

Deadlocks

Definition

- **A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.**
- **An event could be:**
 - **Waiting for a critical section**
 - **Waiting for a condition to change**
 - **Waiting for a physical resource**

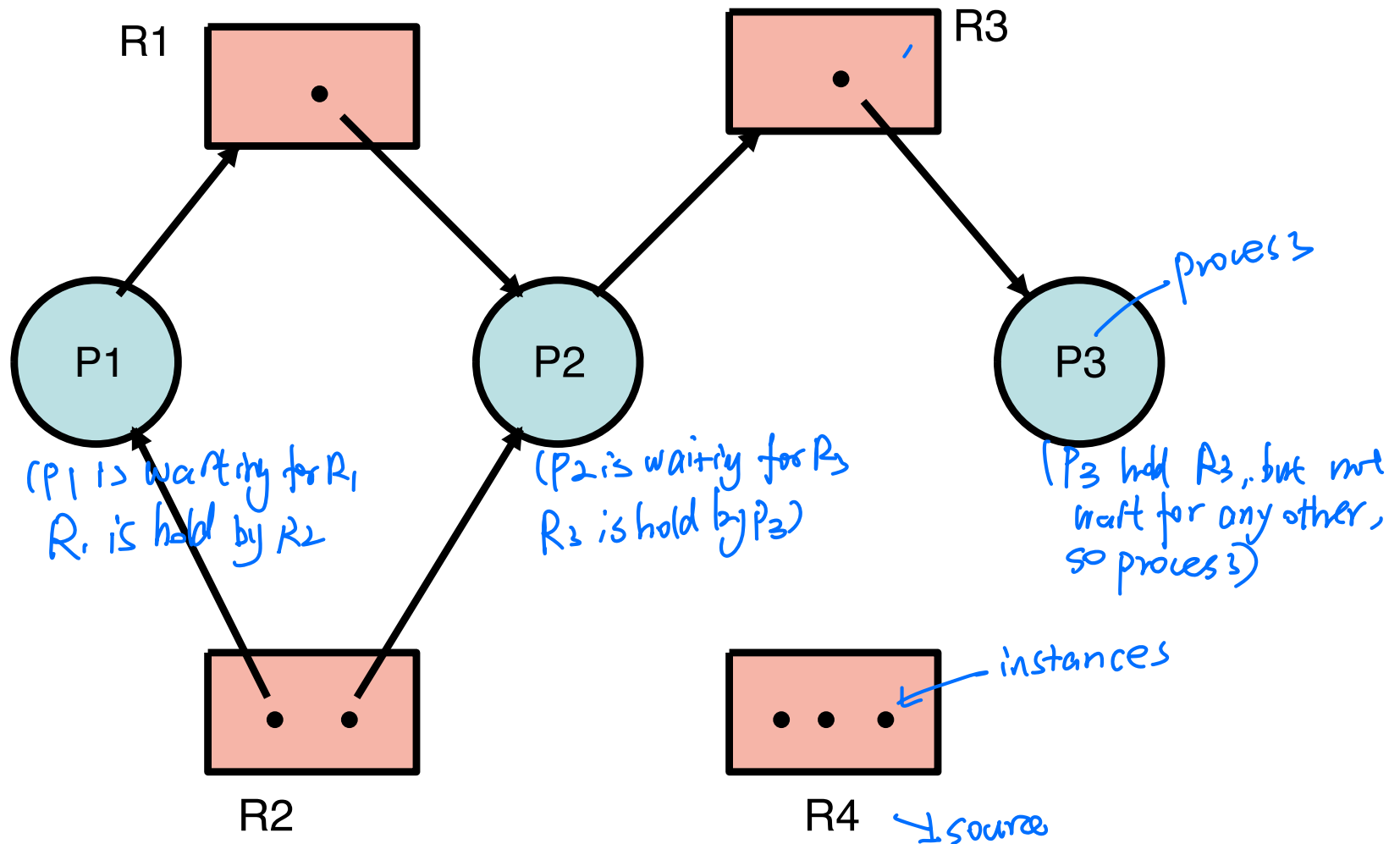
Necessary Conditions for a deadlock

- **Mutual exclusion:** The requesting process is delayed until the resource held by another is released.
— dining philosopher
- **Hold and wait:** A process must be holding at least 1 resource and must be waiting for 1 or more resources held by others.
- **No preemption:** Resources cannot be preempted from one and given to another.
- **Circular wait:** A set (P_0, P_1, \dots, P_n) of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for ... by P_2 , ... P_n is waiting for ... held by P_0 .

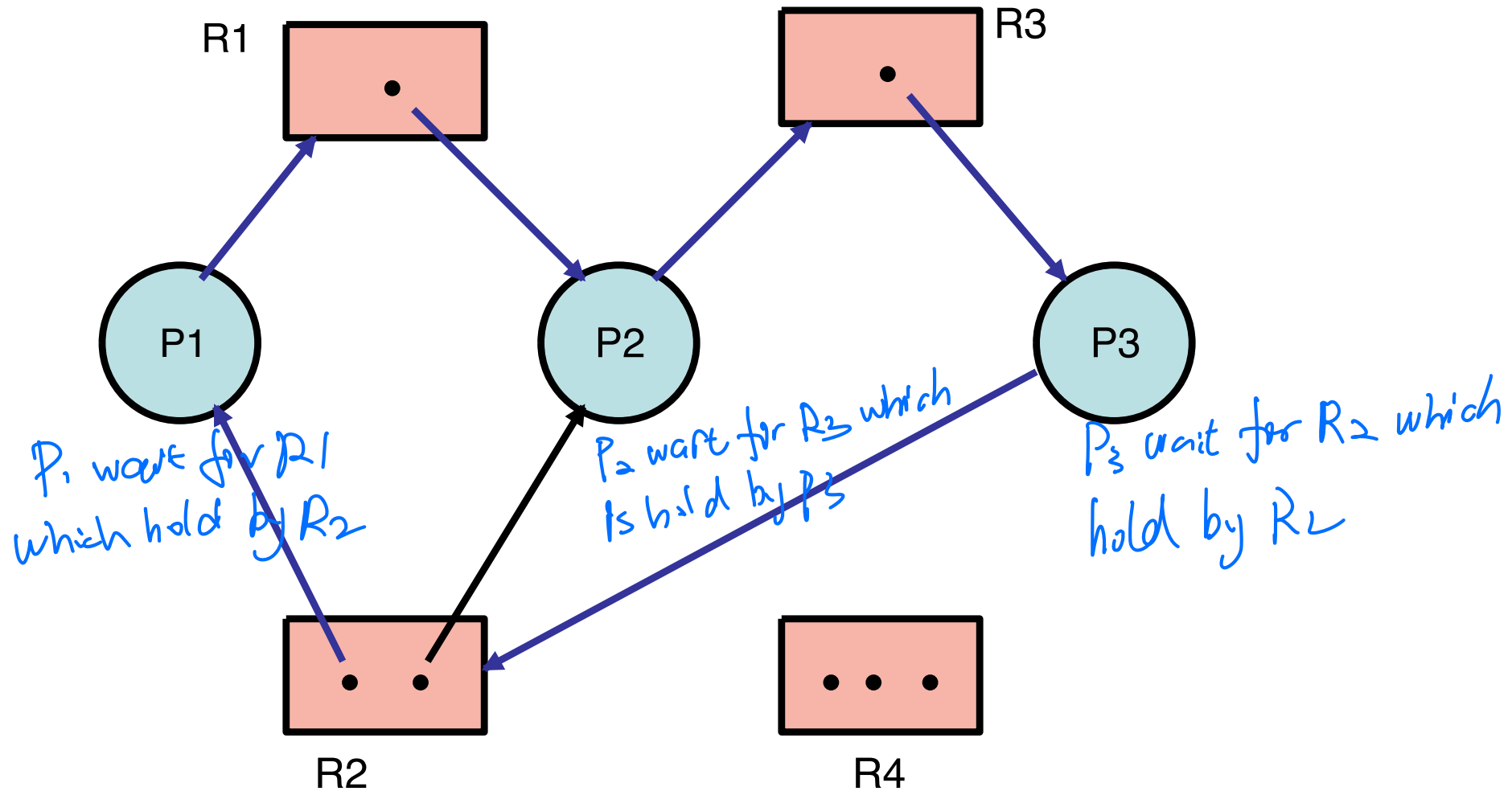
Resource Allocation Graph

- **Vertices (V) = Processes (P_i) and Resources (R_j)**
- **Edges (E) = Assignments ($R_j \rightarrow P_i$, R_j is allocated to P_i) and Request ($P_i \rightarrow R_j$, P_i is waiting for R_j).**
- **For each Resource R_j , there could be multiple instances.**
- **A requesting process can be granted any one of those instances if available.**

An example

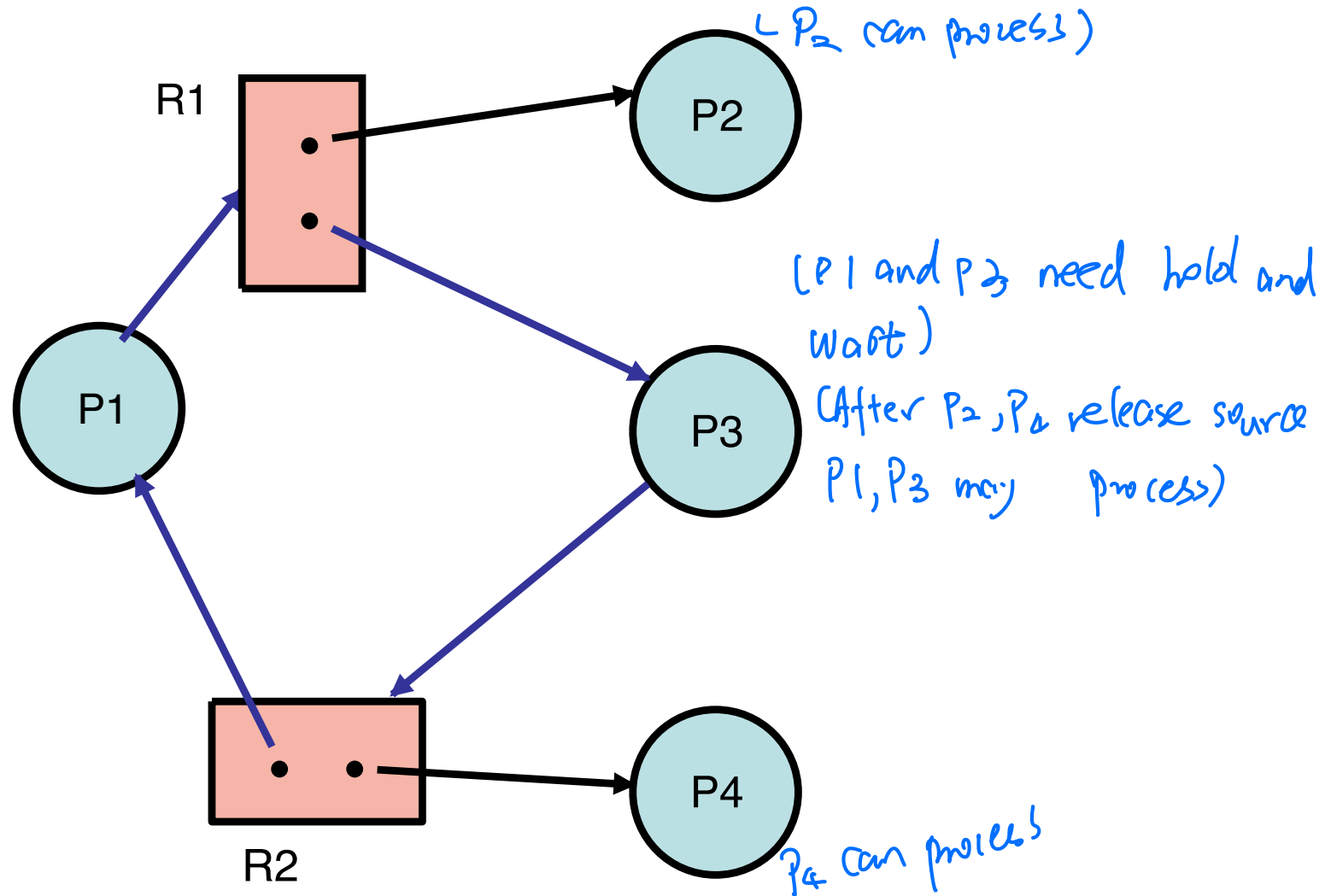


A deadlock



If there is a deadlock, there will be a cycle (Necessary Condition).

A cycle is NOT a sufficient condition



What is a sufficient condition for a deadlock?

A Knot

The reachability set of every node in the set is the set itself.

Strategies for Handling Deadlocks

- Ignore the problem altogether (ostrich algorithm) since it may occur very infrequently, cost of detection/prevention may not be worth it.
- **Detect and recover** after its occurrence.
- **Avoidance** by careful resource allocation
- ➔ • **Prevention** by structurally negating one of the four necessary conditions

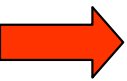
Deadlock Prevention

- **Note that all 4 necessary conditions need to hold for deadlock to occur.**
- **We can try to disallow one of them from happening:**
 - **Mutual exclusion:** This is usually not possible to avoid with many resources.
 - **No preemption:** This is again not easy to address with many resources. Possible for some resources (e.g., CPU)
 - **Hold and Wait:**
 - **Allow at most 1 resource to be held/requested at any time**
 - **Make sure all requests are made at the same time.**
 - ...

– Circular Wait

- **Number the resources, and make sure requests are always made in increasing/decreasing order.**
- **Or make sure you are never holding a lower numbered resource when requesting a higher numbered resource.**

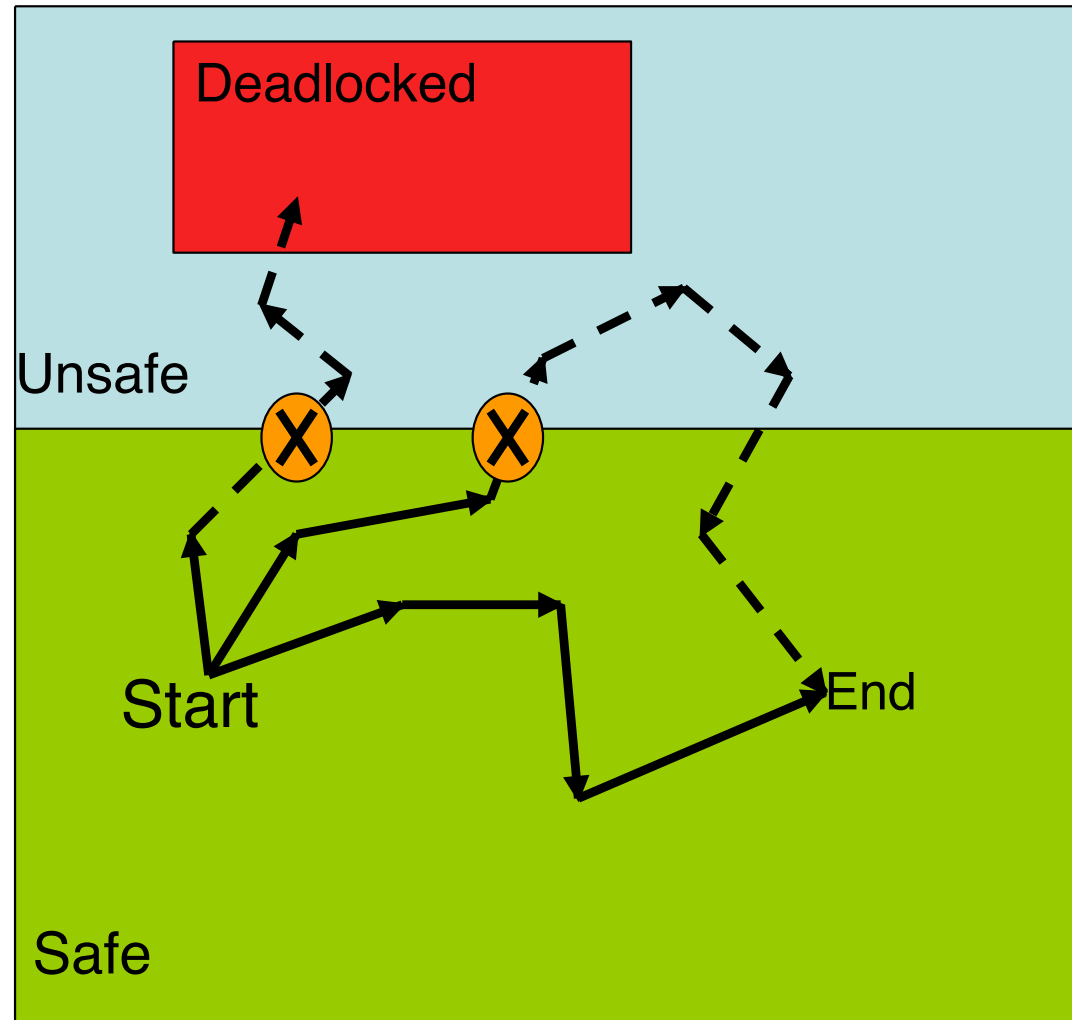
Strategies for Handling Deadlocks

- **Ignore the problem altogether (ostrich algorithm) since it may occur very infrequently, cost of detection/prevention may not be worth it.**
- **Detect and recover after its occurrence.**
-  • **Avoidance by careful resource allocation**
- **Prevention by structurally negating one of the four necessary conditions**

Deadlock Avoidance

- Avoid actions that may lead to a deadlock.
- Visualize the system as a state machine moving from 1 state to another as each instruction is executed.
- A state can be: **safe**, **unsafe** or **deadlocked**.
- Safe state is one where
 - it is not a deadlocked state
 - there is some sequence by which all requests can be satisfied.
- To avoid deadlocks, we try to make only those transitions that will take you from one safe state to another.
- This may be a little conservative, but it avoids deadlocks

State Transitions



Safe State

1 resource with 12 units of that resource available.

Current State: $\text{Free} = (12 - (5 + 2 + 2)) = 3$

	Max. Needs	Currently Allocated	Still May Need
P0	10	5	5
P1	4	2	2
P2	9	2	7

Free = 3

① full fill P1 first

② After reducing P1,
Free = 5 *P1 finish*

After reducing P0,

③ **Free = 10**
Then reduce P2.

This state is **safe** because, there is a sequence (P1 followed by P0 followed by P2) by which max needs of each process can be satisfied.

This is called the **reduction sequence**.

Unsafe State

What if P2 requests 1 more and is allocated 1 more?

New State:

	Max. Needs	Currently Allocated	Still May Need
P0	10	5	5
P1	4	2	2
P2	9	3	6

Free = 2

This is unsafe.

After P1 goes, we only have 4 resources which is not enough for any other process.

Only P1 can be reduced. If P0 and P2 then come and ask for their full needs, the system can become deadlocked.

Hence, by granting P2's request for 1 more, we have moved from a safe to **unsafe** state.

Deadlock avoidance algorithm will NOT allow such a transition, and will not grant P2's request immediately.

- **Deadlock avoidance essentially allows requests to be satisfied only when the allocation of that request would lead to a safe state.**
- **Else do not grant that request immediately.**

Banker's algorithm for deadlock avoidance

- **When a request is made, check to see if after the request is satisfied, there is a (at least one!) sequence of moves that can satisfy all possible requests, i.e., the new state is safe.**
- **If so, satisfy the request, else make the request wait.**

Checking if a state is safe (Generalization for “M” resources)

N processes and M resources

Data Structures:

MaxNeeds[N][M];

Allocated[N][M];

StillMayNeed[N][M];

Free[M];

Temp[M];

Done[N];

```
while () {
    Temp[j]=Free[j] for all j
    Find an i such that
        a) Done[i] = False
        b) StillMayNeed[i,j] <= Temp[j]
    if so {
        Temp[j] += Allocated[i,j] for all j
        Done[i] = TRUE
    }
    else if Done[i] = TRUE for all i then state is safe
    else state is unsafe
}
```

$M \cdot N^2$ steps to detect if a state is safe!

An example

5 processes, 3 resource types A (10 instances), B (5 instances), C (7 instances)

MaxNeeds				Allocated				StillMayNeed				Free		
	A	B	C		A	B	C		A	B	C	A	B	C
P0	7	5	3	P0	0	1	0	P0	7	4	3	3	3	2
P1	3	2	2	P1	2	0	0	P1	1	2	2	P1 5	3	2
P2	9	0	2	P2	3	0	2	P2	6	0	0	P3 7	4	3
P3	2	2	2	P3	2	1	1	P3	0	1	1	P0 7	5	3
P4	4	3	3	P4	0	0	2	P4	4	3	1	P2 10	5	5
												P4 10	5	7

This state is safe, because there is a reduction sequence $\langle P1, P3, P4, P2, P0 \rangle$ that can satisfy all the requests.

Exercise: Formally go through each of the steps that update these matrices for the reduction sequence.

If P1 requests 1 more instance of A and 2 more instances of C
 can we safely allocate these? – Note these are all allocated together!
 and we denote this set of requests as (1,0,2)

If allocated the resulting state would be:

MaxNeeds				Allocated				StillMayNeed				Free			
	A	B	C		A	B	C		A	B	C	A	B	C	
P0	7	5	3	P0	0	1	0	P0	7	4	3	2	3	0	
P1	3	2	2	P1	3	0	2	P1	0	2	0	P1	5	3	2
P2	9	0	2	P2	3	0	2	P2	6	0	0	P3	7	4	3
P3	2	2	2	P3	2	1	1	P3	0	1	1	P4	7	4	5
P4	4	3	3	P4	0	0	2	P4	4	3	1	P0	7	5	5
												P2	10	5	7

This is still safe since there is a reduction sequence $\langle P1, P3, P4, P0, P2 \rangle$
 to satisfy all the requests. (work this out!)
 Hence the requested allocations can be made.

After this allocation, P0 then makes a request for (0,2,0).
If granted the resulting state would be:

MaxNeeds				Allocated				StillMayNeed				Free		
	A	B	C		A	B	C		A	B	C	A	B	C
P0	7	5	3	P0	0	3	0	P0	7	2	3	2	1	0
P1	3	2	2	P1	3	0	2	P1	0	2	0			
P2	9	0	2	P2	3	0	2	P2	6	0	0			
P3	2	2	2	P3	2	1	1	P3	0	1	1			
P4	4	3	3	P4	0	0	2	P4	4	3	1			

This is an UNSAFE state.

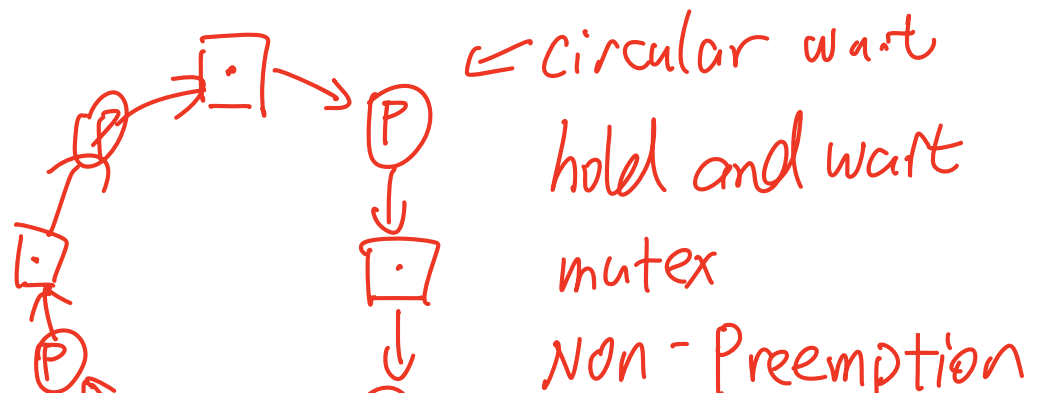
So, this request should NOT be granted.

Strategies for Handling Deadlocks

- Ignore the problem altogether (ostrich algorithm) since it may occur very infrequently, cost of detection/prevention may not be worth it.
- ➔ • Detect and recover after its occurrence.
- Avoidance by careful resource allocation
- Prevention by structurally negating one of the four necessary conditions

Deadlock Detection and Recovery

- If there is only 1 instance of each resource, then a cycle in the resource-allocation graph is a “sufficient” condition for a deadlock, i.e., you can run a cycle-detection algorithm to detect a deadlock.
- With multiple instances of each resource, ???



Detection Algorithm

N processes, M resources

Data structures:

Free[M];

Allocated[N][M];

Request[N][M];

Temp[M];

Done[N];

Basic idea is that there is at least 1 execution which will unblock all processes.

M*N² algorithm!

1. Temp[i] = Free[i] for all i
Done[i] = FALSE unless there is no resources allocated to it.
2. Find an index i such that both
 - (a) Done[i] == FALSE
 - (b) Request[i] <= Temp (vector comp.)
 If no such i, go to step 4.
3. Temp = Temp + Allocated[i] (vector add)
Done[i] = TRUE;
Go to step 2.
4. If Done[i] = FALSE for some i, then there is a deadlock.

Example

5 processes, 3 resource types A (7 instances), B (2 instances), C (6 instances)

Allocated

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2

Request

	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	0	0	2

Free

A	B	C
0	0	0

P0 0 1 0
 P2 3 1 3
 P3 5 2 4
 P1 7 2 4
 P4 7 2 6

This state is NOT deadlocked.

By applying algorithm, the sequence <P0, P2, P3, P1, P4> will result in Done[i] being TRUE for all processes.

If on the other hand, P2 makes an additional request for 1 instance of C

State is:

Allocated

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2

Request

	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	0	0	2

Free

A	B	C
0	0	0

P0 0 1 0
X X X

This is deadlocked!

Even though P0 can proceed, the other 4 processes are deadlocked.

Recovery

- Once deadlock is detected what should we do?
 - Preempt resources (whenever possible)
 - Kill the processes (and forcibly remove resources)
 - Checkpoint processes periodically, and roll them back to last checkpoint (relinquishing any resources they may have acquired since then).