Deadlock

沈司晨，吴卓成，赵英迪

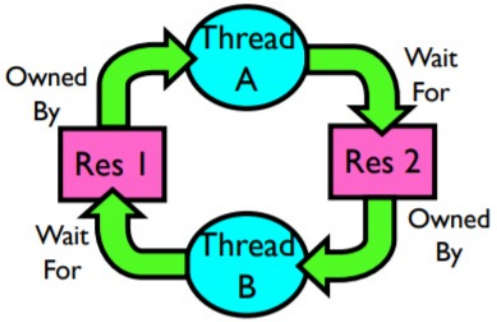
2025/11/02

Deadlock: Characteristics and Handling

Deadlocks

Starvation vs. Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - Low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1
  - Deadlock ⇒ Starvation but not vice versa
    - Starvation can end (but does not have to)
    - Deadlock cannot end without external intervention



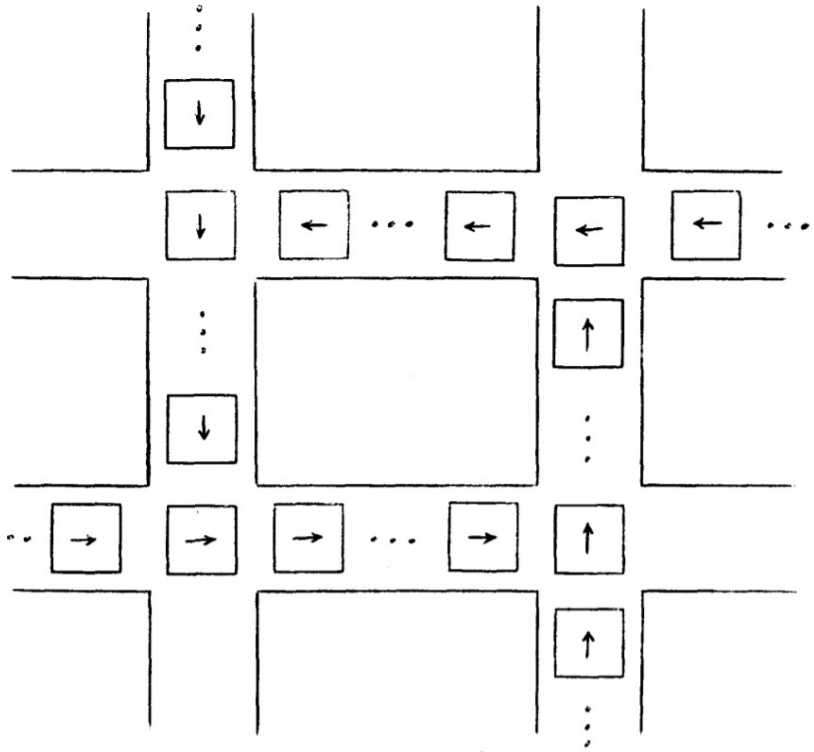
Deadlocks

```
public class DeadlockExample {
    private static final Object resource1 = new Object();
    private static final Object resource2 = new Object();

    public static void main(String[] args) {

        Thread thread1 = new Thread(() -> {
            synchronized (resource1) {
                System.out.println("线程1已获取资源1，等待资源2...");
                try {
                    // 模拟处理逻辑，增加死锁概率
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 尝试获取resource2，此时线程2已持有resource2
                synchronized (resource2) {
                    System.out.println("线程1已获取资源2，执行完成");
                }
            }
        });

        Thread thread2 = new Thread(() -> {
            synchronized (resource2) {
                System.out.println("线程2已获取资源2，等待资源1...");
                try {
                    // 模拟处理逻辑，增加死锁概率
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                // 尝试获取resource1，此时线程1已持有resource1
                synchronized (resource1) {
                    System.out.println("线程2已获取资源1，执行完成");
                }
            }
        });

        thread1.start();
        thread2.start();
    }
}
```

Deadlocks

This deadlock situation has arisen only because all of the following general conditions were operative:

1) Tasks claim exclusive control of the resources they require ("mutual exclusion" condition).
2) Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition).
3) Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion ("no preemption" condition).
4) A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain ("circular wait" condition).



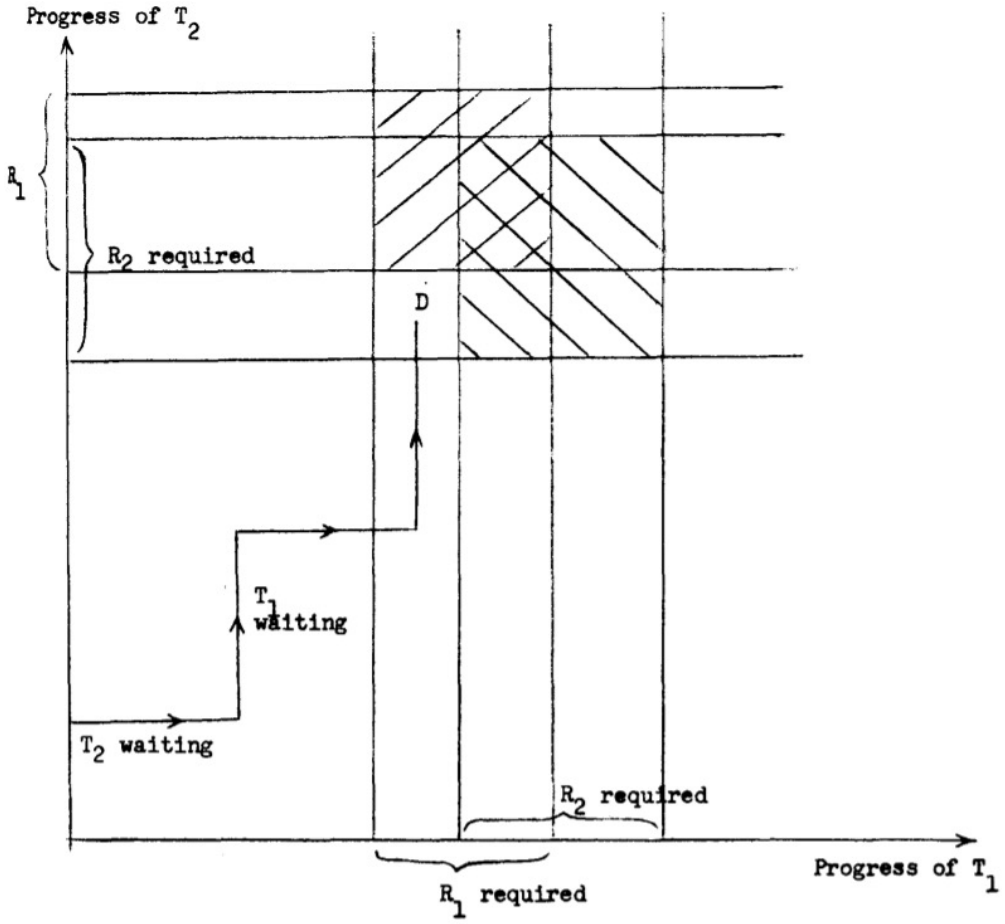
Deadlocks

T1:

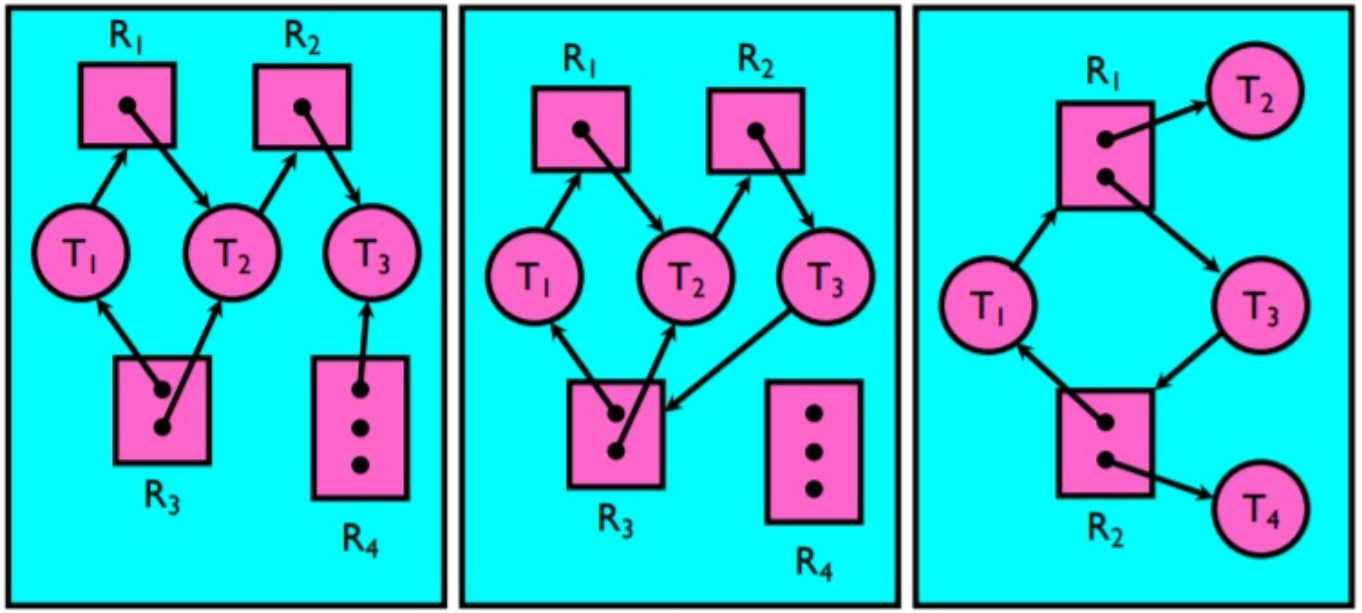
获取R1
耗时……
获取R2
耗时……
释放R1
耗时……
释放R2
耗时……

T2:

获取R2
耗时……
获取R1
耗时……
释放R2
耗时……
释放R1
耗时……



Deadlocks



Simple Resource Allocation Graph

Allocation Graph With Deadlock

Allocation Graph with Cycle, but No Deadlock

Deadlock Immunity

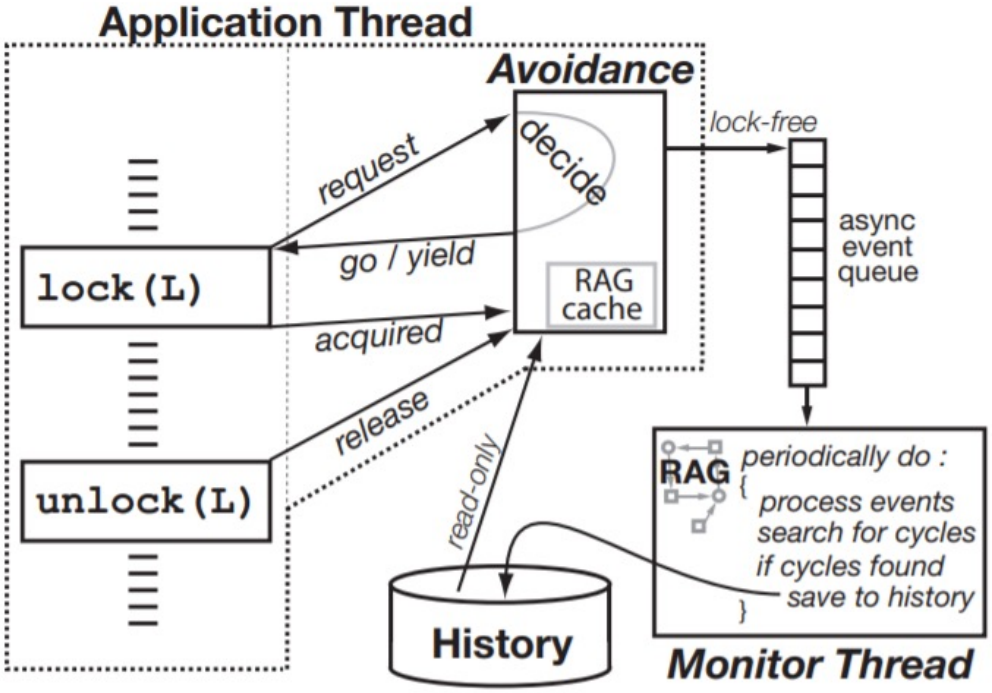
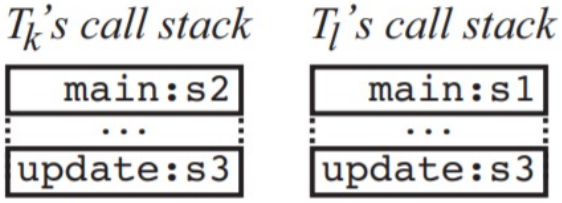


Figure 1: Dimmunix architecture.

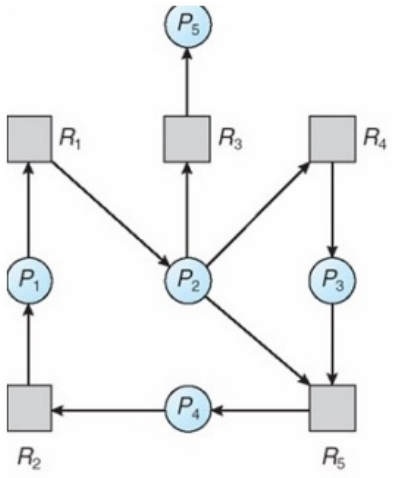
```
    main {
s1:   update(A,B)
      ...
s2:   update(B,A)
    }

    update(x,y) {
s3:   lock(x)
s4:   lock(y)
      ...
      unlock(y)
      unlock(x)
    }
```



Deadlock: Detection & Prevention

Deadlock Detection with Resource Allocation Graphs

- Only one of each type of resource ⇒ look for cycles
- More than one resource of each type
  - More complex deadlock detection algorithm
  - Next page



Several Instances per Resource Type

- **Available**: A vector of length *m* indicates the number of available resources of each type.

- **Allocation**: An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.

- **Request**: An *n* x *m* matrix indicates the current request of each process. If Request $[i_j]$ = k, then process P_i is requesting k more instances of resource type R_j.

Detection Algorithm

Core: Imitate the Operating System

1. Let **Work** and **Finish** be vectors of length m and n, respectively Initialize:
   (a) **Work** = **Available**
   (b) For i = 1,2, …, n, if $\textbf{Allocation}_i \neq 0$, then **Finish**[i] = false; otherwise, **Finish**[i] = true
2. Find an index i such that both:
   (a) **Finish**[i] == false
   (b) $\textbf{Request}_i \leq$ Work

   If no such i exists, go to step 4

3. **Work** = **Work** + $\textbf{Allocation}_i$
   Finish[i] = true
   go to step 2

4. If **Finish**[i] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if **Finish**[i] == false, then P_i is deadlocked

$O(n^2)$

Detection Algorithm

- By ordering the resource requests by size and associating with each task a count of the number of types of resources being requested, a detection algorithm can be devised which has a running time that varies linearly with the number of tasks.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0:

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in Finish[i] = true for all i

Example (Cont.)

- P_2 requests an additional instance of type C

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 1	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- State of system?
  - Can reclaim resources held by process P_0 (not deadlocked), but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

What if Deadlock Detected?

- Terminate process, force it to give up resources
  - Shoot a dining philosopher !?
  - But, not always possible

Terminate by Order—
Remaining Processes can run

- Preempt resources without killing off process
  - Take away resources from process temporarily
  - Does not always fit with semantics of computation

Only Possible when the resource can be preempted
e.g. Toilet System Deadlocks

- Roll back actions of deadlocked process
  - Common technique in databases (transactions)
  - Of course, deadlock may happen once again

A more general technique [14] has been devised that assigns a fixed cost c_i to the removal (forced preemption) of a resource of type r_i from a deadlocked task that is being aborted. Thus, the cost of removing resources from a deadlocked task T_i is:

$$\sum_{j=1}^{s} c_i \cdot g(q_{ij} - v_j) \quad \text{where } g(x) = \begin{matrix} x; x > 0 \\ 0; x \leqslant 0 \end{matrix}$$

An algorithm has been designed that finds a subset of resources that would incur the minimum cost if preempted. The algorithm finds a minimum cost solution by an efficient tree-search procedure that can be characte ized as a branch-and-bound technique. (A gorithm A is used to isolate new sets deadlocked tasks in the sequence of tria removals made by the algorithm.)

Deadlock Prevention: Recall

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - T_1 is waiting for a resource that is held by T_2
    - T_2 is waiting for a resource that is held by T_3
    - ...
    - T_n is waiting for a resource that is held by T_1

- Remove "Mutual Exclusion": not possible for non-sharable resources
- Remove "Hold and Wait" – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
  - Low resource utilization; starvation possible

Deadlock Prevention: Recall

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additi resources held by other threads
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it
- Circular wait
  - There exists a set $\{T_1, \ldots, T_n\}$ of waiting threads
    - T_1 is waiting for a resource that is held by T_2
    - T_2 is waiting for a resource that is held by T_3
    - ...
    - T_n is waiting for a resource that is held by T_1

- Remove "Preemption"
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting

Easy to save and recover: CPU

- Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
- $R=\{R_1,R_2,\ldots,R_m\}$
- One to one function F:R->N
- If a process request a resource R_i, it can request another resource R_j if and only if $F(R_i) < F(R_j)$
- Or, it must first release all resource R_i such that $F(R_i) \geq F(R_j)$

Deadlock: Avoidance

Deadlock Avoidance (via Scheduling)

- Avoidance requires that the system has some additional a priori information available.
  - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.

	T1	**T2**	**T3**	**T4**
R1	yes	yes	no	no
R2	yes	yes	yes	no

Note that T3 can never cause a deadlock by running concurrently with other threads because it only grabs one lock.

Preliminary

- The set P of processes $\{p_1, p_2, p_3, \ldots, p_n\}$.
- Assume only one resource type (can be easily extended to multiple types)
- ***Total***: the total number of instances of resource.
- ***Max***: Vector of length n. If $Max[p_i] = k$, then process p_i may request at most k instances of resource.
- ***Allocation***: Vector of length n. If $Allocation[p_i] = k$, then process p_i currently has k instances of resource.
- ***Need***: Vector of length n. $Need[p_i] = Max[p_i] - Allocation[p_i]$.
- ***Available***: the number of currently available resources.

$$Available = Total - \sum_{i=1}^{n} Allocation[p_i]$$

Safe State

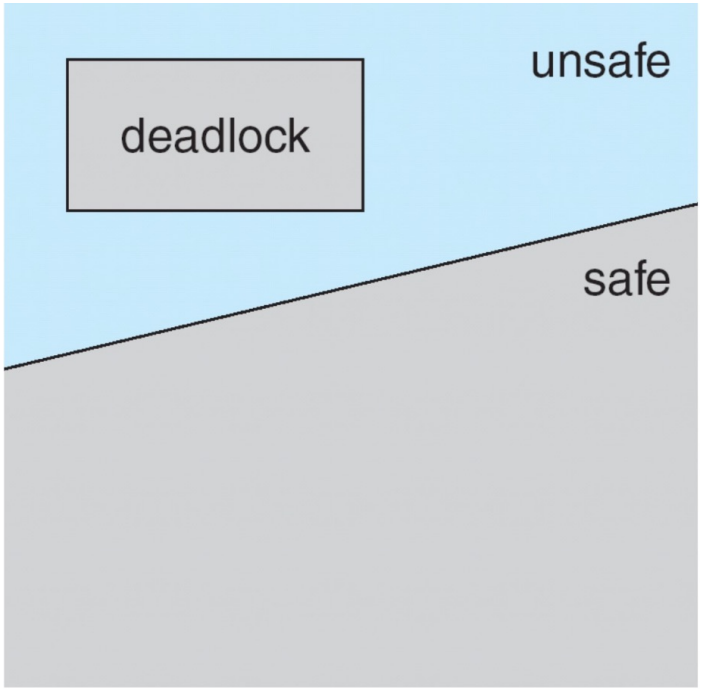
- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.
- The system is in **safe state** if there exists a permutation of all processes, say $\langle p_1, p_2, \ldots, p_n \rangle$ such that for each p_i the resources that p_i can still request can be satisfied by currently available resources + resources held by all p_j, where $j < i$.

$$i.e. Need[p_i] \leq Available + \sum_{j=1}^{i-1} Allocation[p_j], \forall 1 \leq i \leq n$$

  - If what p_i resource needs are not immediately available, then Pi can wait until all p_j have finished.
  - When all 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.

Safe, Unsafe, Deadlock State

- If a system is in safe state → no circular wait → no deadlocks.
- If a system is in unsafe state → **possibility** of deadlock.
- Deadlock avoidance → ensure that a system will never enter an unsafe state.



Banker's Algorithm

- **Idea**: Tries to find such a permutation by keeping

  $Need[p_i] \leq Available + \sum_{j=1}^{i-1} Allocation[p_j], \forall 1 \leq i \leq k$ $(*)$ invariant for all k.
- After having established it (trivially) by means of $k = 1$, it then tries to increase k by 1 under the invariance of $(*)$ until $k = n$.
- We can does so by not changing $\langle p_1, p_2, \ldots, p_{k-1} \rangle$ and search for an h such that

$$k \leq h \leq n \text{ and } Need[p_h] \leq Available + \sum_{j=1}^{k-1} Allocation[p_j]$$

- Then we can swap p_k and p_h, then increase k by 1.
- If no such h exists at some point, the algorithm fails.

Banker's Algorithm: Proof

- Theorem: If Banker's Algorithm fails, then the system is not safe, $i.e.$ no permutation of the processes can satisfy

$$Need[p_i] \leq Available + \sum_{j=1}^{i-1} Allocation[p_j], \forall 1 \leq i \leq n$$

- Proof:
  - Consider the point when the algorithm fails, all processes can be partitioned into two disjoint sets A and B, where $\forall\ b \in B, Need[b] > Available + \sum_{a \in A} Allocation[a]$.
  - Assume there is such a permutation $\langle p_1, p_2, \ldots, p_n \rangle$, let p_i be first process in the permution that belongs to B. Then $p_1, \ldots, p_{i-1} \in A$ and

$$Need[p_i] \leq Available + \sum_{j=1}^{i-1} Allocation[p_j]$$

  - However, this contradicts to $p_i \in B$!

Banker's Algorithm: Example

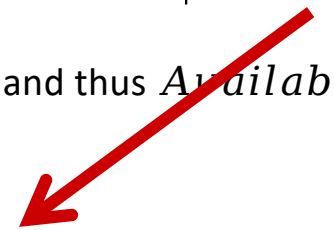
	P1	P2	P3	P4	P5
Max	7	3	9	2	4
Allocation	0	2	3	2	0
Need	7	1	6	0	4

Assume $Total = 10$, and thus $Available = Total - (2 + 3 + 2) = 3$.

Banker's Algorithm: Example

	P1	P2	P3	P4	P5
Max	7	3	9	2	4
Allocation	0	2	3	2	0
Need	7	1	6	0	4

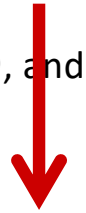
Assume $Total = 10$, and thus $Available = Total - (2 + 3 + 2) = 3$.



$k = 1$:
$Available = 3$

Banker's Algorithm: Example

	P1	P2	P3	P4	P5
Max	3	7	9	2	4
Allocation	2	0	3	2	0
Need	1	7	6	0	4



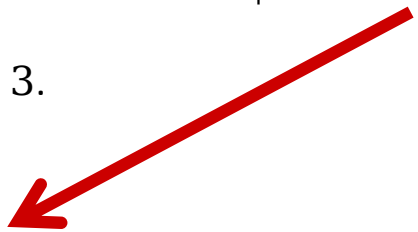
Assume $Total = 10$, and thus $Available = Total - (2 + 3 + 2) = 3$.

$k = 1$:
$Available = 3$

Banker's Algorithm: Example

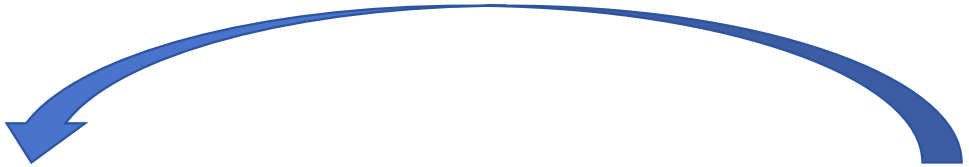
	P1	P2	P3	P4	P5
Max	3	7	9	2	4
Allocation	2	0	3	2	0
Need	1	7	6	0	4

Assume $Total = 10$, and thus $Available = Total - (2 + 3 + 2) = 3$.



$k = 2$:
$Available + Allocation[P1] = 5$

Banker's Algorithm: Example



	P1	P2	P3	P4	P5
Max	3	4	9	2	7
Allocation	2	0	3	2	0
Need	1	4	6	0	7

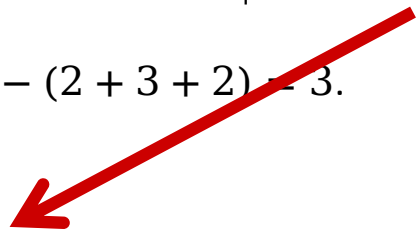
Assume $Total = 10$, and thus $Available = Total - (2 + 3 + 2) = 3$.

$k = 2$:
$Available + Allocation[P1] = 5$

Banker's Algorithm: Example

	P1	P2	P3	P4	P5
Max	3	4	9	2	7
Allocation	2	0	3	2	0
Need	1	4	6	0	7

Assume $Total = 10$, and thus $Available = Total - (2 + 3 + 2) = 3$.



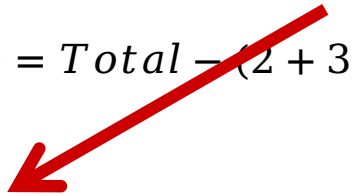
$k = 3$:

$$Available + \sum_{i=1}^{2} Allocation[Pi] = 5$$

Banker's Algorithm: Example

	P1	P2	P3	P4	P5
Max	3	4	2	9	7
Allocation	2	0	2	3	0
Need	1	4	0	6	7

Assume $Total = 10$, and thus $Available = Total - (2 + 3 + 2) = 3$.



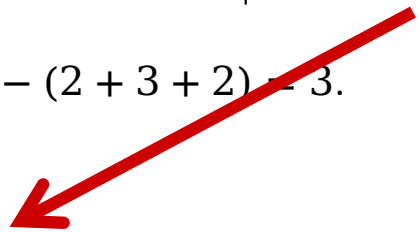
$k = 3$:

$$Available + \sum_{i=1}^{2} Allocation[Pi] = 5$$

Banker's Algorithm: Example

	P1	P2	P3	P4	P5
Max	3	4	2	9	7
Allocation	2	0	2	3	0
Need	1	4	0	6	7

Assume $Total = 10$, and thus $Available = Total - (2 + 3 + 2) = 3$.



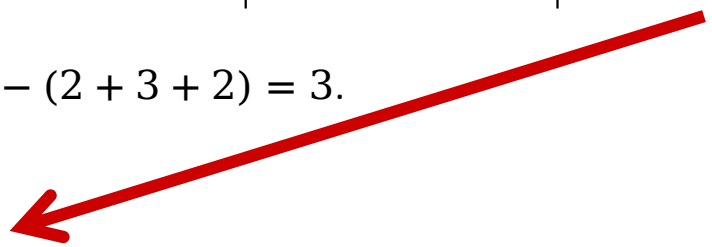
$k = 4$:

$$Available + \sum_{i=1}^{3} Allocation[Pi] = 7$$

Banker's Algorithm: Example

	P1	P2	P3	P4	P5
Max	3	4	2	9	7
Allocation	2	0	2	3	0
Need	1	4	0	6	7

Assume $Total = 10$, and thus $Available = Total - (2 + 3 + 2) = 3$.



$k = 5$:

$$Available + \sum_{i=1}^{4} Allocation[Pi] = 10$$

Defect

	T1	T2	T3	T4
R1	yes	yes	yes	no
R2	yes	yes	yes	no

- Static scheduling leads to a conservative approach where T1, T2, and T3 are all run on the same processor, and thus the total time to complete the jobs is lengthened considerably.

PERFORMANCE↓

Question

What to do if Banker's Algorithm fails?

- Release some processes' resources and let them retry later?
- Can we decrease *Max* value for some processes?
- ...

Thank You!