C_1
P_2	0	0	0	1	0	1	1	0	1				

(e) P_1 **requests** C_2: P_1's $Request = [0,0,1] \leq Available = [0,0,1]$. Let us do a pre-allocation check with banker's algorithm. P_1's new $Allocation = [0,1,1]$, P_1's new $Need = [0,0,0]$ and $Available = [0,0,0]$.

	Allocation			*Max*			Need			*Available*		
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2
P_0	1	0	0	1	1	0	0	1	0	0	0	0
P_1	0	1	1	0	1	1	0	0	0			
P_2	0	0	0	1	0	1	1	0	1			

We can easily find a safe sequence. For example, because P_1's $Need = [0,0,0] \leq Available = [0,0,0]$, it can run and return P_1's $Allocation = [0,1,1]$ making $Available = [0,0,0] + [0,1,1]$. This is equivalent to say that giving P_1 its right chopstick allowing P_1 to eat and releasing both chopsticks. After $Available = [0,1,1]$, then P_0's $Request = [0,1,0]$ can be satisfied. Finally, P_2 can run. Consequently, we have found a safe sequence $< P_1, P_0, P_2 >$. Therefore, this request can be satisfied and the matrix representation becomes:

	Allocation			*Max*			Need			*Available*			
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	*Queue*
P_0	1	0	0	1	1	0	0	1	0	0	0	0	P_2 requests C_2
P_1	0	1	1	0	1	1	0	0	0				P_0 requests C_1
P_2	0	0	0	1	0	1	1	0	1				

(f) <u>P_1 **releases** C_1 **and** C_2</u>: P_1 returns C_1 and C_2, making its $Allocation = [0,0,0]$, $Request = [0,0,0]$ and $Available = [0,1,1]$. Now the new matrix representation is:

	Allocation			*Max*			Need			*Available*			
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	*Queue*
P_0	1	0	0	1	1	0	0	1	0	0	1	1	P_2 requests C_2
P_1	0	0	0	0	1	1	0	0	0				P_0 requests C_1
P_2	0	0	0	1	0	1	1	0	1				

Now let us handle *Queue*. The first is P_2's $Request = [0,0,1]$. Can we satisfy this request? Pretending we can do that. Then, P_2's $Allocation = [0,0,1]$ (*i.e.*, getting C_2 – left chopstick), P_2's $Need = [1,0,1] - [0,0,1] = [1,0,0]$, and $Available = [0,1,0]$. This is equivalent to say that philosophers P_0 and P_2 have their left chopsticks C_0 and C_2, respectively.

	Allocation			*Max*			Need			*Available*		
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2
P_0	1	0	0	1	1	0	0	1	0	0	1	0
P_1	0	0	0	0	1	1	0	0	0			
P_2	0	0	1	1	0	1	1	0	0			

We can easily find a safe sequence $< P_1, P_0, P_2 >$. This is equivalent to say that giving C_1 to philosopher P_0, and after P_0 finishes eating, philosopher P_2 can take P_0's left chopstick C_0, which is P_2's right chopstick. Finally, P_2 has both chopsticks and eat!
After this allocation, we have the following matrix representation:

	Allocation			*Max*			Need			*Available*			
	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	C_0	C_1	C_2	*Queue*
P_0	1	0	0	1	1	0	0	1	0	0	1	0	P_0 requests C_1
P_1	0	0	0	0	1	1	0	0	0				
P_2	0	0	0	1	0	1	1	0	0				

Then, we may allow P_0 to eat, followed by P_2, etc. This finishes one cycle of eating. This next cycle is similar and is not repeated. ∎

4. **[10 points]** Consider the following snapshot of a system in which five resources A, B, C, D and E are available. The system contains a total of 2 instances of A, 1 of resource B, 1 of resource C, 2 resource D and 1 of resource E.

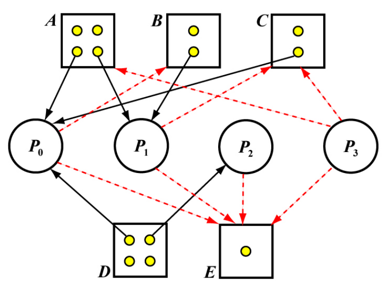
	Allocation					*Request*					*Available*				
	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
P_0	1	0	1	1	0	0	1	0	0	1	2	1	1	2	1
P_1	1	1	0	0	0	0	0	1	0	1					
P_2	0	0	0	1	0	0	0	0	0	1					
P_3	0	0	0	0	0	1	0	1	0	1					

Do the following problems:

- Convert this matrix representation to a resource allocation graph.
- Use the deadlock detection algorithm to determine whether the system contains a deadlock. Which processes are involved in the deadlock?
- While you are use the deadlock detection algorithm, add and remove directed edges of the resource allocation graph.

Always provide your detailed calculation. Or you risk a lower grade.

Answer: The corresponding resource allocation graph is shown below:

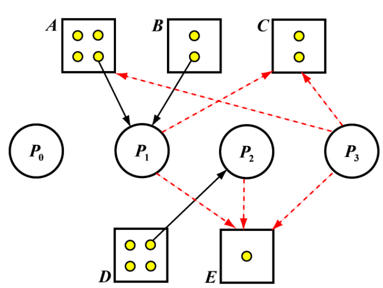


Note that red dashed-lined arrows are used to indicate unsatisfied "requests".

Currently, because $P_0's$ $Request = [0,1,0,0,1] \leq Available = [2,1,1,2,1]$, we run P_0 and reclaim its $Allocation$, and the new $Allocation =$ Old $Allocation + Available = [1,0,1,1,0] + [2,1,1,2,1] = [3,1,2,3,1]$. The new matrix representation becomes:

	Allocation					*Request*					*Available*				
	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
P_0											3	1	2	3	1
P_1	1	1	0	0	0	0	0	1	0	1					
P_2	0	0	0	1	0	0	0	0	0	1					
P_3	0	0	0	0	0	1	0	1	0	1					

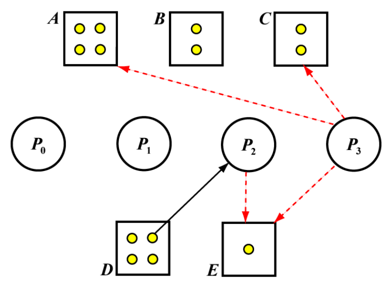
For the resource allocation graph, because all resources allocated to P_0 are returned and P_0 has no new request (so far), the new graph is obtained by removing all request edges and allocation edges as shown below:



Now, we have $P_1's\ Request = [0,0,1,0,1] \le Available = [3,1,2,3,1]$. We can run P_1, and reclaim its $Allocation = [1,1,0,0,0]$. The new $Available = [3,1,2,3,1] + P_1's\ Allocation = [1,1,0,0,0] = [4,2,2,3,1]$. As a result, the new matrix representation is:

	Allocation					*Request*					*Available*				
	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
P_0											4	2	2	3	1
P_1															
P_2	0	0	0	1	0	0	0	0	0	1					
P_3	0	0	0	0	0	1	0	1	0	1					

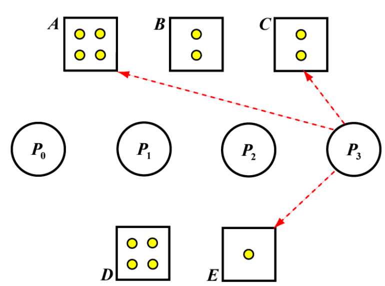
By removing all edges of P_1 yields a new resource allocation graph:



Next, we run P_2 and the new matrix representation and resource allocation graph are:

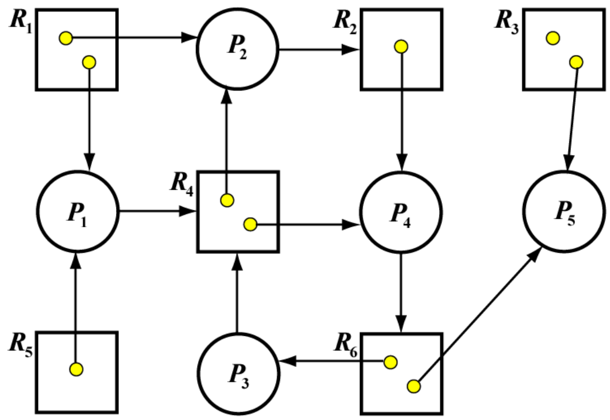
	Allocation					*Request*					*Available*				
	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
P_0											4	2	2	4	1
P_1															
P_2															
P_3	0	0	0	0	0	1	0	1	0	1					

Removing all edges from and to P_1 yields a new resource allocation graph:



Finally, P_3 can run and return all of its resources. Because all involved processes are finished, this system does no have a deadlock. ∎

5. **[10 points]** Consider the following resource allocation graph.



Do the following problems:

(a) Convert it to the matrix representation (*i.e.*, *Allocation*, *Request* and *Available*).

(b) Do a step-by-step execution of the deadlock detection algorithm. For each step, add and remove the directed edges, and redraw the resource allocation graph.

(c) Is there a deadlock? If there is a deadlock, which processes are involved?

Always provide your detailed calculation. Or you risk a lower grade.

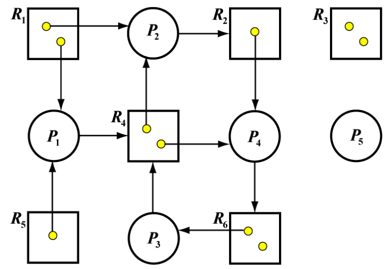
Answer: Note that the given resource allocation graph has cycles: $P_2 \to R_2 \to P_4 \to R_6 \to P_3 \to R_4 \to P_2$ and $P_4 \to R_6 \to P_3 \to R_4 \to P_4$. The following is the corresponding matrix representation.

	Allocation						*Request*						*Available*					
	R_1	R_2	R_3	R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6
P_1	1	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
P_2	1	0	0	1	0	0	0	1	0	0	0	0						
P_3	0	0	0	0	0	1	0	0	0	1	0	0						
P_4	0	1	0	1	0	0	0	0	0	0	0	1						
P_5	0	0	1	0	0	1	0	0	0	0	0	0						

We can run the system based on the deadlock detection algorithm. Note that other than R_3 there is no instance available in R_1, R_2, R_4, R_5 and R_6. The only runnable process is P_5. Because P_5's $Request = [0,0,0,0,0,0] \leq Available = [0,0,1,0,0,0$, we can run P_5 and reclaim its allocated resource. The new *Available* becomes *Available* = old *Available* + P_5's $Allocation = [0,0,1,0,0,0] + [0,0,1,0,0,1] = [0,0,2,0,0,1]$.

	Allocation						*Request*						*Available*					
	R_1	R_2	R_3	R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6
P_1	1	0	0	0	1	0	0	0	0	1	0	0	0	0	2	0	0	1
P_2	1	0	0	1	0	0	0	1	0	0	0	0						
P_3	0	0	0	0	0	1	0	0	0	1	0	0						
P_4	0	1	0	1	0	0	0	0	0	0	0	1						
P_5																		

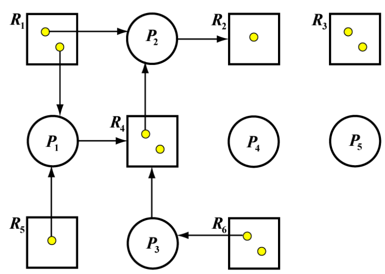
Removing all edged related to P_5 gives the following resource allocation graph:



Because the current $Available = [0,0,2,0,0,1] \geq P_4's\ Request = [0,0,0,0,0,1]$, we can run P_4 and reclaim P_4's allocation [0,1,0,1,0,0]. The new matrix representation is:

	Allocation						*Request*						*Available*					
	R_1	R_2	R_3	R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6
P_1	1	0	0	0	1	0	0	0	0	1	0	0	0	1	2	1	0	1
P_2	1	0	0	1	0	0	0	1	0	0	0	0						
P_3	0	0	0	0	0	1	0	0	0	1	0	0						
P_4																		
P_5																		

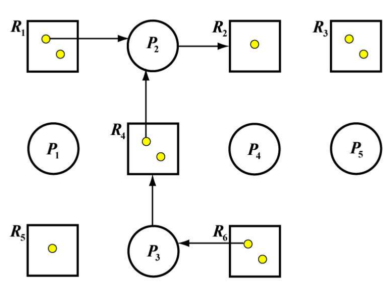
The new version of resource allocation graph is:



Now we can run P_1 because $P_1's\ Request = [0,0,0,1,0,0] \leq Available = [0,1,2,1,0,1]$. After P_1 finishes its work, the new $Available =$ old $Available = [0,1,2,1,0,1] + P_1's\ Allocation = [1,0,0,0,1,0] = [1,1,2,1,1]$. Thus, we have the following new matrix representation:

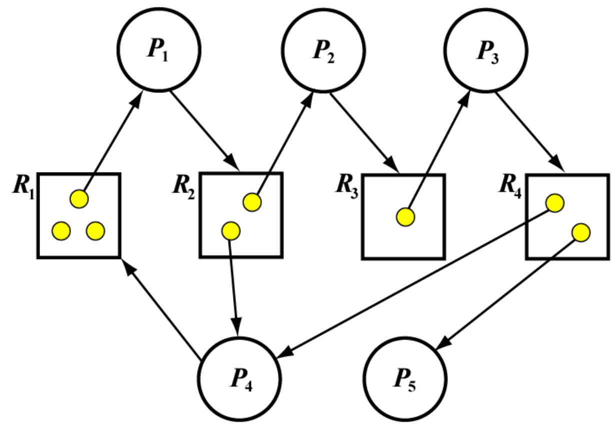
	Allocation						*Request*						*Available*					
	R_1	R_2	R_3	R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6	R_1	R_2	R_3	R_4	R_5	R_6
P_1													1	1	2	1	1	1
P_2	1	0	0	1	0	0	0	1	0	0	0	0						
P_3	0	0	0	0	0	1	0	0	0	1	0	0						
P_4																		
P_5																		

The resource allocation graph becomes:



Next, we run P_2 followed by P_3. In this way, all processes can run and obtain and release the needed resources. Consequently, there is no deadlock in the system. Thus, even though there are cycles in the given resource allocation graph, there is no deadlock! ∎

6. **[10 points]** Consider the following resource allocation graph.



Do the following problems:

(a) Convert it to the matrix representation (*i.e.*, *Allocation*, *request* and *Available*).

(b) Do a step-by-step execution of the deadlock detection algorithm. For each step, add and remove the directed edges, and redraw the resource allocation graph.

(c) Is there a deadlock? If there is a deadlock, which processes are involved?

Always provide your detailed calculation. Or you risk a lower grade.

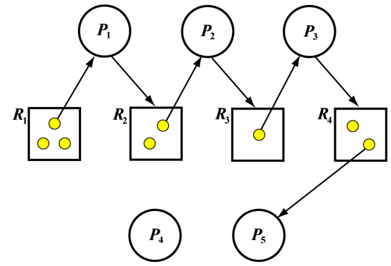
Answer: The matrix representation of the given resource allocation graph is shown below:

	Allocation				*Request*				*Available*			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	1	0	0	0	0	1	0	0	2	0	0	0
P_2	0	1	0	0	0	0	1	0				
P_3	0	0	1	0	0	0	0	1				
P_4	0	1	0	1	1	0	0	0				
P_5	0	0	0	1	0	0	0	0				

Because $P_4's\ Request = [1,0,0,0] \leq Available = [2,0,0,0], P_4$ runs and returns is $Allocation = [0,1,0,1]$ making the new $Available = [2,0,0,0] + [0,1,0,1] = [2,1,0,1]$. The matrix representation becomes:

	Allocation				*Request*				*Available*			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1	1	0	0	0	0	1	0	0	2	1	0	1
P_2	0	1	0	0	0	0	1	0				
P_3	0	0	1	0	0	0	0	1				
P_4												
P_5	0	0	0	1	0	0	0	0				

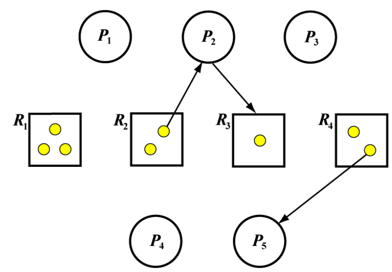
This is the corresponding resource allocation graph:



Then, we can run P_1 because P_1's $Request = [0,1,0,0] \leq Available = [2,1,0,1]$. After reclaiming P_1's $Allocation = [1,0,0,0]$, the new $Available$ is old $Avaliable = [2,1,0,1] + P_1's\ Allocation = [1,0,0,0] = [3,1,0,1]$. The new matrix representation is:

	Allocation				*Request*				*Available*			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1									3	1	0	1
P_2	0	1	0	0	0	0	1	0				
P_3	0	0	1	0	0	0	0	1				
P_4												
P_5	0	0	0	1	0	0	0	0				

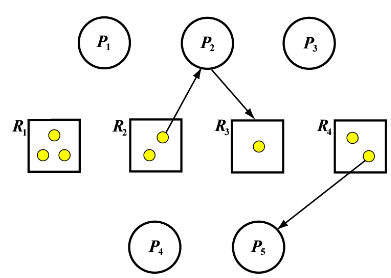
Here is the corresponding resource allocation graph:



The next process is P_3 because P_3's $Request = [0,0,0,1] \leq Available = [3,1,0,1]$. After P_3 finishes its work, its $Allocation = [0,0,1,0]$ is returned to $Available = [3,1,0,1] + [0,0,1,0] = [3,1,1,1]$:

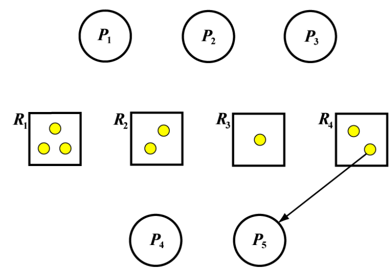
	Allocation				*Request*				*Available*			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1									3	1	1	1
P_2	0	1	0	0	0	0	1	0				
P_3												
P_4												
P_5	0	0	0	1	0	0	0	0				

The resource allocation graph is shown below:



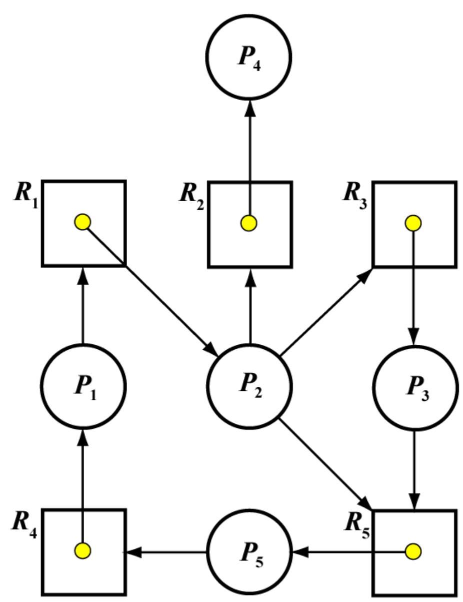
Now we can run P_2 and the yields (*i.e.*, matrix representation and resource allocation graph) are:

	Allocation				*Request*				*Available*			
	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4	R_1	R_2	R_3	R_4
P_1									3	2	1	1
P_2												
P_3												
P_4												
P_5	0	0	0	1	0	0	0	0				



Finally, we can run P_5 and all processes are done! Note that there are cycles $P_1 \to R_2 \to P_4 \to R_1 \to P_1$ and $P_1 \to R_2 \to P_2 \to R_3 \to P_3 \to R_4 \to P_4 \to R_1 \to P_1$. However, there is no deadlock. ∎

7. **[10 points]** Consider the following resource allocation graph.



Do the following problems:

(a) Convert it to the matrix representation (*i.e.*, *Allocation*, *Request* and *Available*).

(b) Do a step-by-step execution of the deadlock detection algorithm. For each step, add and remove the directed edges, and redraw the resource allocation graph.

(c) Is there a deadlock? If there is a deadlock, which processes are involved?

Always provide your detailed calculation. Or you risk a lower grade.

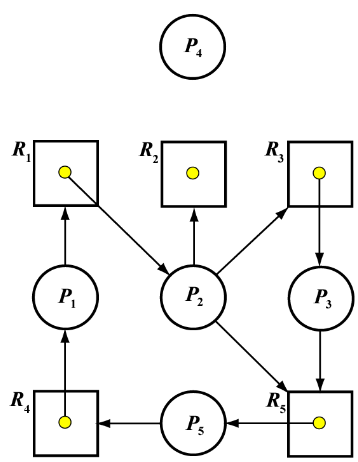
Answer: The matrix representation of the given resource allocation graph is shown below:

	Allocation					*Request*					*Available*				
	R_1	R_2	R_3	R_4	R_5	R_1	R_2	R_3	R_4	R_5	R_1	R_2	R_3	R_4	R_5
P_1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
P_2	1	0	0	0	0	0	1	1	0	1					
P_3	0	0	1	0	0	0	0	0	0	1					
P_4	0	1	0	0	0	0	0	0	0	0					
P_5	0	0	0	0	1	0	0	0	1	0					

P_4 can run because its $Request = [0,0,0,0,0] \leq Available = [0,0,0,0,0]$. The new $Available$ is calculated as old $Available = [0,0,0,0,0] + P_4's\ Allocation = [0,1,0,0,0] = [0,1,0,0,0]$. The new matrix representation is

	Allocation					*Request*					*Available*				
	R_1	R_2	R_3	R_4	R_5	R_1	R_2	R_3	R_4	R_5	R_1	R_2	R_3	R_4	R_5
P_1	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0
P_2	1	0	0	0	0	0	1	1	0	1					
P_3	0	0	1	0	0	0	0	0	0	1					
P_4															
P_5	0	0	0	0	1	0	0	0	1	0					

The corresponding resource allocation graph is:



None of the remaining processes (*i.e.*, P_1, P_2, P_3 and P_5) can run. As a result, we have a deadlock in this system. Note that we have cycles: $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_5 \rightarrow P_5 \rightarrow R_4 \rightarrow P_1$ and $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_5 \rightarrow P_5 \rightarrow R_4 \rightarrow P_1$. Thus, without actually working on a resource allocation graph to reduce it to its final form, having a cycle does not always mean there is a deadlock. ∎