`Deadlock and Starvation` slide deck

toc:

- Housekeeping
- Principles of Deadlocks
    - Deadlock
- Deadlock in Multithreaded Applications
- Deadlock Characterization
    - Resource Allocation Graph
- Methods for Handling Deadlocks
    - Prevention
        - Deny Hold and Wait
        - Deny No Preemption
        - Deny Circular Wait
    - Deadlock Avoidance
        - Safe State
        - Avoidance Algos
            - Resource-Allocation Graph Scheme
            - Banker's Algorithm
- Recovery from Deadlock

Housekeeping

Strike won't be affecting us too much

Principles of Deadlocks

Systems have different resources

Threads must:

- request a resource before using it
- release the resource after using it

The number of resources requested cannot exceed the total number of resources available in the system

- a thread cannot request 2 net interface if there is only 1
- `can't ask for what we don't have`

Request:

- Thread requests the resource
- if you can give it, then give it immediately
- If it can’t be given right now then the requesting thread must wait until it can acquire the resource

Use:

- the thread can operate on the resource
- ex: use the mutex lock to access a process’ critical section

Release:

- resource is released and made usable again

Reusable Resources:

- can be used safely by one process at a time
- not depleted by that use
- not consumable
- ex: proessors, I/O channels, memory (main and secondary), I/O devices, data structures (files, databases, and semaphores)

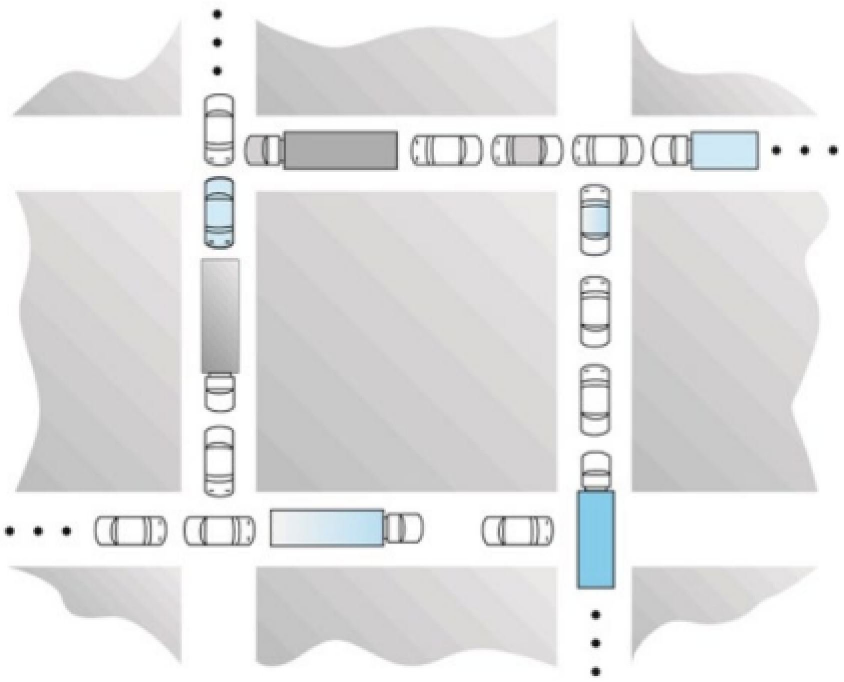
Consumable Resources:

- created/produced and destroyed/consumed
- ceases to exist after being acquired by the consuming process
- ex: interrupts, singals, messages, info in I/O buffers

Deadlock

A deadlock is the **permanent** block.

Two or more processes are waiting for the other process to release a shared resource.



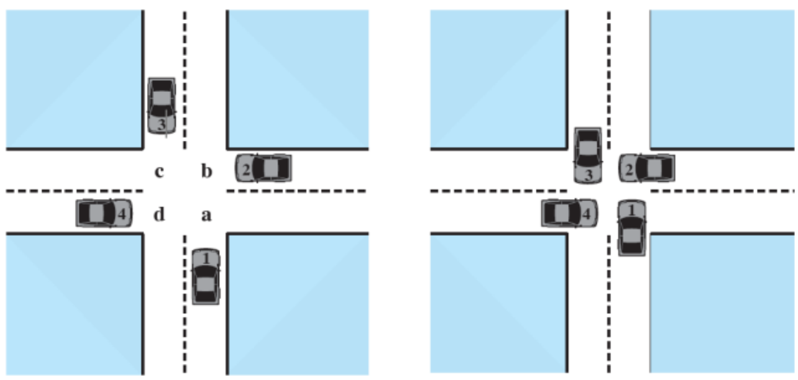
Traffic Deadlock

A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set.

The block is permanenet because none of the events ever get triggered.

Illustration of Deadlock

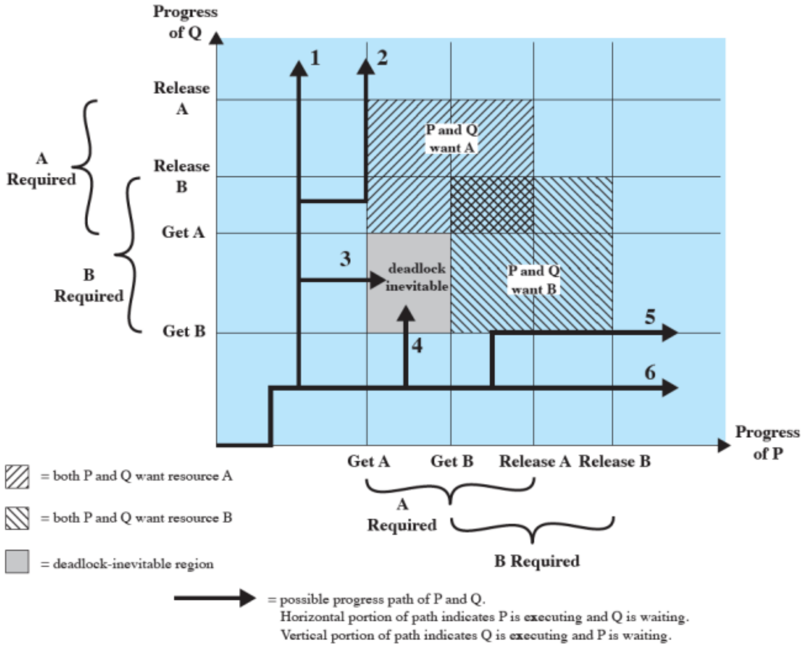
- All deadlocks involve conflicting needs for resources by two or more processes.



(a) Deadlock possible (b) Deadlock

Joint Progress Diagram – Deadlock

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •



Trace 1 sees process Q execute all of its desire operations before process P can perform any single operation

Trace 2 sees process Q get A then make process P wait until it was done.

Trace 3 sees Q get B and P get A. 4 does the same but in reverse order

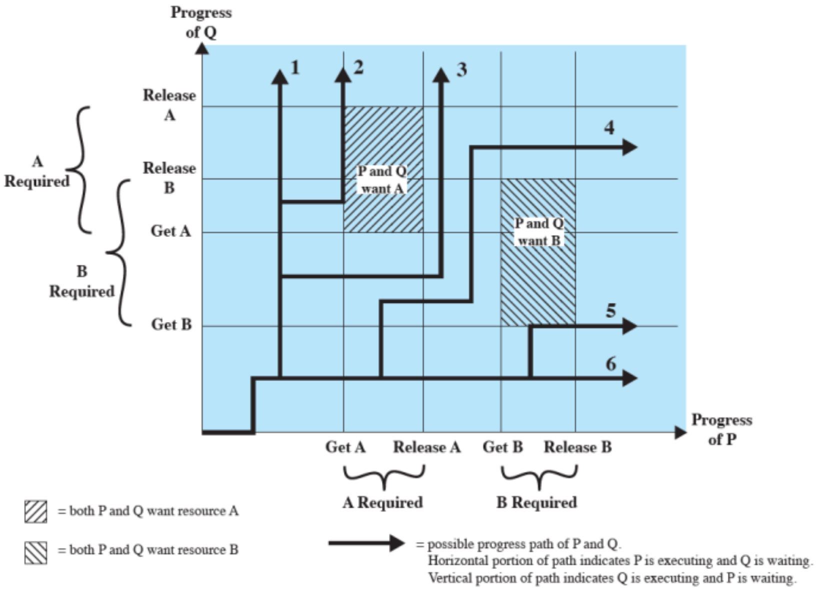
Traces 3 & 4 eventually leads to a deadlock as each process will want the other's acquired resource that they can't release until they get the resource they want.

Trace 5 sees process P get B then make process Q wait until it's done.

Trace 6 sees P exectute everything before Q gets make some meaningful progress.

Joint Progress Diagram – No Deadlock

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •



Joint Progress Diagram

> "I would like to leave that for the banking algorithm" - Prof, no one knows what she meant by that

This rearrangement is not always a possible solution since we don't know what the order of the instructions/operations will be beforehand.

Traces 1, 2, 5, and 6 are unaffected but now traces 3 and 4 don't lead to deadlock.

instead with Trace 3

- Q gets B
- P gets A then releases A
- Q gets A, releases B, then releases A
- P gets B then releases B

Similarly with trace 4

- 2 processes that compete for exclusive access to a disk file **D** and a tape drive **T**.
- **Questions**: Are these reusable or consumable resources? Will there be a deadlock? What sequence of execution will result in a deadlock?

Step	Process P Action	Step	Process Q Action
p_0	Request (D)	q_0	Request (T)
p_1	Lock (D)	q_1	Lock (T)
p_2	Request (T)	q_2	Request (D)
p_3	Lock (T)	q_3	Lock (D)
p_4	Perform function	q_4	Perform function
p_5	Unlock (D)	q_5	Unlock (T)
p_6	Unlock (T)	q_6	Unlock (D)

These are reusable resources.

There will be a deadlock.

$p_0 q_0 p_1 q_1$ introduces the possibility of a deadlock

$p_0 q_0 p_1 q_1 p_2 q_2$ is where the deadlock happens.

- Memory Space is available for allocation of **200Kbytes**, and the following sequence of requests occur.
- **Question**: Are these reusable or consumable resources? When will there be a deadlock?

P1	P2
...	...
Request 80 Kbytes;	Request 70 Kbytes;
...	...
Request 60 Kbytes;	Request 80 Kbytes;

These are reusable resources.

There will be a deadlock.

P1 and P2 can make their requests then the next requests will cause a deadlock.

P1 or P2 can make both requests but then upon the next request from the other process we will run into deadlock. That is unless there is some kind of release condition that we don’t see beyond the processes’ second requests.

The cause of the deadlock being that we don’t have enough resources to give.

- Each process is attempting to receive a message from the other process and **then** send a message to the other process:
- **Question**: Are these reusable or consumable resources? Will there be a deadlock?

P1	P2
...	...
Receive (P2);	Receive (P1);
...	...
Send (P2, M1);	Send (P1, M2);

Consumable resource.

There is a deadlock as they are both waiting on each other.

If receive is not blocking - i.e. the processes just make themself open to receiving and don't wait - then there is no deadlock.

It depends on the nature of the receive operation.

```
Prof will tell us if it's blocking or not blocking.
```

Deadlock in Multithreaded Applications

2 mutex locks are created and initialized:

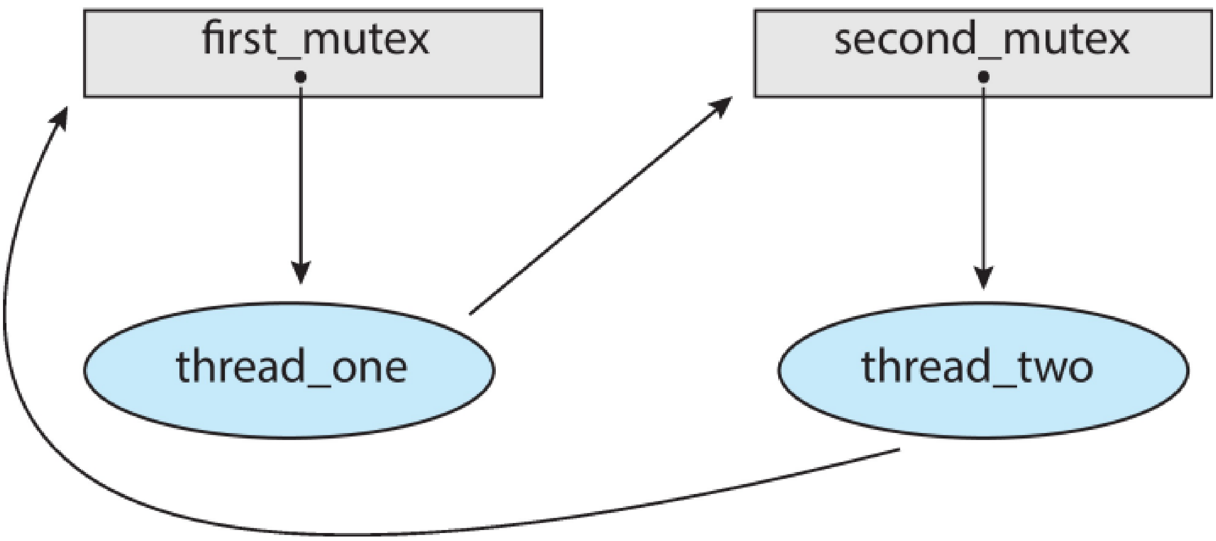
```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

2 threads are created and both threads have access to both mutex locks.

```c
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    pthread_mutex_lock(&first_mutex);
    pthread_mutex_lock(&second_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&second_mutex);
    pthread_mutex_unlock(&first_mutex);

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

Deadlock is possible if thread 1 acquires `first_mutex` and thread 2 acquires `second_mutex`

Each thread then waits for the other's mutex.



THe order depends on how the threads are scheduled by the cpu scheduler. hard to test for deadlocks as they may only occur under certain scheduling circumstances.

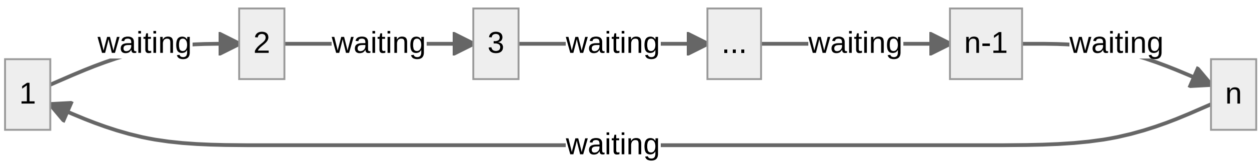
this is the livelock of waiting continually trying to acquire knowledge check:

- a deadlocked state occurs whenever
  - every process in a set is waiting for an event that can only be cauesd by another process in the set
- Deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set, while livelock occurs when a thread continuously attempts an action that fails.
  - true
- in the dining philosophers problem, there is a possibility of deadlock but not livelock
  - false
  - everyone picks a fork up then there is deadlock
  - if they keep picking it up and putting it down then there is livelock

Deadlock Characterization

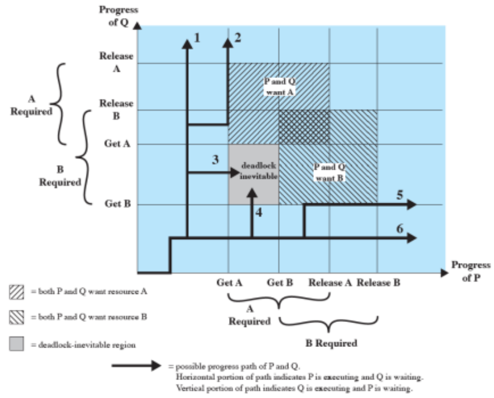
deadlock can arise if 4 conditions hold at the same time:

- mutual exclusion
  - only one process at a time can use a resource
- hold and wait
  - a process holding at least one resource is waiting to acquire additional resources held by other processes
- no preemption (for resources no the process)
  - resource only released by the process holding it after it's done its task
  - resource cannot be preempted
    - processor can't take resources from process and give it to other processes arbitrarily
- circular wait
  - process 1 is waiting on process 2
  - process 2 is waiting on process 3
  - …
  - process n-1 is waiting on process n
  - process n is waiting on process 1



- **All four conditions must hold** for a deadlock to occur.

Possibility of Deadlock	**Existence of Deadlock**
1.Mutual exclusion 2.No preemption 3.Hold and wait	1.Mutual exclusion 2.No preemption 3.Hold and wait 4.Circular wait



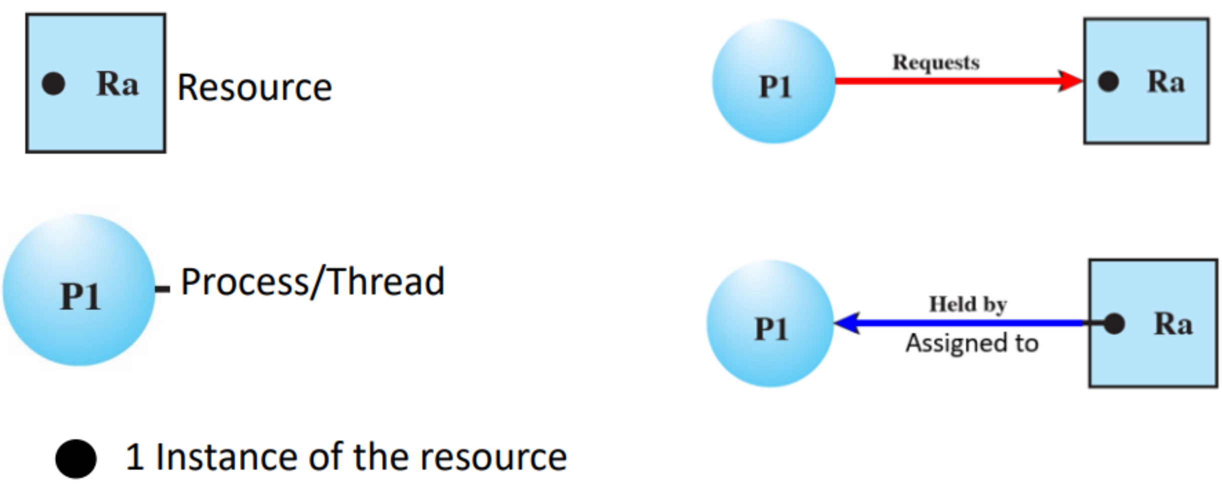
Resource Allocation Graph

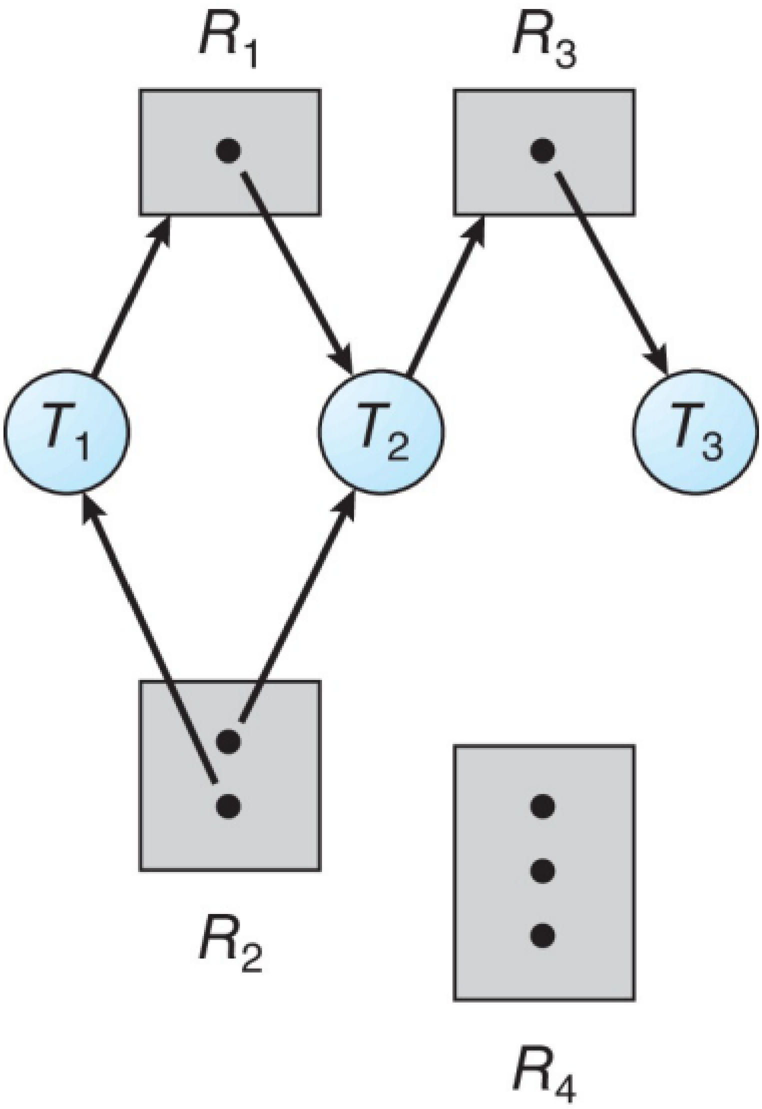
- Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph.
  - A set of vertices *V* and a set of edges *E*.
  - V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system

A digraph with vertices V partitioned into P processes and R resources

request edge – directed edge $P_i \rightarrow R_j$

assignment edge – directed edge $R_j \rightarrow P_i$





There are 3 threads and 4 resources.

.	T_1	T_2	T_3
R_1	req	assigned	
R_2	assigned	assigned	
R_3		req	assigned
R_4			

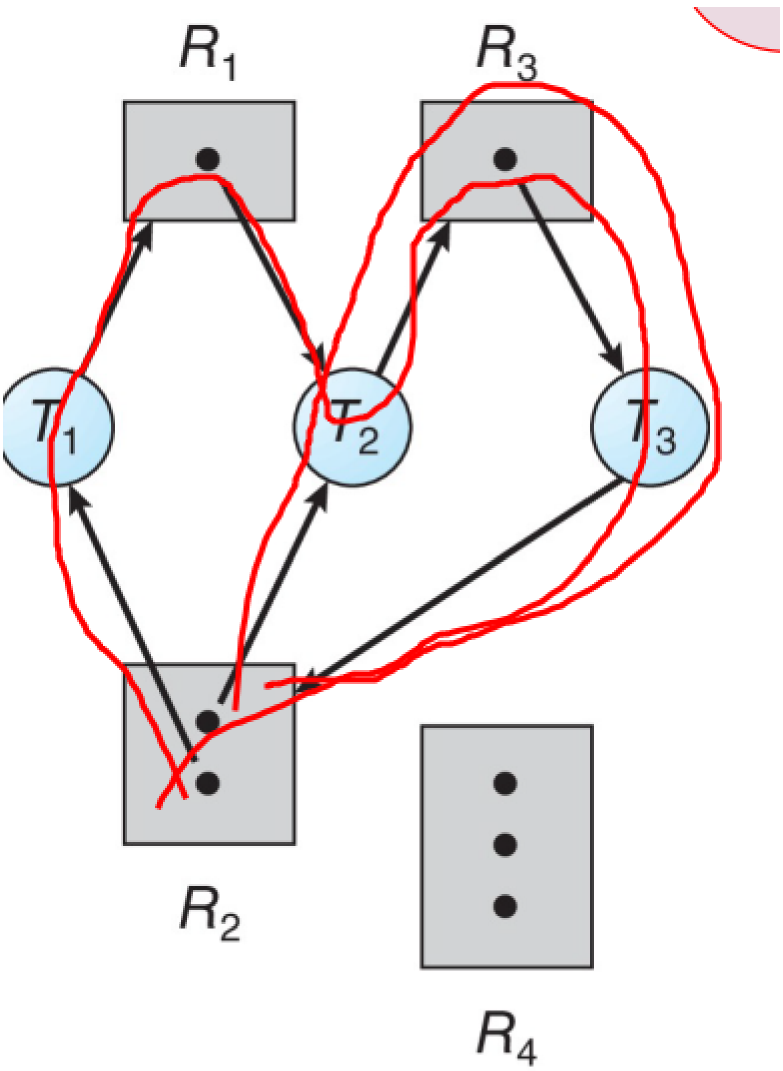
There are 1 instance of R_1 and R_2

2 of R_2 and 3 of R_4

There are no cycles in the graph.

Therefore there are no deadlocks

If there is no cycle then there is no deadlock.



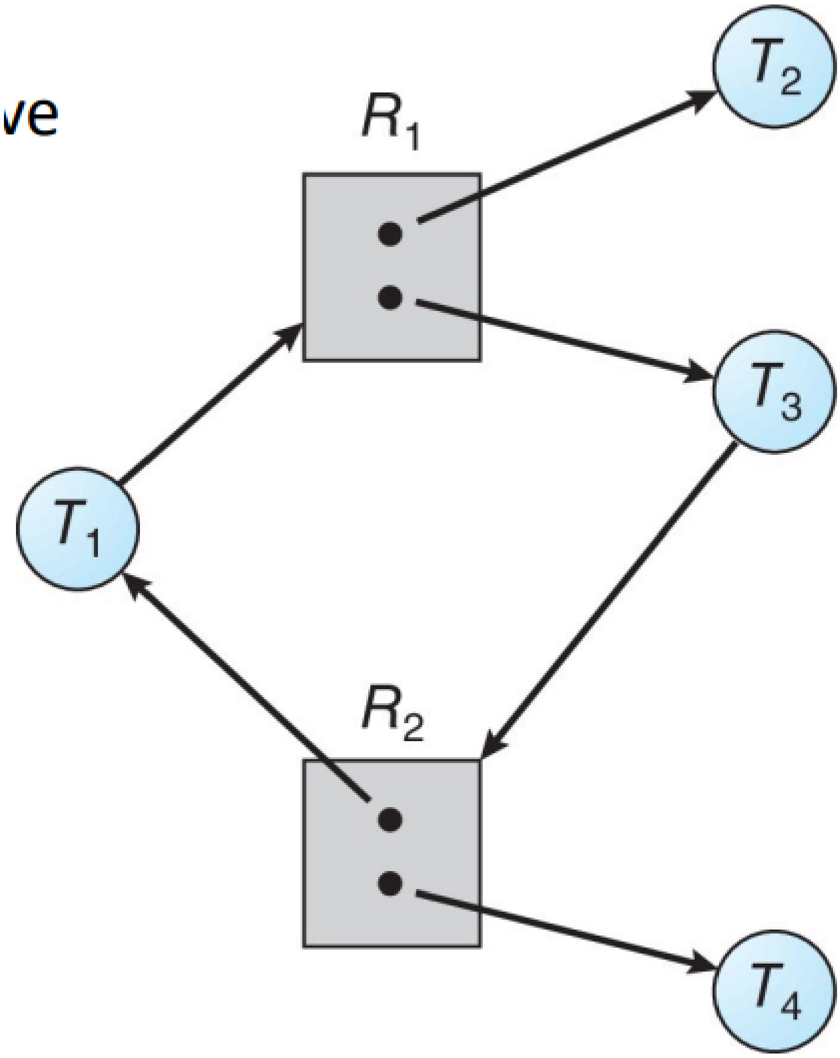
There are 2 cycles.

T1 is assigned R2 and wants R1

T2 is assigned R1+R2 and wants R3

T3 is assigned R3 and wants R2 which it can't get since there's no more instances of R2

```
It might seem like there's no deadlock if we look at the smaller cycle.

T2 is assigned R2 and wants R3
T3 is assigned R3 and wants R2 which it can get since there's another instance of R2

T3 finishes with R3 and gives it up for T2.

This is a static situation so we have a deadlock
```



There is a cycle with the middle 4 nodes.

There is no deadlock.

The outter threads of T_2 and T_4 can finish and release their instances of R_1 and R_2 respectively.

Then T_1 and T_3 can get hold of the resources and work on them.

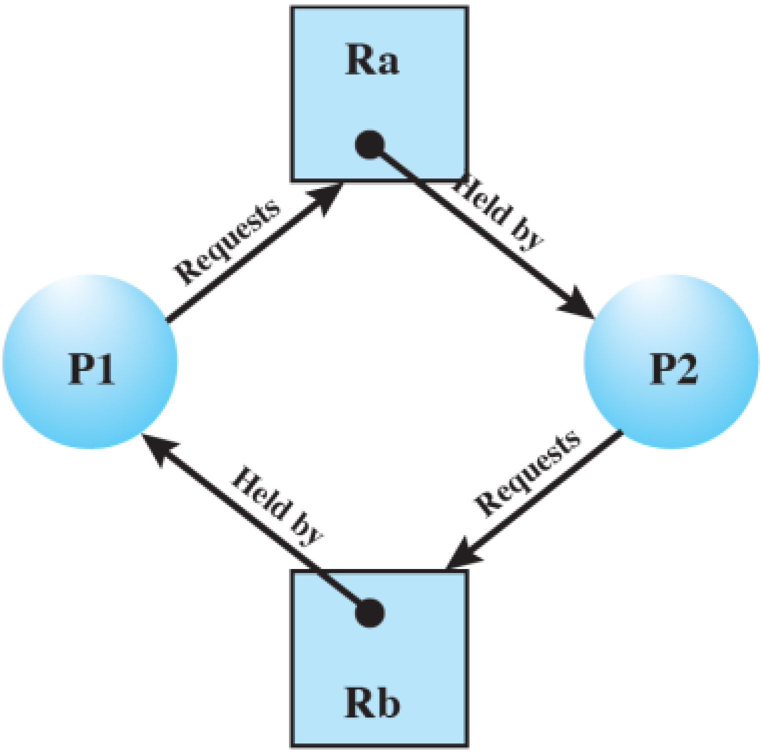
If there is no cycle, then there is no deadlock

If there is a cycle then there may or may not be a deadlock

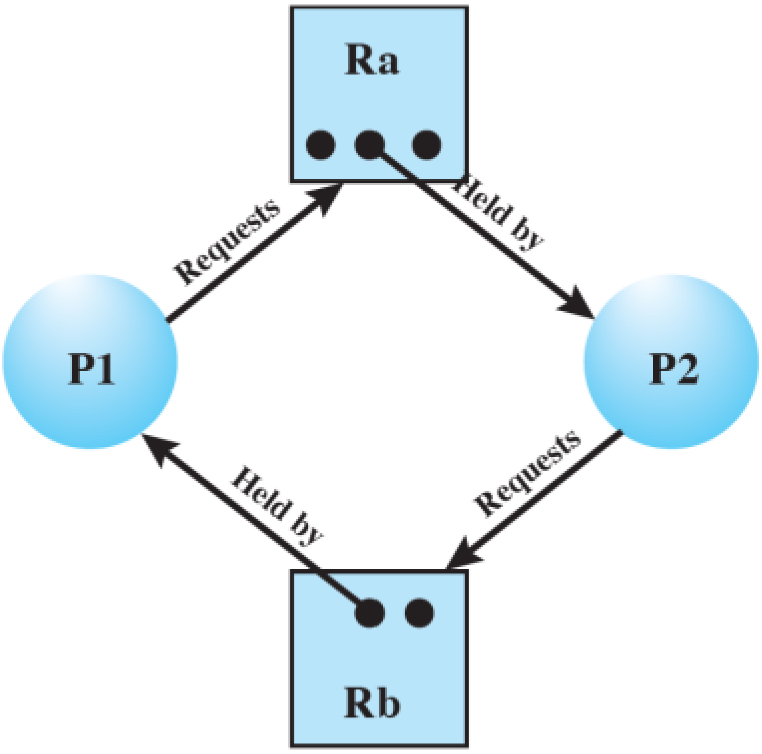
- if only one instance per resource type, then deadlock
- if there are several instances per resource type, possibility of deadlock

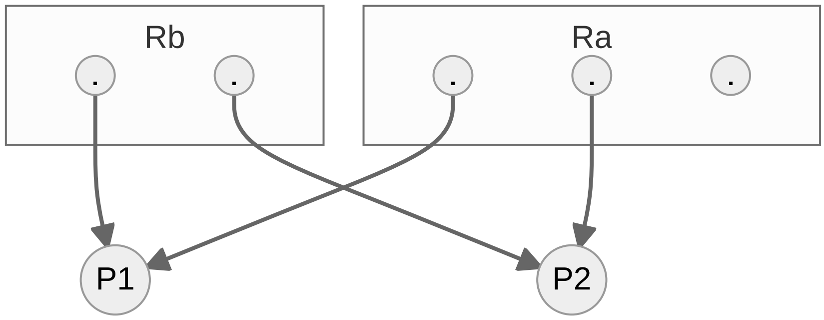
---

deadlock:

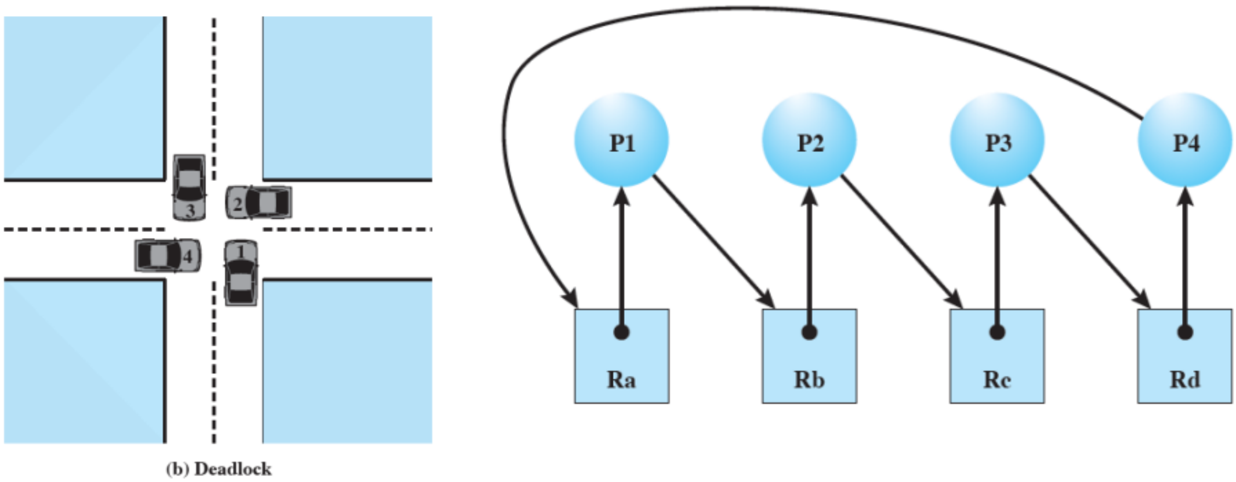

Ra
Requests
Held by
P1
P2
Held by
Requests
Rb

no deadlock:


Ra
Requests
Held by
P1
P2
Held by
Requests
Rb



Resource Allocation graph for the car deadlock example showing circular wait



We can take the numbers on the cars to be the numbers of the processes.

Car 1 is sitting lane **Ra** and wants to cross lane Rb

Car 2 is sitting lane Rb and wants to cross lane Rc

Car 3 is sitting lane Rc and wants to cross lane Rd

Car 4 is sitting lane Rd and wants to cross lane **Ra**

Knowledge Check

- one necessary condition for deadlock is ___, which states taht a process must be holding one resource and waiting to acquire additional resources
  - a: hold and wait
- a cycle in a resource-allocation graph is ___
  - d. a necessary and sufficient condition for a deadlock in the case that each resource has exactly one instance
- if a resource-allocation graph has a cycle, the system must be in a deadlocked state.
  - false

Methods for Handling Deadlocks

In general, there are 3 ways to deal with deadlocks:

1. prevention and avoidance
2. detection and recovery
3. ignoring it

most OSes including windows and linux just pretend that deadlocks don’t happen `???`

Prevention

In order to prevent deadlocks we have to invalidate one of the 4 necessary conditions for deadlock

```
as a refresher

deadlock can arise if 4 conditions hold at the same time:
- mutual exclusion
  - only one process at a time can use a resource
- hold and wait
  - a process holding at least one resource is waiting to acquire additional
resources held by other processes
- no preemption (for resources no the process)
  - resource only released by the process holding it after it's done its task
  - resource cannot be preempted
    - processor can't take resources from process and give it to other processes
arbitrarily
- circular wait
  - process 1 is waiting on process 2
  - process 2 is waiting on process 3
  - ...
  - process n-1 is waiting on process n
  - process n is waiting on process 1
```

Though there are shareable resources, we cannot invalidate mutual exclusion as a whole since some resources just cannot be shared and used at the same time.

Deny Hold and Wait

We can invalidate hold and wait by ensuring that whenever a requests a resource it won’t be holding any other resources. Put another way, If the process is holding a resource then they won’t be making any requests.

how:

- require process to request and allocate all resources before execution
  - **dynamic nature of resource requesting makes this impractical**

- only allow resource requests when process has no resource allocated
    - must release resources before requesting for more

cons:

- low resource utilization
    - resources allocated but not used
- starvation possible
    - a thread may have to wait indefinitely if it needs several popular resources
    - those resources will always end up allocated to processes that will only need those resources
    - `I imagine that even with bounded wait time considerations, priority will cause this to happen as well`

Deny No Preemption

`reminder: non-preemptive is when a process enters into the processor and only leaves when it's finished or voluntarily leaves when waiting on something`

if

- a process is holding some resources
- the process requests another resources
- that resource cannot be immediately allocated to it

then **all resources currently being held are preempted**, released by the processor.

Preempted resources are added to the list of resources that the process is waiting for.

The process starts again once it regains all of the resources it's requesting (the old preempted ones and the new ones).

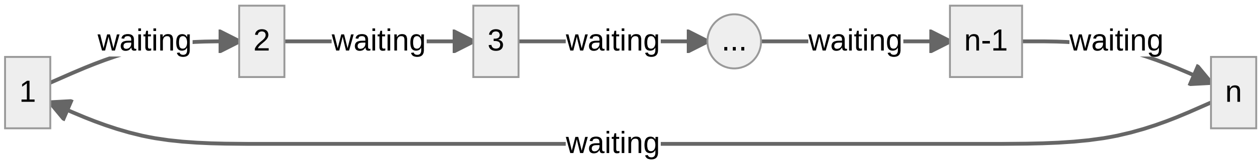
This is often `selectively` applied to resources whose state can be easily saved and restored later, such as cpu registers and database transactions.

cons:

- cannot be applied to resources like mutex locks and semaphores which are the type of resources where deadlock occurs most commonly
    - it's hard to save the condition of a mutex when a process wants to lock for a critical section
    - not easy to store and retrieve $\rightarrow$ not easy to preempt
    - `we can't do this for the shit we actually care about bruhhhhhhh`

Deny Circular Wait

This is the most common approach



how:

- assign each resource a unique number
- resources must be acquired in the order of their unique number

---

Knowledge check:

- to handle deadlocks, operating systems most often ___
  - a. pretend that deadlocks never occur
- both deadlock prevention and deadlock avoidance techniques ensure that the system will never enter a deadlocked state
  - true
- most operating systems choose to ignore deadlocks, because
  - d. all of the above
  - handling is expensive, they occur infrequently, livelock recovery methods can be use on deadlocks

Deadlock Avoidance

Dynamically made decision to see if an allocation request will potentially lead to a deadlock if granted.

System considers

- currently available resources
- currently allocated resources
- and the future requests and releases of each thread

requires knowledge of future process requests.

basics:

- state
  - reflects current allocation of resources to processes
- safe state
  - state where there is at least one execution path where all the processes will finish
  - there will be no deadlock
- unsafe state
  - state resulting in possibility of deadlock
- avoidance

- ensuring system will never enter an unsafe state

Safe State

When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

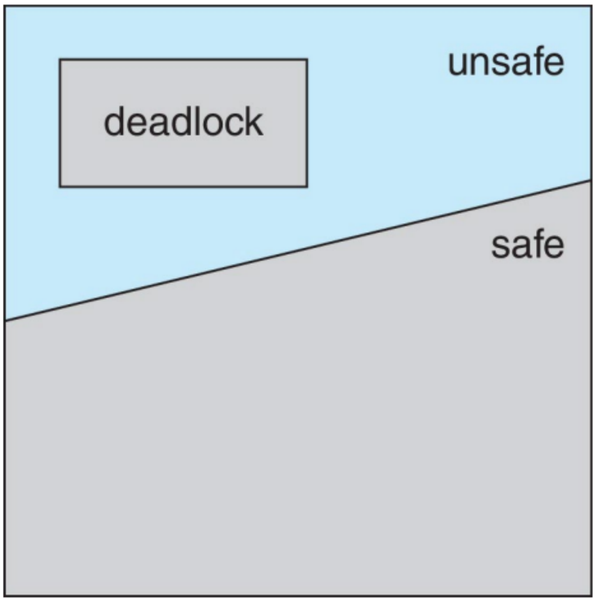
System is in **safe state** if:

- There exists a sequence $<P_1, P_2, ..., P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

That is:

- If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on..

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.



- A system has 12 resources and 3 threads T0, T1, and T2 shown below.
- At t_0, T0 is holding 5 resources, T1 holding two, and T2 holding two.
- Question: Does the sequence <T1, T0, T2> satisfies the safety condition?
  - T1: 2 + 2 –> done, release back 4 –> 5 available.
  - T0: 5 + 5 –> done, release back 10 –> 10 available.
  - T2: 2 + 9 –> done, release back 9 –> 12 available.

	Maximum Needs	Current Needs
T_0	10	5
T_1	4	2
T_2	9	2

In the above example we only have 3 resources available.

For T1 we give it 2 then release back 4 once it's done.

Then the above steps just go as shown.

So we are in a safe state

- A system has 12 resources and 3 threads T0, T1, and T2 shown below.
- At t_1, thread T2 requests and is allocated one more resource.
- Question: Is the system now in a safe state?
  - T2: 3, may request 6 –> at t_1 : 2 available.
  - T0: 5, may request 5 –> at t_1 : 2 available.
  - T1: 2 + 2 –> done, release back 4 –> at t_2 : 4 available.

	Maximum Needs	Current Needs
T_0	10	5
T_1	4	2
T_2	9	**3**

This is the same example as seen above but now T_2 needs 1 more resource, requests the resource and receives it, resulting in only 2 resources available at the beginning.

We can start the sequence with T_1 and allocate 2 then get back 2 once done.

We end up with only 4 available resources which isn't enough to satisfy the maximum needs of T_0 and T_2

Avoidance Algos

There are 2 algos:

1. using a resource-allocation when there's a single instance of a resource type
2. use the Banker's Algo when there's multiple instances of a resource type

Resource-Allocation Graph Scheme

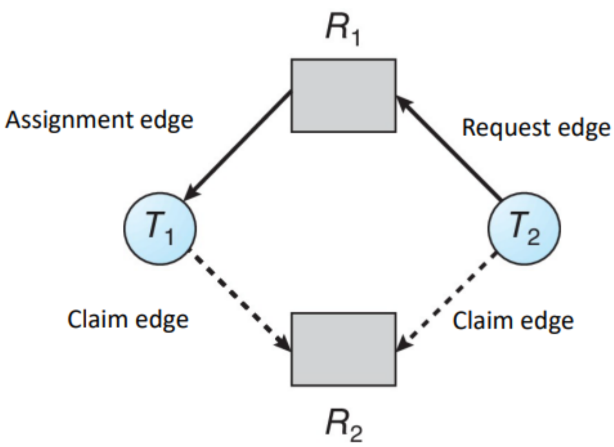
Claim edge $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j represented by a dashed line

- Claim edge converts to **request edge** when a process requests a resource
- Request edge converted to an **assignment edge** when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge

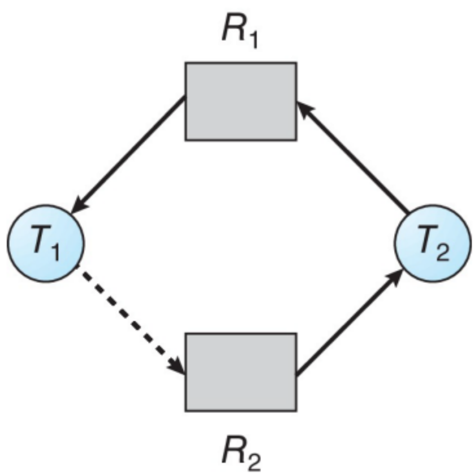
Resources must be claimed *a priori* in the system.

`a priori` = `ahead of time`

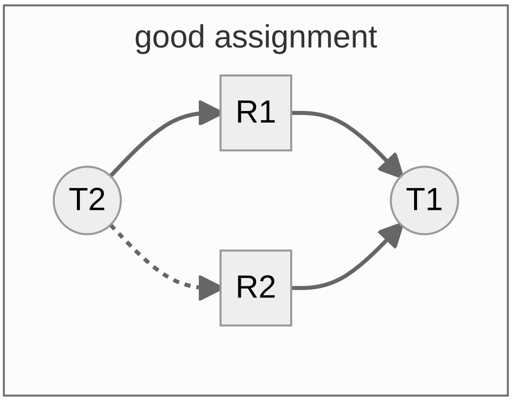
claims turn into requests which turn into assignments before turning back into claim edges



Although R2 is currently free,
we cannot allocate it to T2.



An unsafe state in a resource-allocation graph.



Suppose that process P_i requests a resource R_j

The request can be granted **only** if converting the request edge to an assignment edge **does not result in the formation of a cycle** in the resource allocation graph.

make sure there is no cycles forming if you convert a request edge to an assignment edge

Banker's Algorithm

`prof misspells this is as Baker at some point. It's Banker.`

conditions:

- multiple instances of resources
- each process must have a prior claim maximum use
- when a process requests a resource, it may have to wait
- when aprocess gets all its resources it must return them in a finite amount of time

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available.
- **Max**: $n \times m$ matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j.
- **Allocation**: $n \times m$ matrix. If $Allocation[i,j] = k$ then P_i is currently allocated k instances of R_j.
- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task.

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j]$$

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:
   Work = Available
   Finish **[*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an index ***i*** such that both:
   (a) ***Finish*** **[*i*] = *false***
   (b) $\boldsymbol{Need_i \leq Work}$
   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish**[*i*] = *true***
   go to step 2

4. If ***Finish*** **[*i*] == *true*** for all ***i***, then the system is in a **safe** state otherwise we are in **unsafe** state.

$\boldsymbol{Request_i}$ = request vector for process $\boldsymbol{P_i}$ If $\boldsymbol{Request_i[j] = k}$ then process $\boldsymbol{P_i}$ wants ***k*** instances of resource type $\boldsymbol{R_j}$

1. If $\boldsymbol{Request_i \leq Need_i}$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If $\boldsymbol{Request_i \leq Available}$, go to step 3. Otherwise, $\boldsymbol{P_i}$ must wait, since resources are not available.
3. Pretend to allocate requested resources to $\boldsymbol{P_i}$ by modifying the state as follows:

   $\boldsymbol{Available = Available - Request_i;}$

   $\boldsymbol{Allocation_i = Allocation_i + Request_i;}$

   $\boldsymbol{Need_i = Need_i - Request_i;}$

- If **safe** $\Rightarrow$ the resources are allocated to $\boldsymbol{P_i}$ *(use Safety alg. in the prev slide).*
- If **unsafe** $\Rightarrow$ $\boldsymbol{P_i}$ must wait, and the old resource-allocation state is restored.

Example of Banker's Algorithm /1

- 5 processes P_0 through P_4;
- 3 resource types:

  A (10 instances), *B* (5 instances), and *C* (7 instances)

- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Q: Does the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfy safety criteria?

A: yes

we start with $< 3, 3, 2 >$

P_1 can be allocated and give back $< 5, 3, 2 >$

P_3 can be allocated and give back $< 7, 4, 3 >$

P_4 can be allocated and give back $< 7, 4, 5 >$

P_2 can be allocated and give back $< 10, 4, 7 >$, making all instances of A and C available

P_0 can be allocated and give back $< 10, 5, 7 >$, making all instances of all resources available.

Example of Banker's Algorithm /3

- Does the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfy safety criteria?

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	3 3 2
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $< \boldsymbol{P_1, P_3, P_4, P_2, P_0} >$ **satisfies** safety requirement.

you do the same process with this.

A: the sequence satisfies safety requirement

Example: P1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Is the new state safe?
- Executing safety algorithm shows that sequence $< \boldsymbol{P_1, P_3, P_4, P_0, P_2} >$ **satisfies** safety requirement.
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

prof will pick this up next time

Recovery from Deadlock