Part III
Synchronization
Deadlocks and Livelocks

You think you know when you learn,
are more sure when you can write,
even more when you can teach,
but certain when you can program.

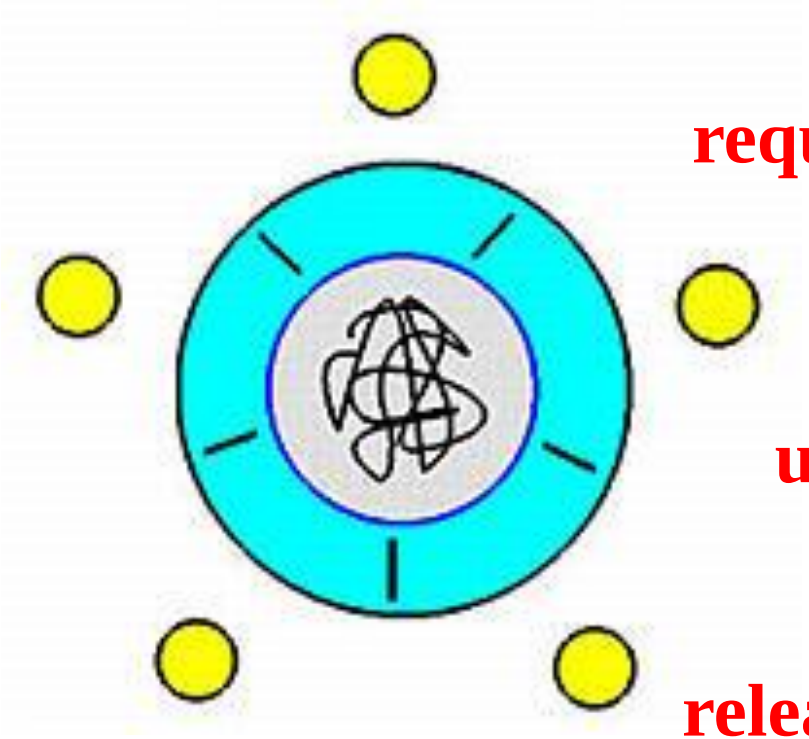
Alan J. Perlis

Deadlock Basics

System Model: 1/2

- System resources are used in the following way:
  - **Request**: If a process makes a request (i.e., semaphore wait or monitor acquire) to use a system resource which cannot be granted immediately, then the requesting process blocks until it can acquire the resource successfully.
  - **Use**: The process operates on the resource (i.e., in critical section).
  - **Release**: The process releases the resource (i.e., semaphore signal or monitor release).

System Model: 2/2



outer critical section

left chop locked

```
Semaphore C[5] = 1;
```

request

```
C[i].wait();
C[(i+1)%5].wait();
```

use

has 2 chops and eats

```
C[(i+1)%5].signal();
C[i].signal();
```

release

inner critical section

right chop locked

Deadlock: Definition

- **A set of processes is in a deadlock state when every process in the set is waiting for an event that can only be caused by another process in the same set.**
- **The key here is that processes are all in the waiting state.**

Deadlock Necessary Conditions

- **If a deadlock occurs, then each of the following four conditions must hold.**
  - **Mutual Exclusion**: At least one resource must be held in a non-sharable way.
  - **Hold-and- Wait**: A process must be holding a resource and waiting for another.
  - **No Preemption**: Resource cannot be preempted.
  - **Circular Waiting**: P_1 waits for P_2, P_2 waits for P_3, …, P_{n-1} waits for P_n, and P_n waits for P_1.

Deadlock Necessary Conditions

- **Note that the conditions are *necessary*.**
- **This means if a deadlock occurs ALL conditions are met.**
- **Because $p \Rightarrow q$ is equivalent to $\neg q \Rightarrow \neg p$, where $\neg q$ means not all conditions are met and $\neg p$ means no deadlock, as long as one of the four conditions fails there will be no deadlock.**

Deadlock Prevention

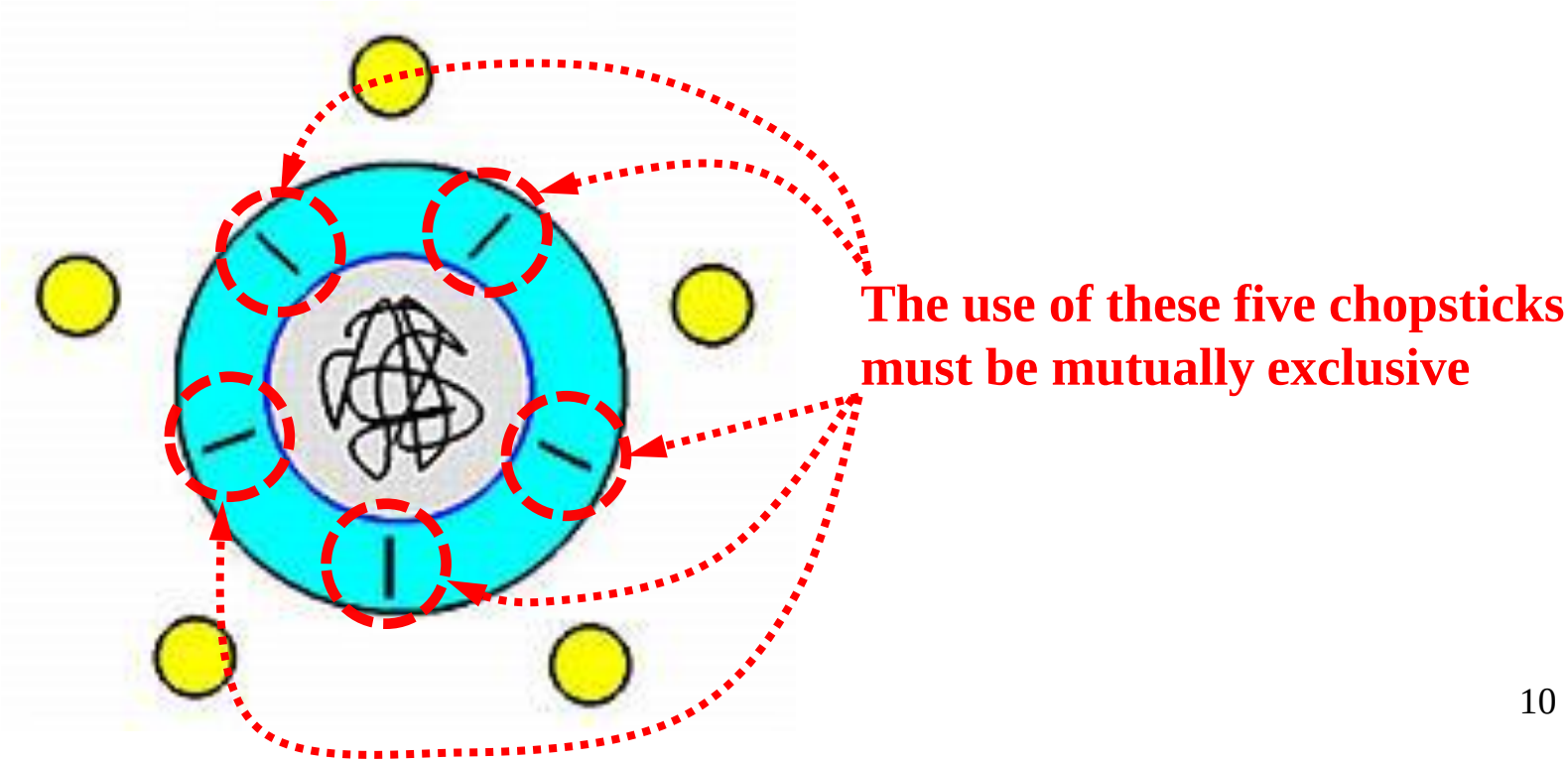
Make sure deadlock will never happen

Deadlock Prevention: 1/13

- **Deadlock Prevention** means making sure deadlocks never occur.
- To this end, if we can make sure at least one of the four conditions fails, there will be no deadlock.

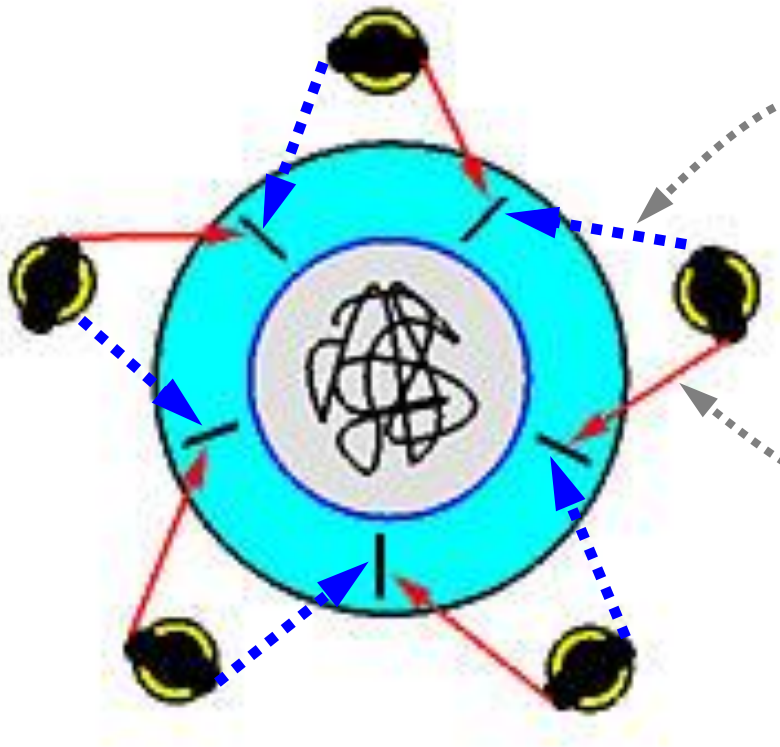
Deadlock Prevention: 2/13
Mutual Exclusion

- **Mutual Exclusion**: Some sharable resources must be accessed exclusively, which means we cannot deny mutual exclusion



Deadlock Prevention: 3/13
Hold-and-Wait

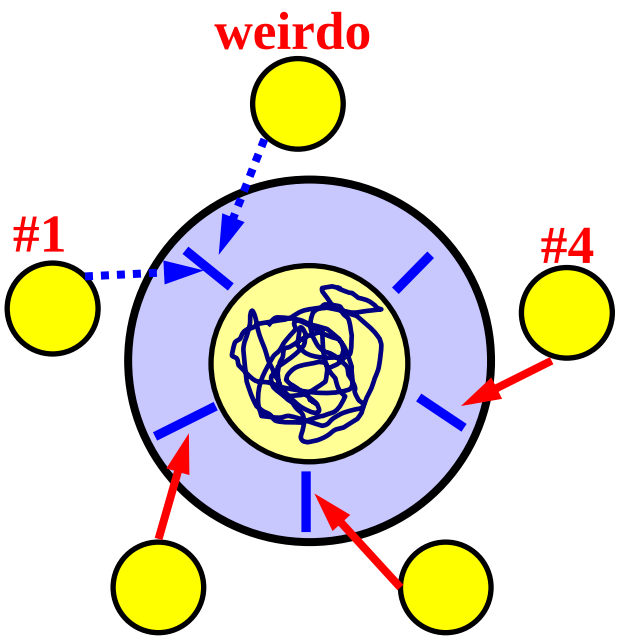
- **Hold-and-Wait**: A process holds some resources and requests for other resources.

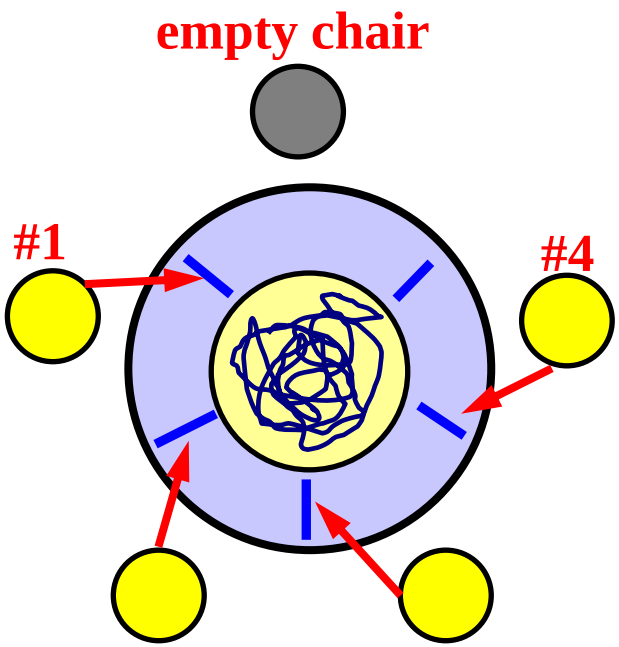


Deadlock Prevention: 4/13 Hold-and-Wait

- **Solution**: Make sure no process can hold some resources and request for other resources.
- Two strategies are possible (the monitor solution to the philosophers problem):
  - A process must acquire *all* resources before it runs.
  - When a process requests for resources, it must hold none (i.e., returning resources before requesting for more).
- Resource utilization may be low, since many resources will be held and unused for a long time.
- Starvation is possible. A process that needs some popular resources may have to wait indefinitely.

Deadlock Prevention: 5/13
Hold-and-Wait





If weirdo is faster than #1, #1 cannot eat
and the weirdo or #4 can eat but not both.
If weirdo is slower than #1, #4 can eat
Since there is no hold-and-wait,
there is no deadlock.

In this case, #4 has no right neighbor
and can take his right chop.
Since there is no hold-and-wait,
there is no deadlock.

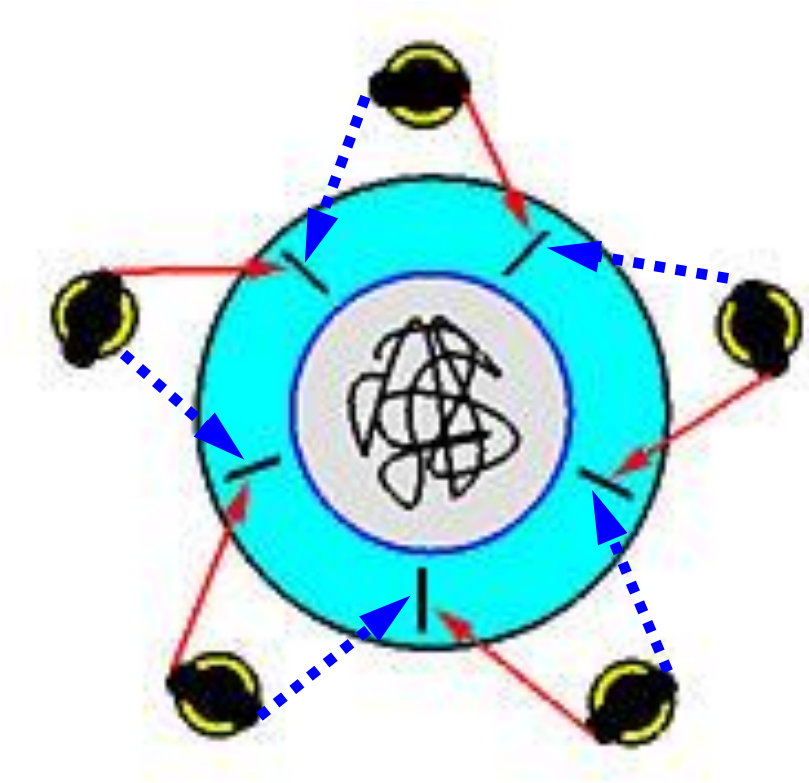
The solution with `THINKING-HUNGRY-EATING` states forces a philosopher to have both chops before eating. Hence, no hold-and-wait.

Deadlock Prevention: 6/13
No Preemption

- **This means resources being held by a process cannot be taken away (i.e., no preemption).**
- **To negate this no preemption condition, a process may deallocate all resources it holds so that the other processes can use.**
- **This is sometimes not doable: while philosopher *i* is eating, his neighbors cannot take *i*'s chops away forcing *i* to stop eating.**
- **Some resources cannot be reproduced cheaply (e.g., printer). However, the CPU can be preempted. Remember interrupts?**

Deadlock Prevention: 7/13
Circular Waiting

- **Circular Waiting**: P_1 waits for P_2, P_2 waits for P_3, …, P_{n-1} waits for P_n, and P_n waits for P_1.



The weirdo, 4-chair, and monitor solutions all avoid circular waiting and there is no deadlock.

How to break circular waiting in general?
Hierarchical Ordering of resources

Deadlock Prevention: 8/13

Circular Waiting

Hierarchical Ordering: 1/6

- **Resources are ordered in a *hierarchical* way.**
- **A process can only request resources higher than those it owns.**
- **To request lower order resources, a process must release all resources higher than or equal to the requesting ones.**

Deadlock Prevention: 9/13

Circular Waiting

Hierarchical Ordering: 2/6

- **Suppose a system orders its resources into 9 levels, L_1, L_2, …, L_8, L_9.**
- **Suppose the highest level of resources a process P_i has is L_5.**
- **Then, P_i can only request resources at L_6, L_7, L_8 and L_9.**
- **If P_i wishes to use a resource at L_3, P_i must release all of its resources at L_3, L_4 and L_5.**

Deadlock Prevention: 10/13

Circular Waiting

Hierarchical Ordering: 3/6

- **With Hierarchical Ordering, no circular waiting can occur.**
  - **Suppose that processes P_1, P_2, …, P_n involve in a resource waiting chain: P_i ($1 \le i < n$) waits for some higher-level resources being held by P_{i+1}.**
  - **The resource level of P_n is the highest of all P_i's.**
    1. **If P_n does not need any resource, it may run and release some resources for other processes to use.**
    2. **If P_n requests further resources at a higher level, because P_n is the last in the chain P_n can have these resources and run. Thus, we are back to (1) and Deadlock-free again!**

Deadlock Prevention: 11/13

Circular Waiting

Hierarchical Ordering: 4/6

- **Let us do a proof formally.**
- **Let $L(P_i)$ be the highest-level of resources P_i has.**
  - **Because P_i requests for resources at a higher-level being held by P_{i+1}, we have $L(P_i) < L(P_{i+1})$.**
  - **Therefore, we have**

$$L(P_1) < L(P_2) < \ldots < L(P_{n-1}) < L(P_n)$$

  - **If P_n waits for a resource being held by P_1, we have**

$$L(P_n) < L(P_1)$$

  - **We have $L(P_1) < L(P_n) < L(P_1)$, which is impossible.**
  - **Therefore, circular wait is impossible.**

Deadlock Prevention: 12/13

Circular Waiting

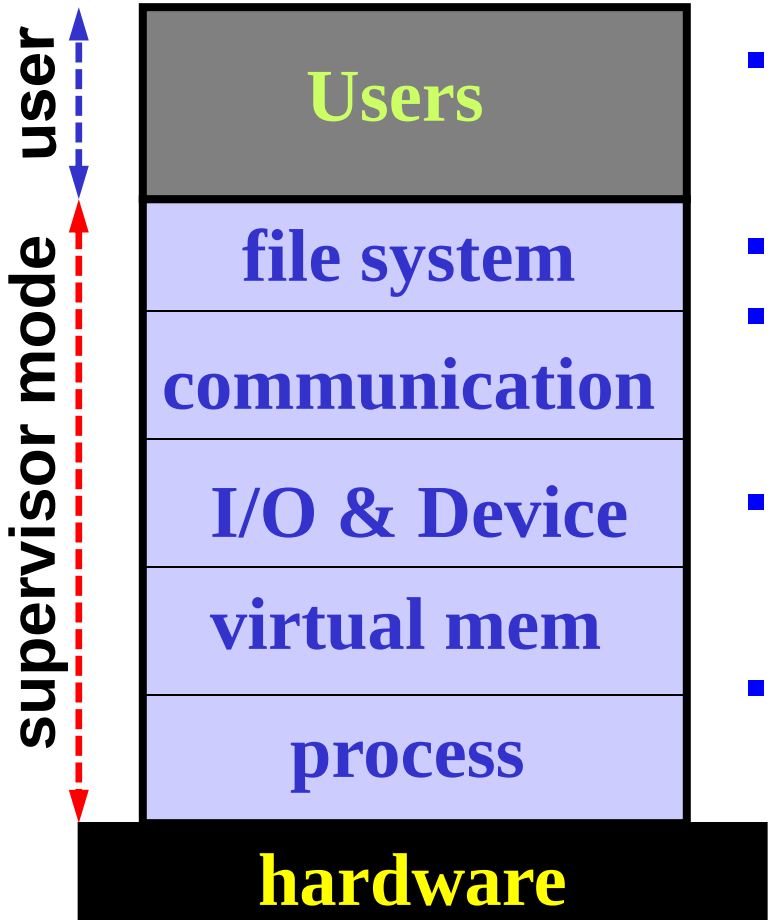
Hierarchical Ordering: 5/6

- **Let us revisit the dining philosophers problem.**
- **Chopsticks are ordered as $C_0 < C_1 < C_2 < C_3 < C_4$.**
- **Philosopher i requests chopstick C_i followed by C_{i+1} (*i.e.*, lower to higher).**
- **Philosopher 4 cannot request C_4 followed by C_0, because this violates the hierarchical ordering. To ensure hierarchical ordering, philosopher 4 must request C_0 followed by C_4 (lower to higher).**
- **Isn't this the weirdo (or lefty-righty) version?**
- **Therefore, the weirdo version is deadlock-free.**

Deadlock Prevention: 13/13

Circular Waiting

Hierarchical Ordering: 6/6



- The hierarchical ordering was used in the **THE** operating system developed by Dijkstra.
- The lowest layer is ***process management***.
- Each layer ***only uses the operations provided by lower layers*** and does not have to know their implementation.
- Each layer hides the existence of certain data structures, operations and hardware from higher-level layers.
- **A major difficulty in the hierarchical ordering is that it is difficult to arrange resources into a linear ordering.**

Edsger W. Dijkstra, The Structure of the "THE"-Multiprogramming System, *Communications of the ACM*, Vol. 11 (1968), No. 5 (May), pp. 341-346.

THE = **T**echnische **H**ogeschool Eindhoven

Deadlock Avoidance

Deadlocks can be avoided but not totally prevented

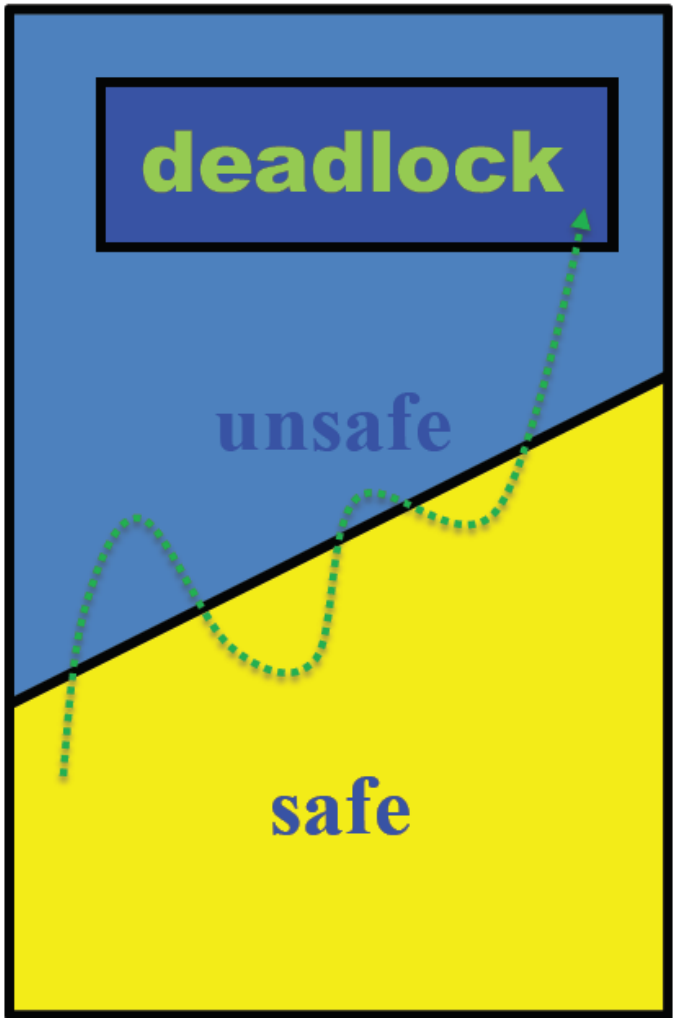
Safe Sequences and Safe State

Safe Sequences

- **Each process provides the maximum number of resources of each type it needs.**
- **There are algorithms that can ensure the system will never enter a deadlock state. This is deadlock avoidance.**
- **A sequence of processes <P_1, P_2, …, P_n> is a safe sequence if for each process P_i in the sequence, its resource requests can be satisfied by the remaining resources and the sum of all resources that are being held by P_1, P_2, …, P_{i-1}. This means we can suspend P_i and run P_1, P_2, …, P_{i-1} until they complete. Then, P_i will have all resources to run.**

Safe State

- A state is **safe** if the system can allocate resources to each process (up to its maximum, of course) in some order and still avoid a deadlock.
- Thus, a state is **safe** if there is a safe sequence. Otherwise, if no safe sequence exists, the system state is **unsafe**.
- An unsafe state is not necessarily a deadlock state. On the other hand, a deadlock state is an unsafe state.



Example: 1/2

- A system has 12 tapes and three processes *A*, *B*, *C*. At time t_0, we have:

3 free tapes

	Max needs	Current holding	Will need
A	10	5	5
B	4	2	2
C	9	2	7

- Then, <*B*, *A*, *C*> is a safe sequence (safe state).
- The system has 12-(5+2+2)=3 free tapes.
- Because *B* needs 2 tapes, it can take 2, run, and return 4. After *B* completes, the system has (3-2)+4=5 tapes. *A* now can take all 5 tapes and run. Finally, *A* returns 10 tapes for *C* to take 7 of them.

Example: 2/2

- A system has 12 tapes and three processes *A*, *B*, *C*. At time t_1, *C* has one more tape:

	Max needs	Current holding	Will need
A	10	5	5
B	4	2	2
C	9	3	6

- The system has 12-(5+2+3)=2 free tapes.
- At this point, only *B* can take these 2 and run. It returns 4, making 4 free tapes available.
- But, none of *A* and *C* can run, and a deadlock occurs.
- The problem is due to granting *C* one more tape.

Deadlock Avoidance and Banker’s Algorithm

Deadlock Avoidance

- A **deadlock avoidance algorithm** ensures that the system is always in a safe state. Therefore, no deadlock can occur.
- Resource requests are granted **only if in doing so the system is still in a safe state**.
- Consequently, resource utilization may be ***lower*** than those systems without using a deadlock avoidance algorithm.

Basic Idea: 1/2

- **Suppose you are a banker with a limited asset.**
- **If a customer *A* wishes to borrow *X* amount.**
- **If you have no less than *X*, *A* can have it.**
- **If you have less than *X*, *A*'s request cannot be satisfied, and must wait until other customers pay their loans off so that the returned amount plus the amount in hand can satisfy *A*'s request.**

Basic Idea: 2/2

- **Suppose you have $1,000 and your brothers Adam, Bill and Chuck borrowed $400, $300 and $100, respectively.**
- **Thus, you have only $200 in hand.**
- **What if Adam asks for $600 more?**
- **You don't have that amount for him. But, you think: if Bill and Chuck will return their $400 and $300 (assuming everyone is honest), respectively, you will have enough for Adam.**
- **In this way, you just ask Adam to wait until Bill and Chuck return the money.**

Assumptions

- **The system has *m* resource types.**
- **Each resource type has several units.**
- **There are *n* processes, each of which declares its maximum needs although these resources may not all be used at the same time.**
- **The deadlock avoidance system requires several matrices to keep track the states of resource allocation.**

Data Structures: 1/2

- **The following arrays are used:**
  - ***Available*[1..*m*]: one entry for each resource. *Available*[*i*]=*k* means resource type *i* has *k* units available.**
  - ***Max*[1..*n*,1..*m*]: maximum demand of each process. *Max*[*i*,*j*]=*k* means process *i* needs *k* units of resource *j*.**
  - ***Allocation*[1..*n*,1..*m*]: resources allocated to each process. *Allocation*[*i*,*j*]=*k* means process *i* is currently allocated *k* units of resource *j*.**
  - ***Need*[1..*n*,1..*m*]: the remaining resource need of each process. *Need*[*i*,*j*]=*k* means process *i* needs *k* more units of resource *j*.**
  - **Thus, *Max* = *Allocation* + *Need*.**

Data Structures: 2/2

- **We will use $A[i,*]$ to indicate the i-th row of matrix A.**
- **Given two arrays $A[1..m]$ and $B[1..m]$,**
  - $A \leq B$ **if** $A[i] \leq B[i]$ **for all** i**.**
- **Given $A[1..n,1..m]$ and $B[1..n,1..m]$,**
  - $A[i,*] \leq B[i,*]$ **if** $A[i,j] \leq B[i,j]$ **for all** j**.**

Finding a Safety Sequence
The Safety Algorithm

1. Let *Work*[1..*m*] and *Finish*[1..*n*] be two working arrays.
2. *Work* := *Available* and *Finish*[*i*]=*FALSE* for all *i*
3. Find an *i* such that the following both hold
   - *Finish*[*i*] = *FALSE* // process *i* is not yet done
   - *Need*[*i*,*] ≤ *Work* // its need can be satisfied

   If no such *i* exists, go to Step 5
4. *Work* = *Work* + *Allocation*[*i*,*] // run it and reclaim

   Finish[*i*] = *TRUE* // process *i* completes

   go to Step 3
5. If *Finish*[*i*] = *TRUE* for all *i*, the system is in a safe state.

add this process *i* to the safe sequence

if all processes can be added to the safe sequence,
a safe sequence is constructed, and the system is safe.

Banker's Algorithm: 1/3

- In addition to *Available*, *Max*, *Allocation* and *Need*, we need one more array *Request*[1..*n*,1..*m*] to ensure a resource request can be granted.
- *Request*[*i*,*j*] = *k*: process *i* requests *k* units of resource type *j*.
- Process *i* calls **Request()** with an argument *x*[1..*m*], where *x*[*j*] = *k* means process *i* requests *k* units of resource type *j*.
- After process *i* finishes using some resources, it calls **Release()** with argument *x*[1..*m*], where *x*[*j*] = *k* means process *i* releases *k* units of resource type *j*.

Banker's Algorithm: 2/3

- Process *i* calls **Request(*x*)** with its resource request in argument *x*.
- The **Resource-Request** algorithm uses *x* to update the *Request*[1..*n*,1..*m*] matrix.
- Then, **Resource-Request** runs the checking procedure by calling the **Safety Algorithm** to determine whether the system is in a safe state if the request is allocated.
- If it is safe, the process gets the resources. Otherwise, it is blocked until the process can get the needed resources.

Banker's Algorithm: 3/3

Resource-Request

1. Let *Request*[1..*n*,1..*m*] be the request matrix. *Request*[*i*,*j*]=*k* means process *i* requests *k* units of resource *j*.
2. If *Request*[*i*,*] ≤ *Need*[*i*,*], go to Step 3. Otherwise, it is an error.
3. If *Request*[*i*,*] ≤ *Available*, go to Step 4. Otherwise, process *i* waits.
4. Do the following:

   Available = *Available* – *Request*[*i*,*]

   Allocation[*i*,*] = *Allocation*[*i*,*] + *Request*[*i*,*]

   Need[*i*,*] = *Need*[*i*,*] – *Request*[*i*,*]
5. Call the Safety Algorithm.
6. If the request is safe, the requested resources are granted. Otherwise, process *i* waits and the resource-allocation tables are restored back to the original.

Example 1: 1/4

- Consider a system of 5 processes *A*, *B*, *C*, *D* and *E*, and 3 resource types (*X*=10, *Y*=5, *Z*=7). At time t_0, we have

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	7	5	3	7	4	3	3	3	2
B	2	0	0	3	2	2	1	2	2			
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

- The table is always scanned **top-down**.
- Because *B*'s *Need* = [1,2,2] ≤ *Available* = [3,2,2], *B* runs.
- After *B* returns its *Allocation*. Therefore, *Allocation* = [2,0,0] + [3,3,2] = [5,3,2].
- **Safe Sequence**: *B*

Example 1: 2/4

- Consider a system of 5 processes *A*, *B*, *C*, *D* and *E*, and 3 resource types (*X*=10, *Y*=5, *Z*=7). At time t_0, we have

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	7	5	3	7	4	3	5	3	2
B												
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

- The table is always scanned **top-down**.
- *D* runs next. After this, *Available* = [5,3,2] + [2,1,1]=[7,4,3].
- **Safe Sequence**: *B*, *D*

Example 1: 3/4

- Consider a system of 5 processes *A*, *B*, *C*, *D* and *E*, and 3 resource types (*X*=10, *Y*=5, *Z*=7). At time t_0, we have

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	7	5	3	7	4	3	7	4	3
B												
C	3	0	2	9	0	2	6	0	0			
D												
E	0	0	2	4	3	3	4	3	1			

- The table is always scanned **top-down**.
- *A* runs next.
- Then, *Available* = [7,4,3] + [0,1,0] = [7,5,3].
- **Safe Sequence**: *B*, *D*, *A*

Example 1: 4/4

- Consider a system of 5 processes *A*, *B*, *C*, *D* and *E*, and 3 resource types (*X*=10, *Y*=5, *Z*=7). At time t_0, we have

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A										7	5	3
B												
C	3	0	2	9	0	2	6	0	0			
D												
E	0	0	2	4	3	3	4	3	1			

- The table is always scanned **top-down**.
- Because *C*'s [6,0,0] ≤ *Available* = [7,5,3], *C* runs.
- After this, *Available* = [7,5,3] + [3,0,2] = [10,5,7] and *E* runs.
- **Safe Sequence**: *B*, *D*, *A*, *C*, *E*.
- There are other safe sequences: <*D*,*E*,*B*,*A*,*C*>, <*D*,*B*,*A*,*E*,*C*>, …

Example 2: 1/3

- **Now suppose process *B* asks for 1 *X* and 2 *Z*s. More precisely, $Request_B$ = [1,0,2].** ***Is the system still in a safe state if this request is granted?***
- **Since $Request_B$ = [1,0,2] ≤ *Available* = [3,3,2], this request may be granted if the system is safe.**
- **If this request is granted, we have the following:**

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	7	5	3	7	4	3	2	3	0
B	3	0	2	3	2	2	0	2	0			
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

[3,0,2]=[2,0,0]+[1,0,2] **[0,2,0]=[1,2,2]-[1,0,2]** **[2,3,0]=[3,3,2]-[1,0,2]**

Example 2: 2/3

	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	***Y***	***Z***	***X***	***Y***	***Z***	***X***	***Y***	***Z***	***X***	***Y***	***Z***
A	0	1	0	7	5	3	7	4	3	2	3	0
B	3	0	2	3	2	2	0	2	0			
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

- ***Is the system in a safe state after this allocation?***
- **Yes, because the safety algorithm will provide a safe sequence <*B*,*D*,*E*,*A*,*C*>. Verify it yourself.**
- **Hence, *B*'s request of [1,0,2] can safely be made.**

Example 2: 3/3

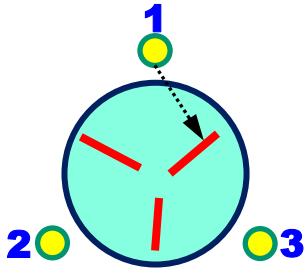
	Allocation			*Max*			*Need=Max-Alloc*			*Available*		
	X	***Y***	***Z***	***X***	***Y***	***Z***	***X***	***Y***	***Z***	***X***	***Y***	***Z***
A	0	1	0	7	5	3	7	4	3	2	3	0
B	3	0	2	3	2	2	0	2	0			
C	3	0	2	9	0	2	6	0	0			
D	2	1	1	2	2	2	0	1	1			
E	0	0	2	4	3	3	4	3	1			

- **Then, *E*'s request $Request_E$=[3,3,0] cannot be granted because $Request_E$ = [3,3,0] ≤ [2,3,0] is false.**
- ***A*'s request $Request_A$=[0,2,0] cannot be granted because the system will be unsafe.**
- **If $Request_A$ = [0,2,0] is granted, *Available* = [2,1,0] = [2,3,0] – [0,2,0].**
- **None of the five processes can finish and the system is unsafe.**

Example 3: 1/12

- **Let us look at how Banker's algorithm handles the dining philosophers problem.**
- **We use three philosophers P_1, P_2 and P_3 and three chopsticks C_1, C_2 and C_3.**
- **Philosopher P_i uses chopsticks C_i and $C_{i\%3+1}$.**
- **What if P_1 requests C_1 (left), P_2 requests C_2 (left), P_3 requests C_3 (left), P_1 requests C_2 (right), P_2 requests C_3 (right), P_3 requests C_1 (right).**
- **This request sequence can cause a deadlock.**
- **Can Banker's algorithm overcome this problem?**

Example 3: 2/12



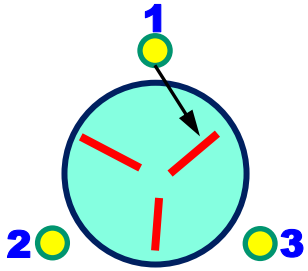
- **Initially we have the following:**

	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
P_1	0	0	0	1	1	0	1	1	0	1	1	1			
P_2	0	0	0	0	1	1	0	1	1						
P_3	0	0	0	1	0	1	1	0	1						

- ***P_1* requests to use *C_1*, *Request* = [1,0,0].**

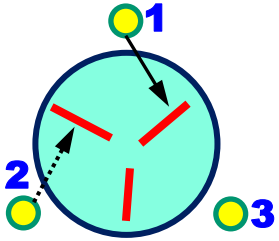
	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
P_1	0	0	0	1	1	0	1	1	0	1	1	1	1	0	0
P_2	0	0	0	0	1	1	0	1	1						
P_3	0	0	0	1	0	1	1	0	1						

Example 3: 3/12



- **Assume this request is granted:**

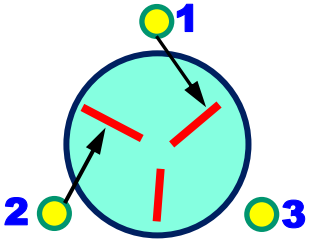
	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
P_1	1	0	0	1	1	0	0	1	0	0	1	1	1	0	0
P_2	0	0	0	0	1	1	0	1	1						
P_3	0	0	0	1	0	1	1	0	1						



- **Obviously, the system is safe.**
- ***P_2* requests to use *C_2*, *Request* = [0,1,0].**

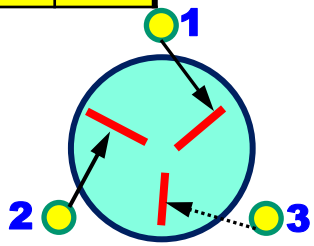
	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
P_1	1	0	0	1	1	0	0	1	0	0	1	1			
P_2	0	0	0	0	1	1	0	1	1				0	1	0
P_3	0	0	0	1	0	1	1	0	1						

Example 3: 4/12



- **Assume this request is granted:**

	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
P_1	1	0	0	1	1	0	0	1	0	0	0	1			
P_2	0	1	0	0	1	1	0	0	1				0	1	0
P_3	0	0	0	1	0	1	1	0	1						



- **Obviously, the system is safe, again.**
- ***P_3* requests to use *C_3*, *Request* = [0,0,1].**

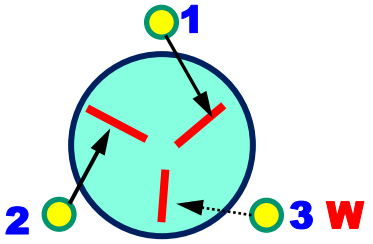
	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
P_1	1	0	0	1	1	0	0	1	0	0	0	1			
P_2	0	1	0	0	1	1	0	0	1						
P_3	0	0	0	1	0	1	1	0	1				0	0	1

Example 3: 5/12

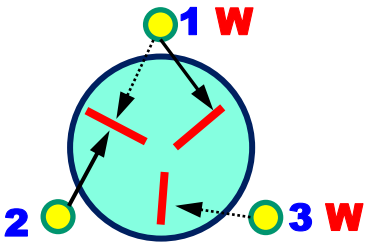
- **Assume this request is granted:**

	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*		
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3
P_1	1	0	0	1	1	0	0	1	0	0	0	0			
P_2	0	1	0	0	1	1	0	0	1						
P_3	0	0	1	1	0	1	1	0	0				0	0	1

- **Is the system safe?**
- **No! <u>There is no safe sequence</u>, because none of the three *Need* array rows is ≤ *Available*.**
- **As a result, P_3 must wait, and a possible deadlock is avoided.**



Example 3: 6/12



- P_3's request cannot be granted:

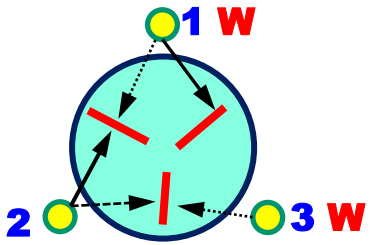
	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*			
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	
P_1	1	0	0	1	1	0	0	1	0	0	0	1				
P_2	0	1	0	0	1	1	0	0	1							
P_3	0	0	0	1	0	1	1	0	1				0	0	1	wait

- P_1 requests to use C_2, *Request* = [0,1,0].

	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*			
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	
P_1	1	0	0	1	1	0	0	1	0	0	0	1	0	1	0	
P_2	0	1	0	0	1	1	0	0	1							
P_3	0	0	0	1	0	1	1	0	1				0	0	1	wait

- Because P_1's *Request* is not ≤ *Available*, P_1 waits!

Example 3: 7/12



- **Now we have the following:**

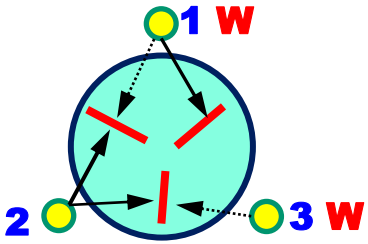
	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*			
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	
P_1	1	0	0	1	1	0	0	1	0	0	0	1	0	1	0	wait
P_2	0	1	0	0	1	1	0	0	1							
P_3	0	0	0	1	0	1	1	0	1				0	0	1	wait

- **P_2 requests C_3? Thus, *Request* = [0,0,1].**

	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*			
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	
P_1	1	0	0	1	1	0	0	1	0	0	0	1	0	1	0	wait
P_2	0	1	0	0	1	1	0	0	1				0	0	1	
P_3	0	0	0	1	0	1	1	0	1				0	0	1	wait

- **Can this request be satisfied?**

Example 3: 8/12



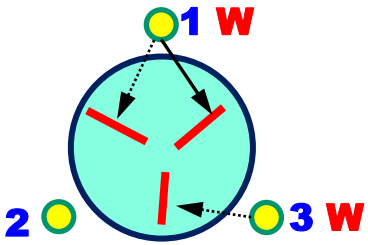
- Assume P_2's *Request* = [0,0,1] is granted:

	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*			
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	
P_1	1	0	0	1	1	0	0	1	0	0	0	0	0	1	0	wait
P_2	0	1	1	0	1	1	0	0	0				0	0	1	
P_3	0	0	0	1	0	1	1	0	1				0	0	1	wait

- Because P_2's *Need* ≤ *Available*, P_2 can run. Note that P_1 and P_3 are waiting, and the system is safe.
- After P_2 finishes using the requested resources, P_2 releases C_2 and C_3 :

	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*			
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	
P_1	1	0	0	1	1	0	0	1	0	0	1	1	0	1	0	wait
P_2	0	0	0	0	1	1	0	1	1							
P_3	0	0	0	1	0	1	1	0	1				0	0	1	wait

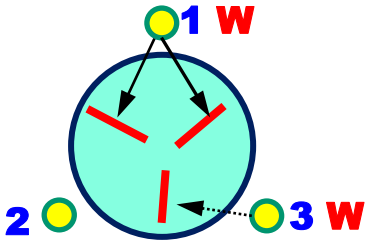
Example 3: 9/12



- Now we have the following:

	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*			
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	
P_1	1	0	0	1	1	0	0	1	0	0	1	1	0	1	0	wait
P_2	0	0	0	0	1	1	0	1	1							
P_3	0	0	0	1	0	1	1	0	1				0	0	1	wait

- P_1 and P_3 are waiting. Can one of them run?
- Because P_1's *Need* = [0,1,0] ≤ *Available* = [0,1,1], P_1 can run.
- Note that P_3 cannot run, because its *Need* is not ≤ *Available*.



Example 3: 10/12

- **After P_1 finishes, we have the following:**

	Allocation			*Max*			*Need=Max-Alloc*			*Available*			*Request*			
	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	C_1	C_2	C_3	
P_1	0	0	0	1	1	0	1	1	0	1	1	1				
P_2	0	0	0	0	1	1	0	1	1							
P_3	0	0	0	1	0	1	1	0	1				0	0	1	**wait**

- **Now P_3 can run, because its *Need* = [0,0,1] ≤ *Available* = [1,1,1].**
- **Please verify the remaining activities.**
- **Banker's algorithm avoids possible deadlocks.**

Example 3: 11/12

- **Please verify the following:**
  - **We have seen the version that requires each philosopher to pick up BOTH chopsticks at the same time. Thus, requests by P_1, P_2 and P_3 are [1,1,0], [0,1,1] and [1,0,1], respectively. Use Banker's algorithm to show that it is deadlock-free.**
  - **Do the same with the weirdo (or lefty-righty) version.**

Example 3: 12/12

- **After a process returns its resources, a waiting process may be released if its *Request* ≤ *Available* and the system is safe.**
- **When searching for a waiting process to be resumed, one may implement some search order such as**
  - **Priority**
  - **FIFO**
  - **Aging: priority gets higher as waiting longer**
  - **Some form of "fitting": first-fit, best-fit, worst-fit**
- **Banker's algorithm can easily be implemented with a monitor.**

Deadlock Detection

Deadlocks are not avoided but they will be detected

Deadlock Detection: 1/2

- **If deadlock prevention or avoidance are not used, then a deadlock situation may occur. Thus, we need:**
  - **An algorithm that can examine the system state to determine if a deadlock has occurred. This is a *deadlock detection* algorithm.**
  - **An algorithm that can help recover from a deadlock. This is a *recovery* algorithm.**

Deadlock Detection: 2/2

- **A deadlock detection algorithm does not have to know the maximum need *Max* of a process.**
- **Thus, the current need *Need* is not available.**
- **A deadlock detection algorithm only uses *Available*, *Allocation* and *Request*.**
- **A deadlock detection algorithm works like Banker's algorithm without *Need*.**
  - **Whenever *Request* ≤ *Available*, the requested resources are allocated.**
  - **Those processes that cannot get the requested resources may involve in a deadlock.**

Deadlock Detection Algorithm

1. Let *Work*[1..*m*] and *Finish*[1..*n*] be two working arrays.
2. *Work* := *Available* and *Finish*[*i*]=*FALSE* for all *i*
3. Find an *i* such that both
   - *Finish*[*i*] = *FALSE* // process *i* is not yet done
   - *Request*[*i*,*] ≤ *Work* // its request can be satisfied

   If no such *i* exists, go to Step 5
4. *Work* = *Work* + *Allocation*[*i*,*] // run it and reclaim

   Finish[*i*] = *TRUE* // process *i* completes

   go to Step 3
5. If *Finish*[*i*] = *TRUE* for all *i*, the system is in a safe state. If *Finish*[*i*] = *FALSE*, then process P_i is deadlocked.

Use *Request* here rather than *Need* as in the safety algorithm

Example 1: 1/5

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	0	0	0	0	0	0
B	2	0	0	2	0	2			
C	3	0	3	0	0	0			
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			

- **Suppose maximum available resource is [7,2,6] and the current state of resource allocation is shown above.**
- ***Is the system deadlocked?***

Example 1: 2/5

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	0	0	0	0	0	0
B	2	0	0	2	0	2			
C	3	0	3	0	0	0			
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			

- **Because *A*'s *Request* ≤ *Available*, run *A* first.**
- **Then, *A* returns [0,1,0] and the new *Available* = [0,0,0] + [0,1,0] = [0,1,0].**

Example 1: 3/5

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A							0	1	0
B	2	0	0	2	0	2			
C	3	0	3	0	0	0			
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			

- **Because *C*'s *Request* ≤ *Available*, run *C*.**
- **Then, *C* returns [3,0,3] and the new *Available* = [0,1,0] + [3,0,3] = [3,1,3].**

Example 1: 4/5

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A							3	1	3
B	2	0	0	2	0	2			
C									
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			

- **Because *B*'s *Request* ≤ *Available*, run *B*.**
- **Then, *B* returns [2,0,0] and the new *Available* = [3,1,3] + [2,0,0] = [5,1,3].**

Example 1: 5/5

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A							5	1	3
B									
C									
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			

- **Then, we can run either *D* or *E*.**
- **As a result, this is not a deadlock state.**

Example 2: 1/3

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	0	0	0	0	0	0
B	2	0	0	2	0	2			
C	3	0	3	0	0	1			
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			

- **Suppose *C* requests for one more resource *Z*.**
- ***Is this a deadlock state?***

Example 2: 2/3

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A	0	1	0	0	0	0	0	0	0
B	2	0	0	2	0	2			
C	3	0	3	0	0	1			
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			

- **Because *A*'s *Request* ≤ *Available*, *A* runs.**
- **Then, *A* returns its *Allocation*, and the new *Available* = [0,0,0] + [0,1,0] = [0,1,0].**

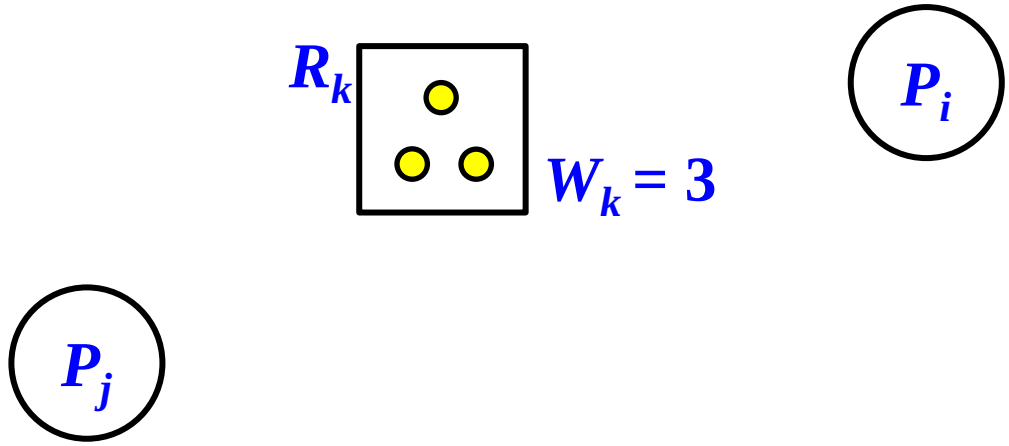
Example 2: 3/3

	Allocation			*Request*			*Available*		
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*
A							0	1	0
B	2	0	0	2	0	2			
C	3	0	3	0	0	1			
D	2	1	1	1	0	0			
E	0	0	2	0	0	2			

- **None of the four remaining processes can run.**
- **Therefore, this is a deadlock state.**

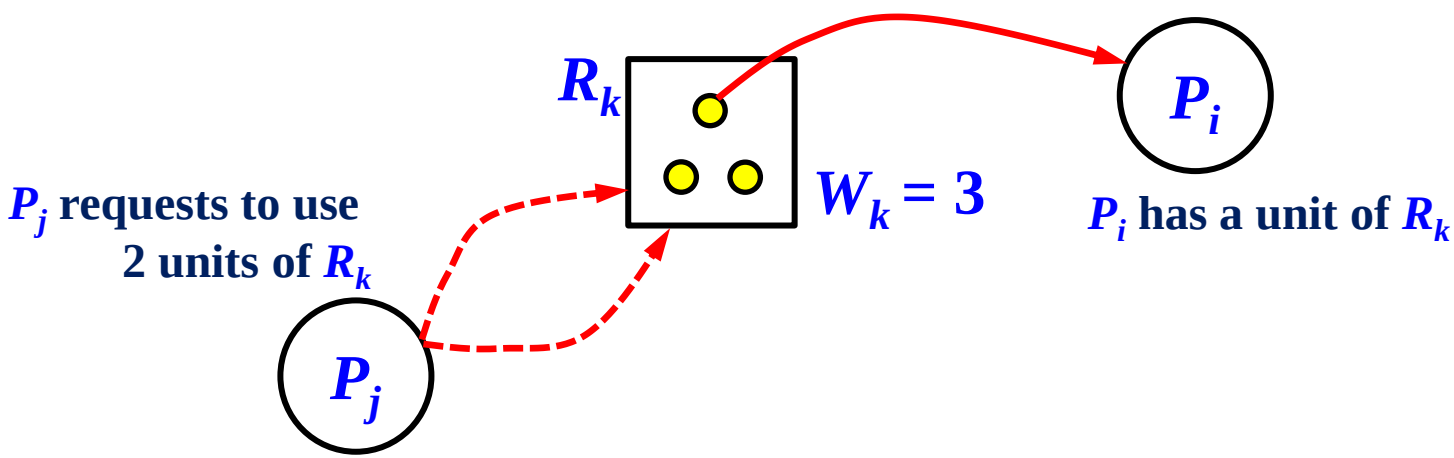
Resource Allocation Graph: 1/9

- Suppose we have m resource types, R_1, R_2, …, R_m and resource type R_i has W_i units.
- Each resource type is represented by a *rectangular* node in which each "dot" indicates a unit. Therefore, resource type R_i has W_i dots.
- Suppose there are n processes P_1, P_2, …, P_n, each process is represented by a *circular* node.



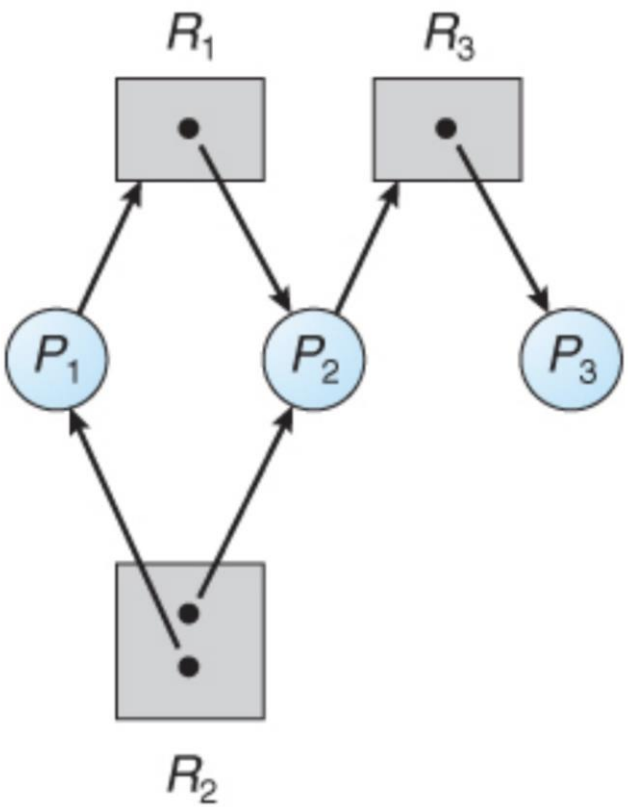
Resource Allocation Graph: 2/9

- If process P_i makes a request to use a unit of resource R_j, draw a directed edge from P_i to R_j.
- If process P_i receives a resource of type R_j, draw a directed edge from a **dot** of R_j to P_i.
- This is a **Resource Allocation Graph**, **RAG**.



Resource Allocation Graph: 3/9

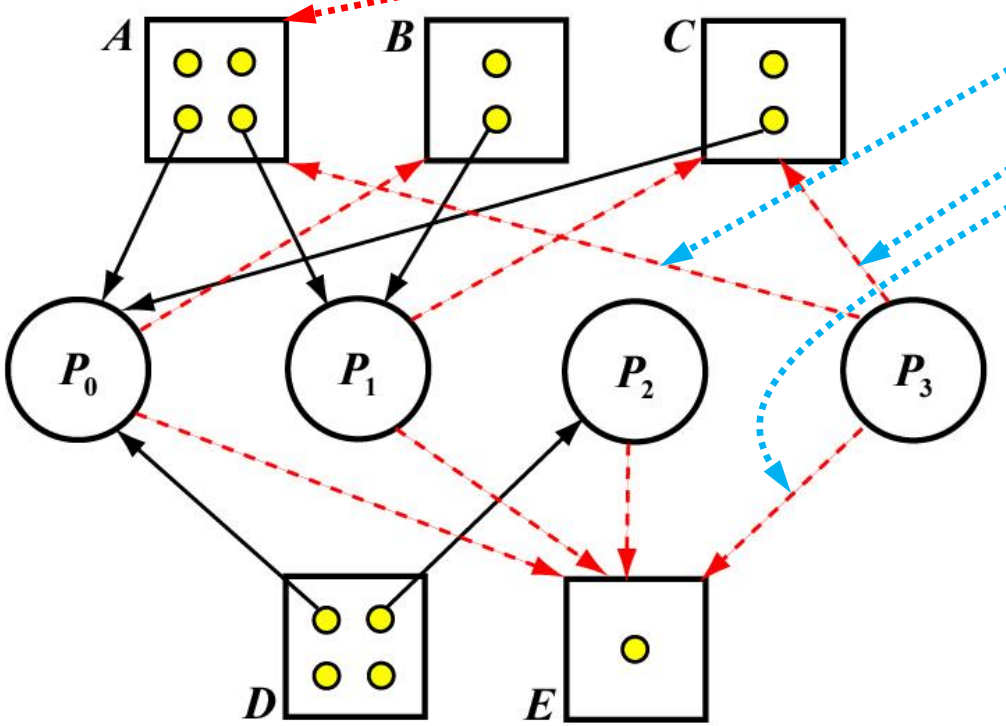
- **The right diagram has 3 processes P_1, P_2 and P_3 and 3 types of resources R_1, R_2 and R_3.**
- **Process P_1 has a unit of R_2 and requests a unit of R_1.**
- **Process P_2 has a unit of R_1 and a unit of R_2, and requests a unit of R_3.**
- **Process P_3 has a unit of R_3.**



Resource Allocation Graph: 4/9

- One can convert a table to a **RAG** and vice versa.

	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					

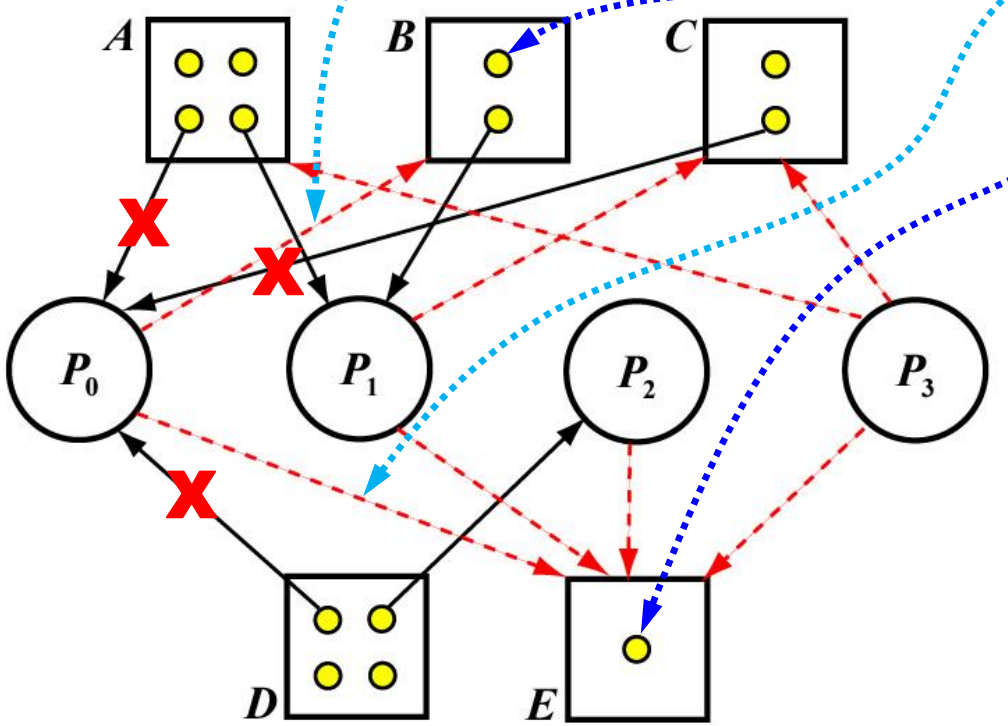


- ✓ The total number of units of each resource type can easily be computed.
- ✓ Dashed arrows are requests
- ✓ Solid arrows are allocations.

Resource Allocation Graph: 5/9

- **Both representations have the same meaning:**

	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					

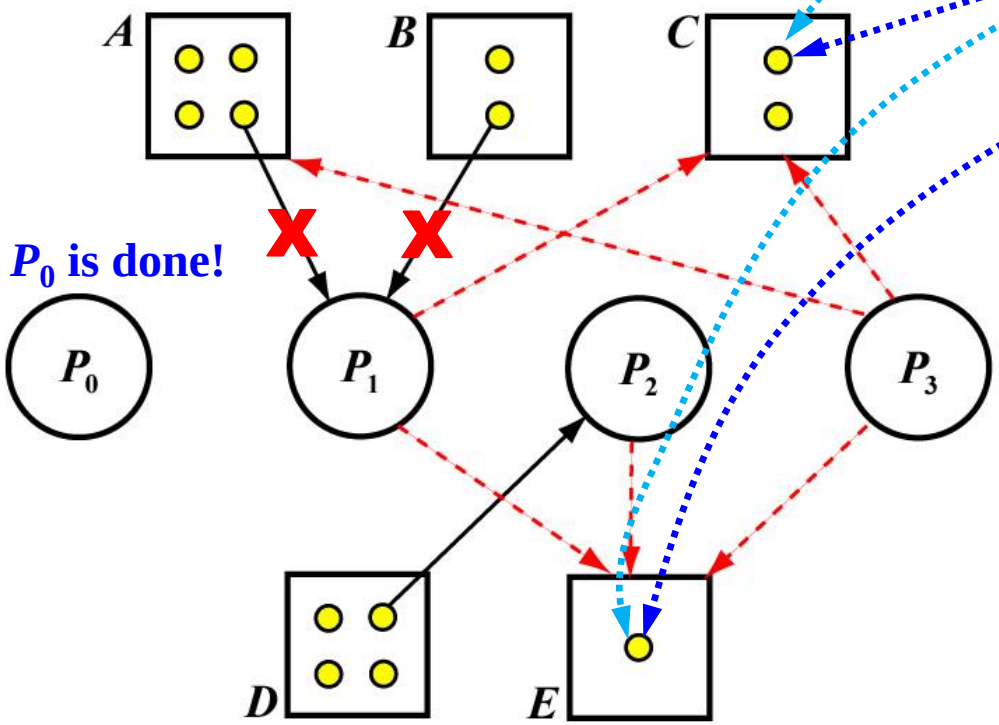


- ✓ Because P_0's *Request* ≤ *Available*, P_0 runs.
- ✓ Then, P_0 returns [1,0,1,1,0].
- ✓ The new *Available* = [2,1,1,2,1]+[1,0,1,1,0] = [3,1,2,3,1].
- ✓ Returning [1,0,1,1,0] means removing the directed edges pointing to P_0.

Resource Allocation Graph: 6/9

- **Both representations have the same meaning:**

	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					

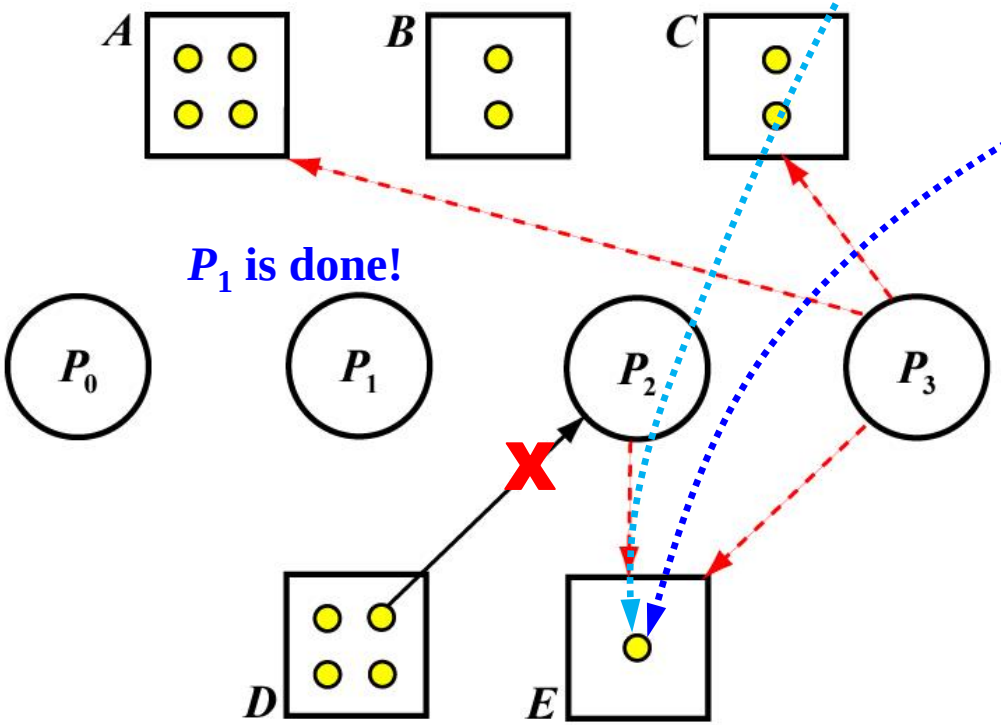


- ✓ Because P_1's *Request* ≤ *Available*, P_1 runs.
- ✓ Then, P_1 returns [1,1,0,0,0].
- ✓ The new *Available* = [3,1,2,3,1]+[1,1,0,0,0] = [4,2,2,3,1].
- ✓ Returning [1,1,0,0,0] means removing the directed edges pointing to P_1.

Resource Allocation Graph: 7/9

- **Both representations have the same meaning:**

	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					

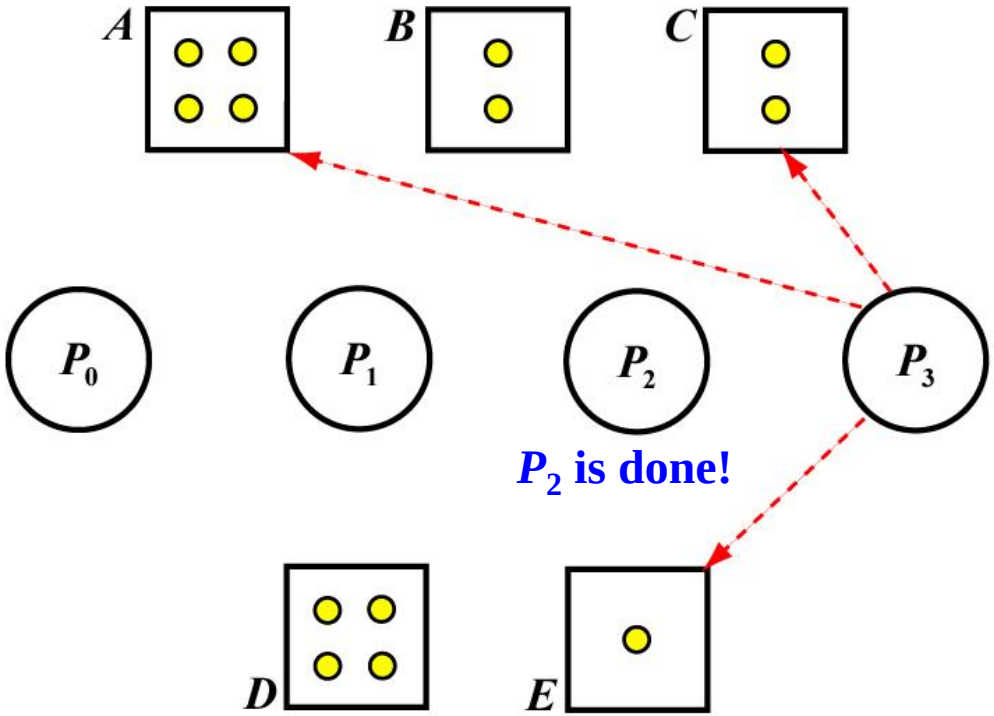


- ✓ Because P_2's *Request* ≤ *Available*, P_2 runs.
- ✓ Then, P_2 returns [0,0,0,1,0].
- ✓ The new *Available* = [4,2,2,3,1]+[0,0,0,1,0] = [4,2,2,4,1].
- ✓ Returning [0,0,0,1,0] means removing the directed edges pointing to P_2.

Resource Allocation Graph: 8/9

- **Both representations have the same meaning:**

	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					

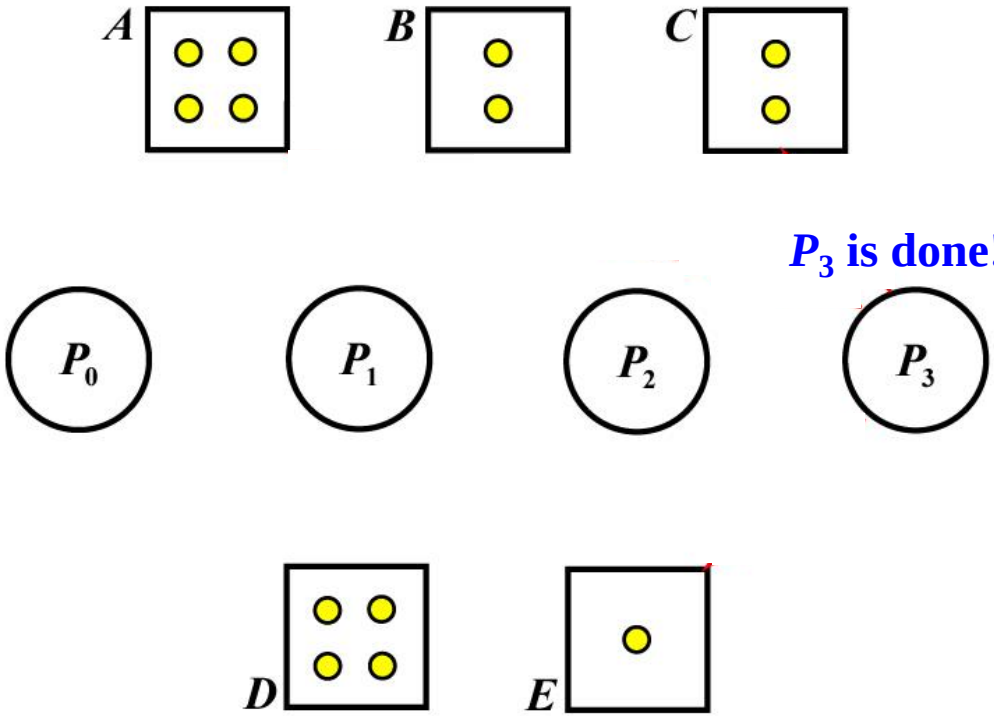


- ✓ Because P_3's *Request* ≤ *Available*, P_3 runs.
- ✓ Then, P_3 returns [0,0,0,0,0].
- ✓ The new *Available* = [4,2,2,4,1]+[0,0,0,0,0] = [4,2,2,4,1].
- ✓ Returning [0,0,0,0,0] means removing the directed edges pointing to P_3.

Resource Allocation Graph: 9/9

- **Both representations have the same meaning:**

	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					

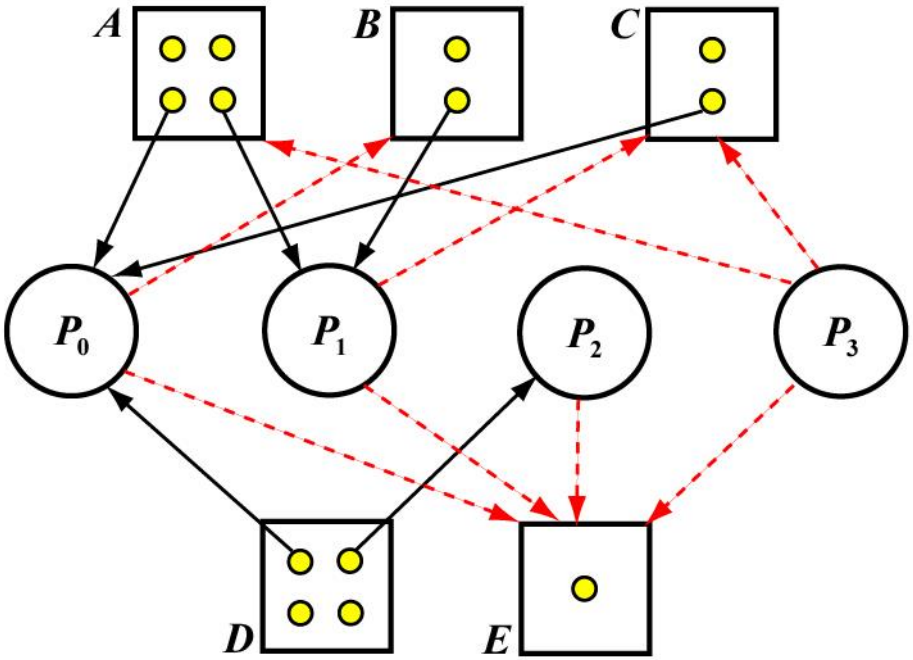


P_3 is done!

- ✓ Because P_3's *Request* ≤ *Available*, P_3 runs.
- ✓ Then, P_3 returns [0,0,0,0,0].
- ✓ The new *Available* = [4,2,2,4,1]+[0,0,0,0,0] = [4,2,2,4,1].
- ✓ Returning [0,0,0,0,0] means removing the directed edges pointing to P_3.

RAG Reduction

- Consider a process P_i, which is neither *blocked* nor an *isolated* node.
- A **RAG** is *reduced* by a process P_i by removing all edges to and from P_i.
- This reduction can be done *if* P_i can have all its requests satisfied (i.e., *Request* $\leq$ *Available*).
- A **RAG** is *completely reducible* if all edges of the graph can be removed by a sequence of reductions.



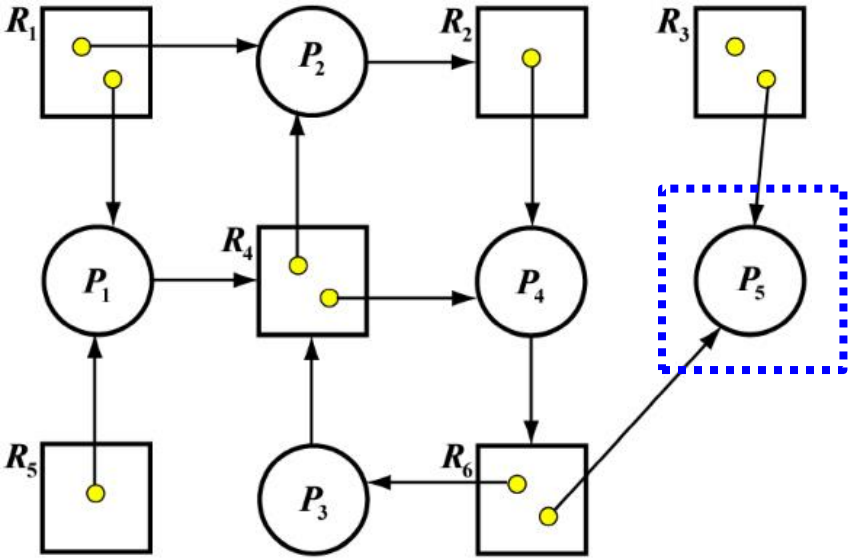
- P_0 is a valid reduction, because P_0's needs (*i.e.*, 1 unit of *B* and 1 unit of *E*) are available.
- P_1, P_2 and P_3 can also be good candidates.

The Deadlock Theorem: 1/8

- **The Deadlock Theorem**: **A state of resource allocation is a deadlock state if and only if its corresponding RAG is not completely reducible.**
- **The necessary conditions for having a deadlock implies that hold-and-wait and circular waiting must both hold.**
- **Therefore, this incompletely reducible RAG must contain a cycle (i.e., circular waiting).**
- **If a RAG does not contain a cycle, the state of resource allocation is deadlock-free.**
- **It is easy to determine whether a RAG contains a cycle (e.g., depth-first search).**

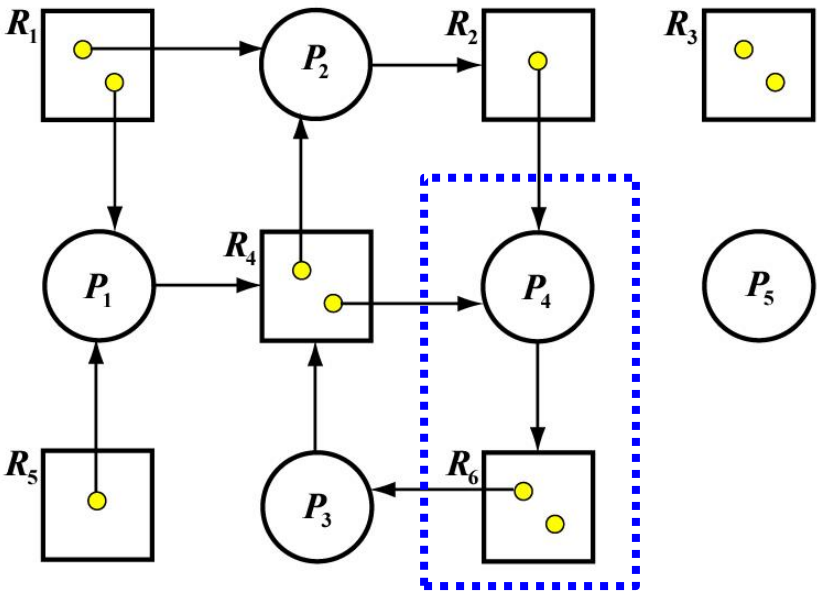
The Deadlock Theorem: 2/8

- Consider the right **RAG**.
- This **RAG** obviously can be reduced by process P_5, because P_5 has everything it needs.
- Therefore, all edges to and from process P_5 are removed.



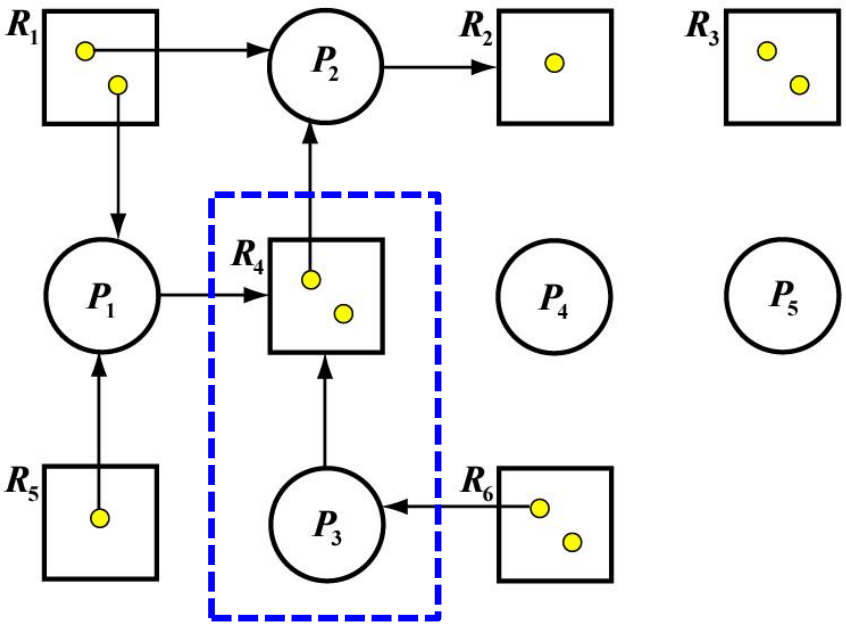
The Deadlock Theorem: 3/8

- After removing all edges from and to P_5, we have the right **RAG**.
- Now P_4 is a candidate, because it needs a unit of resource R_6, which is available.
- Then, all edges to and from process P_4 are removed.



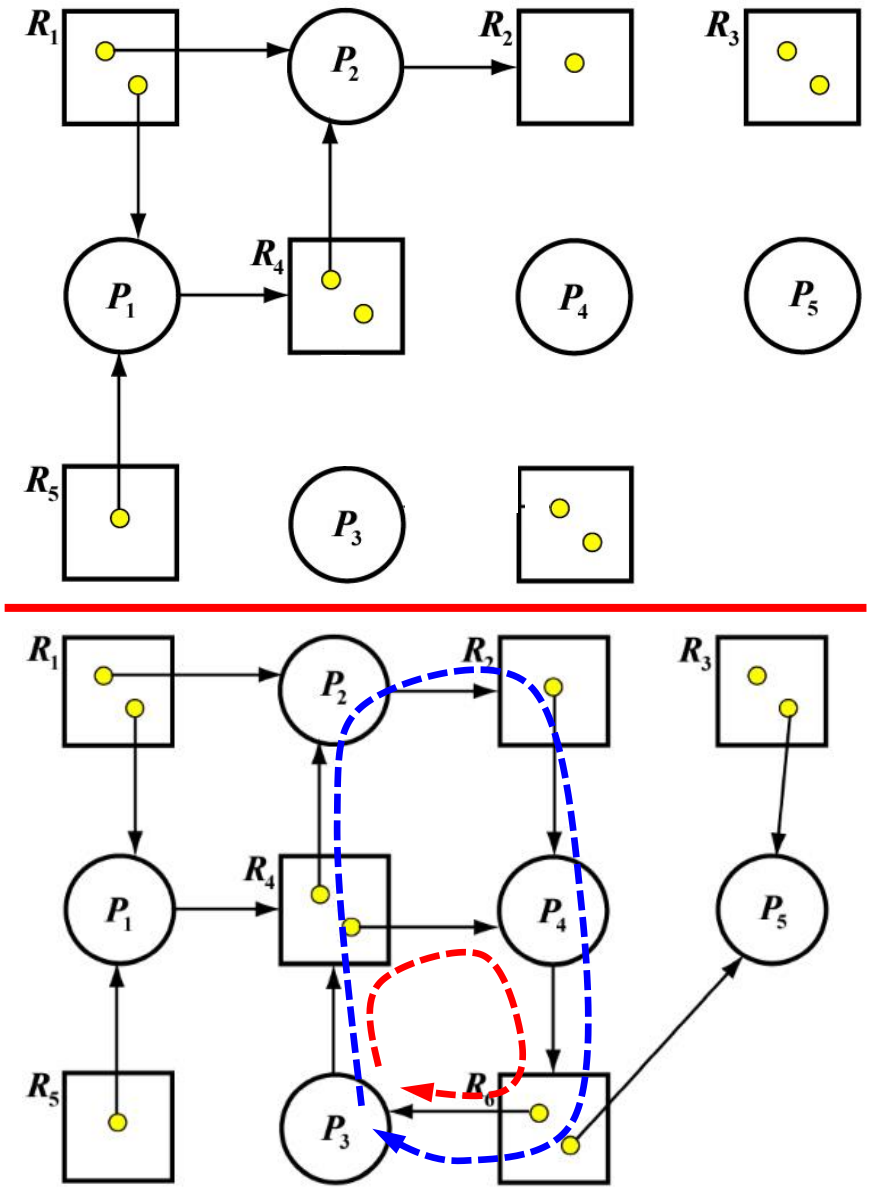
The Deadlock Theorem: 4/8

- After removing all edges from and to P_4, we have the right **RAG**.
- Now P_3 is a candidate, because it needs a unit of resource R_4, which is available.
- Then, all edges to and from process P_3 are removed.



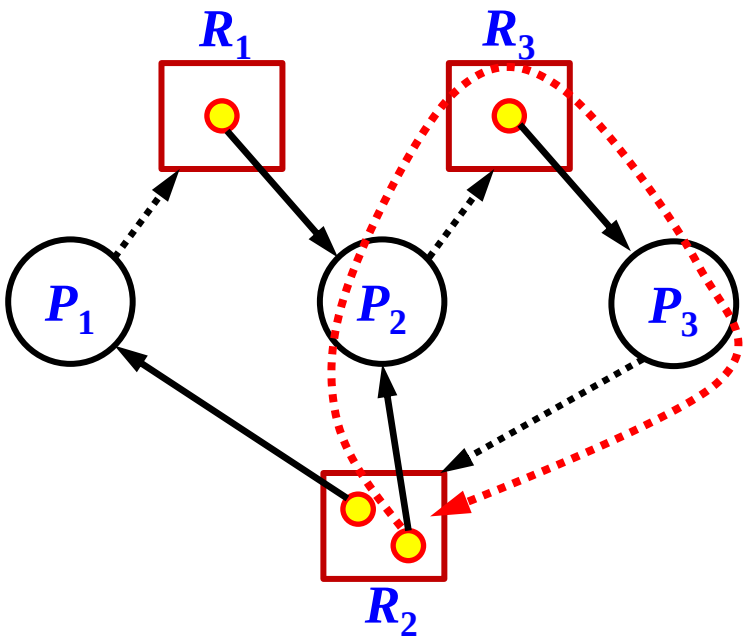
The Deadlock Theorem: 5/8

- Now, P_2 can take the only unit of R_2 and run.
- Then, process P_1 runs, and the reduction completes.
- Note that process P_1 can run because its only need is a unit of R_4, and this needed unit is available.
- This **RAG** is deadlock-free, but it contains cycles!



The Deadlock Theorem: 6/8

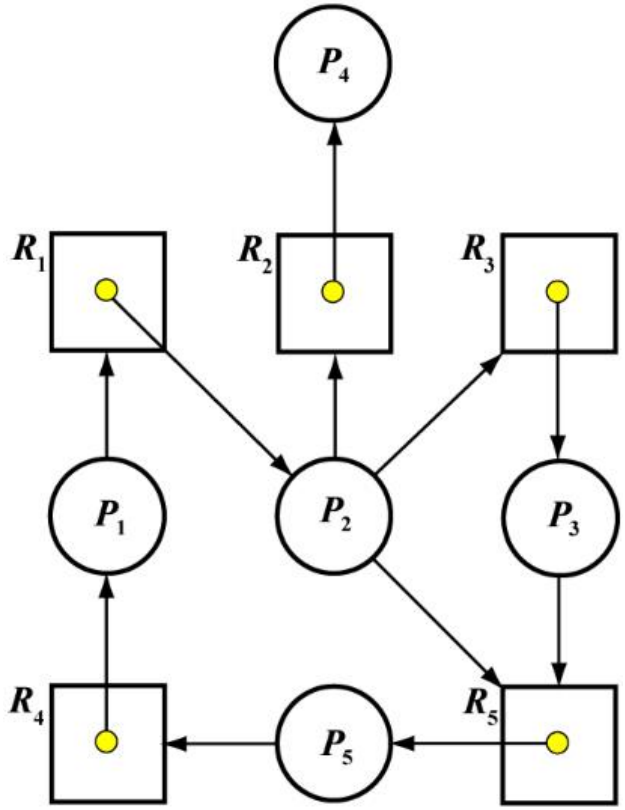
- P_3's request (1 unit of R_2) cannot be met.
- P_2's request (1 unit of R_3) cannot be met.
- P_1's request (1 unit of R_1) cannot be met.
- Because this **RAG** cannot be completely reduced, a deadlock exists.
- There is a cycle.



The existence of a cycle does not always imply a deadlock state as discussed on this and the previous slides.

The Deadlock Theorem: 7/8

- **Exercise**: Convert the following **RAG** to their table representations and do a step-by-step execution of the deadlock detection and **RAG** reduction. Is this deadlock-free?



The Deadlock Theorem: 8/8

- **What we can conclude is: if a RAG has no directed cycle, the corresponding allocation state is deadlock-free.**
- **If a RAG has a directed cycle, we have two cases:**
  - **If there is only one unit per resource type, there is a deadlock. This is a perfect circular waiting.**
  - **Otherwise, a deadlock is only a possibility. Use RAG reduction or deadlock detection to determine whether the system is in a deadlock state.**

The Use of a Detection Algorithm

- **The deadlock detection algorithm is not run when a request is made.**
- **It is run periodically when it is needed.**
  - **If deadlocks occur frequently, the detection algorithm should be invoked frequently.**
  - **Once per hour or whenever CPU utilization becomes low (*i.e.*, below 40%). Low CPU utilization usually means more processes are waiting.**

How to Recover

- **When a detection algorithm determines a deadlock has occurred, the algorithm may inform the system administrator to deal with it. Or, allow the system to *recover* from a deadlock.**
- **There are two options.**
  - **Process Termination**
  - **Resource Preemption**
- **These two options are not mutually exclusive (*i.e.*, can have both if needed).**

Process Termination

- **Abort all deadlocked processes**
- **Abort one process at a time until the deadlock cycle is eliminated**
- **Problems:**
  - **Aborting a process may not be easy. What if a process is updating or printing a large file? The system must find some way to maintain the states of the files and printer before they can be reused.**
  - **Termination may be determined by the priority/importance of a process.**

Resource Preemption

- **Selecting a victim: which resources and which processes are to be preempted?**
- **Rollback: If we preempt a resource from a process, what should be done with that process?**
  - **Total Rollback: abort the process and restart it**
  - **Partial Rollback: rollback the process only as far as necessary to break the deadlock.**
- **Starvation: We cannot always pick the same process as a victim. Some limit must be set.**

Livelock

Livelock: 1/3

- **Livelock: If two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work.**
- **These processes are not in the waiting state, and they are running concurrently.**
- **This is different from a deadlock because in a deadlock all processes are in the waiting state.**

Livelock: 2/3

```
MutexLock Mutex1, Mutex2;

Mutex1.Lock();                  // lock Mutex1
while (Mutex2.isLocked()) {     // loop until Mutex2 is open
   Mutex1.Unlock();             //    release Mutex1 (yield)
   // wait for a while          //    wait for a while
   Mutex1.Lock();               //    reacquire Mutex1
}                               // OK, Mutex2 is open
Mutex2.Lock();                  // lock Mutex2. have both
```

- **There are two locks, a process locks its own lock first.**
- **Then, checks whether the other lock is locked.**
  - **If the other lock is not locked, this process locks it.**
  - **Otherwise, this process releases its lock and locks its own lock again later**

Livelock: 3/3

```
MutexLock Mutex1, Mutex2;
```

Process 1

```
Mutex1.Lock();
while (Mutex2.isLocked()) {
    Mutex1.Unlock();
    // wait for a while
    Mutex1.Lock();
}
Mutex2.Lock();
```

Process 2

```
Mutex2.Lock();
while (Mutex1.isLocked()) {
    Mutex2.Unlock();
    // wait for a while
    Mutex2.Lock();
}
Mutex1.Lock();
```

- What if both processes execute in a fully synchronized way?
- We have a **livelock**, each process reacts to the other's activity.
- Nothing meaningful is achieved.
- To avoid this type of livelock, **order the locking sequence in a hierarchical way** (i.e., both lock `Mutex1` **first** followed by `Mutex2`.
- Can we combine both locks into only one lock?

The End