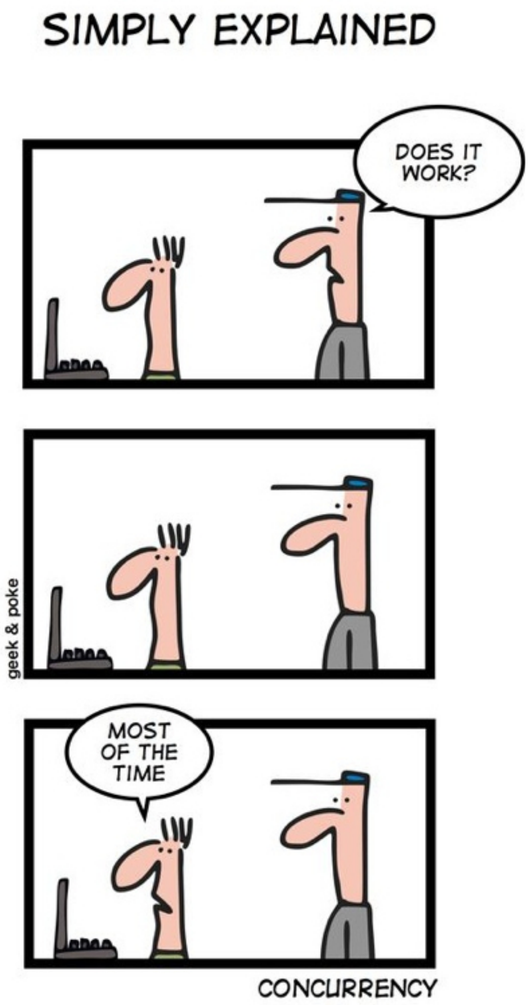
Concurrency (Part V):
Mutual Exclusion, Synchronization, **Deadlock, and Starvation**

Professor Travis Peters
CSCI 460 Operating Systems
Fall 2019

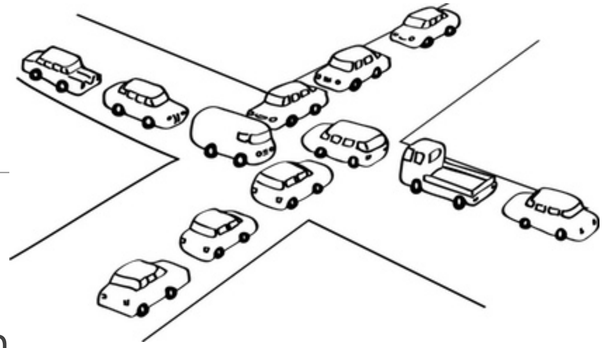
Some slides & figures adapted from Stallings instructor resources.

Some slides adapted from Adam Bates's F'18 CS423 course @ UIUC
https://courses.engr.illinois.edu/cs423/sp2018/schedule.html



–http://www.datamation.com/news/tech-comics-quantum-physics-2.html

Goals for Today



Learning Objectives

- Understand deadlock and starvation, as well as strategies to address them.

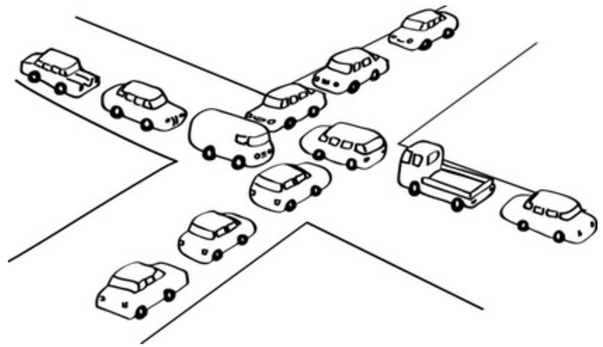
Announcements

- YOUR & NCUR 2020
  - http://www.montana.edu/your/
    Do cool research @ MSU as an undergrad!
  - http://www.montana.edu/ncur2020/
    Submit an abstract to present your research or volunteer @ NCUR 2020!
- *Coming Up…*
  - Homework 3 (Chapters 5-6)
  - Exam will be held ***in-class*** next week on **Monday (10/7)**

Deadlock

- The ***permanent*** blocking of a set of processes that either compete for system resources or communicate with each other.
- A set of processes is deadlocked when each process is blocked awaiting an event.
- We say *permanent* because none of the events is ever triggered…

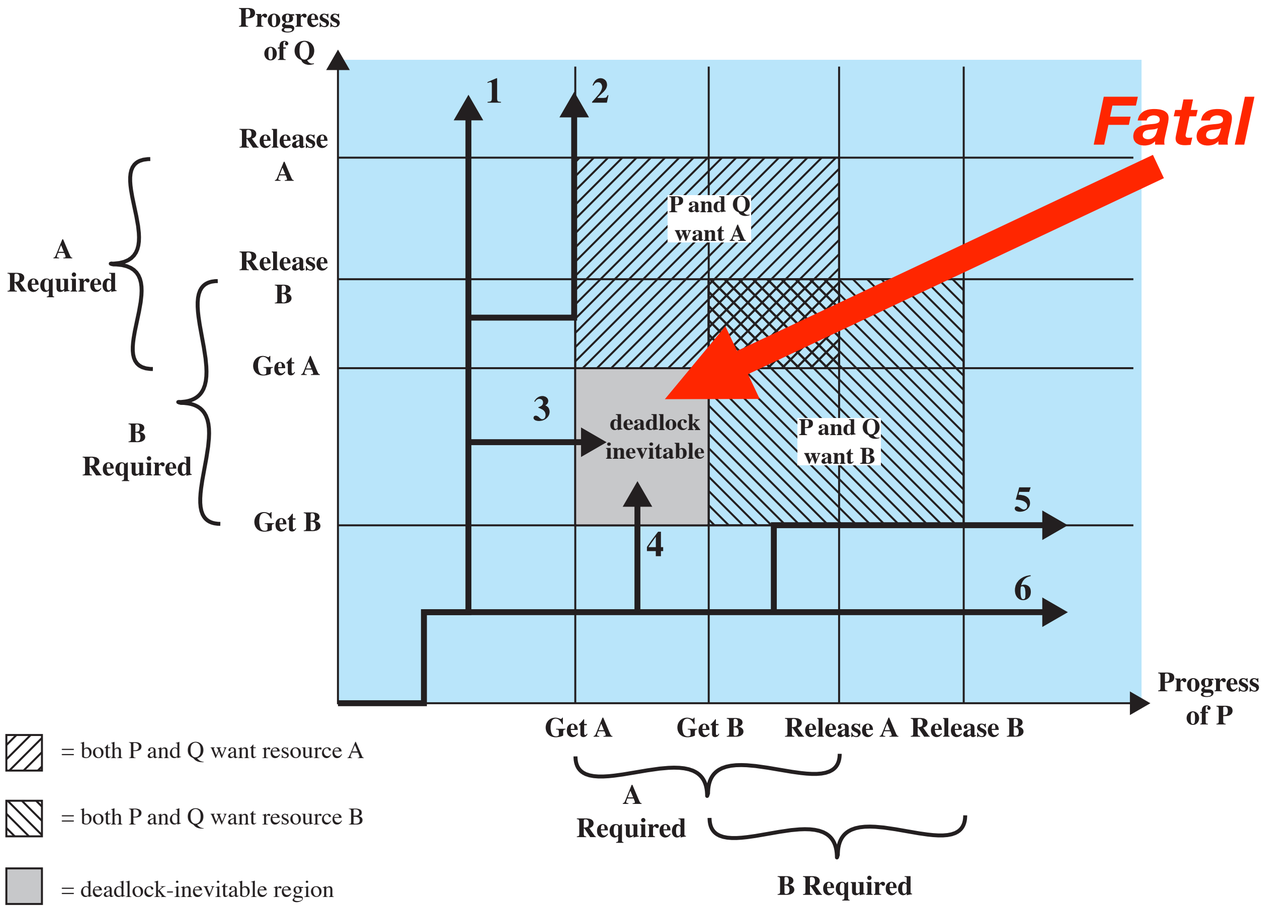
Unfortunately, there is no efficient solution in the general case…



Example (1): 2 Competing Processes, 2 Required Resources

*Visualized w/ a **Joint Progress Diagram***

Not all execution paths lead to deadlock…



P	Q
…	…
Get A	Get B
…	…
Get B	Get A
…	…
Release A	Release B
…	…
Release B	Release A
…	…

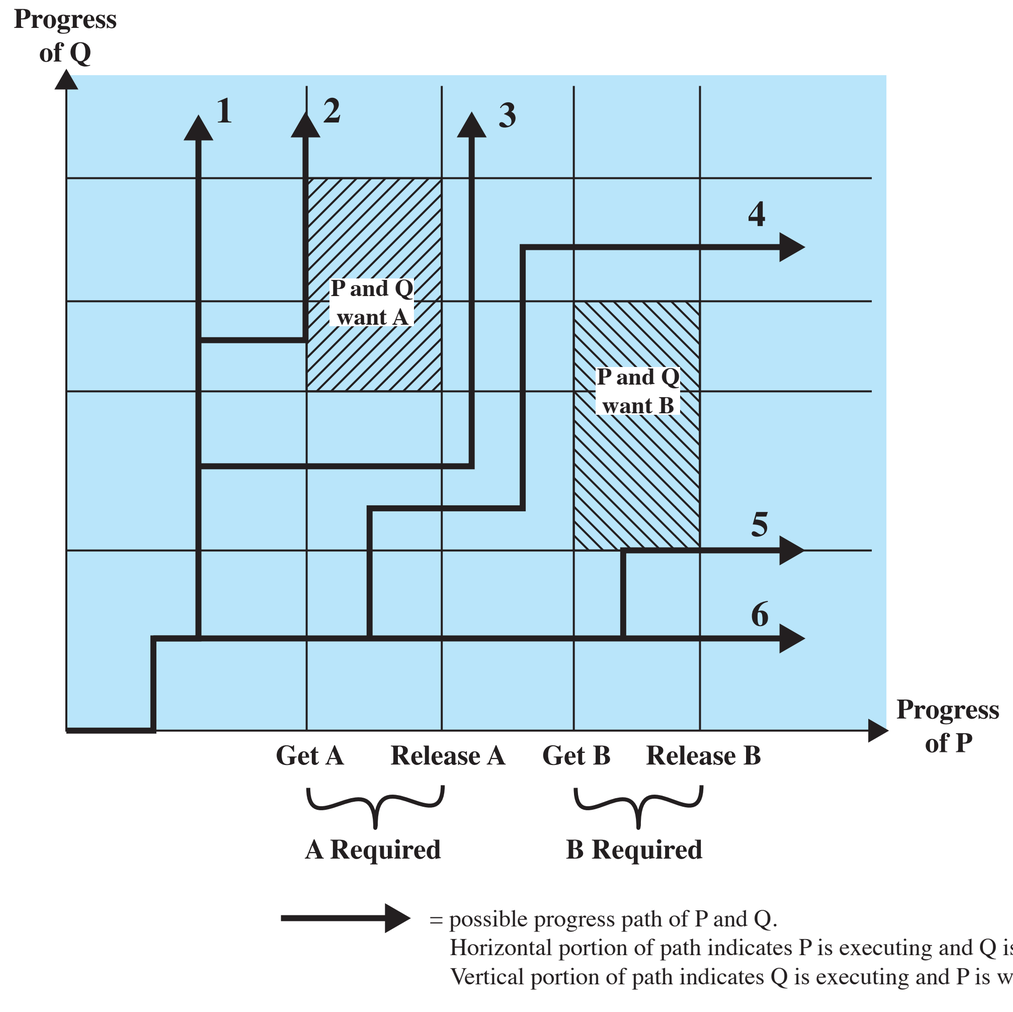
Deadlock is only inevitable if both processes enter the fatal region!

Example (2): 2 Competing Processes, 2 Required Resources

*Visualized w/ a **Joint Progress Diagram***

If P doesn't require both resources at the same time…

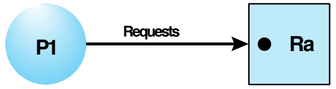
P	Q
…	…
Get A	Get B
…	…
Release A	Get A
…	…
Get B	Release B
…	…
Release B	Release A
…	…



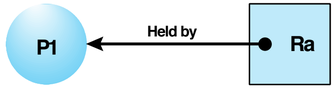
…no deadlock!

No General Solution… But We Do Have Common Strategies!

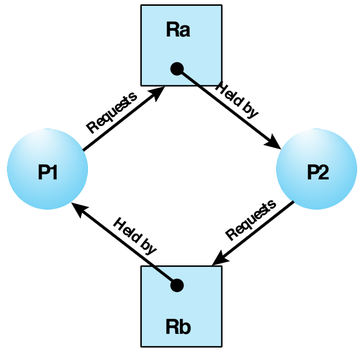
- Strategies for Addressing Deadlock
  - Prevention — *disallow/prevent one of the **deadlock conditions**; i.e., possibility of deadlock is excluded*
  - Avoidance — *do not grant resource request if it might lead to deadlock*
  - Detection — *grant resource requests when possible; periodically check for deadlock + take action to recover*



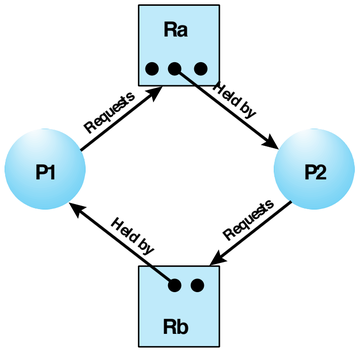
(a) Resouce is requested



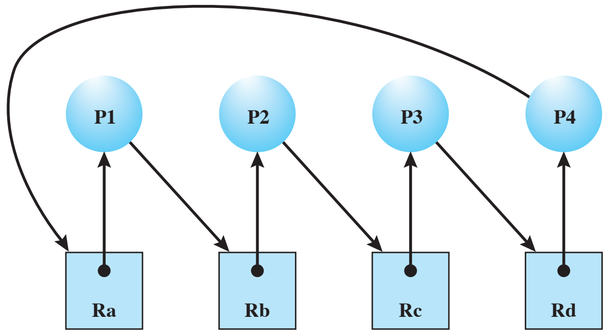
(b) Resource is held



(c) Circular wait



(d) No deadlock



The Conditions for Deadlock

For deadlock to occur, ***all 4 must hold****…*

1. Mutual Exclusion

   Only 1 process can use a resource at a time. No other process may access that resource.

2. Hold and Wait

   A process may hold allocated resource while awaiting assignment of other resources.

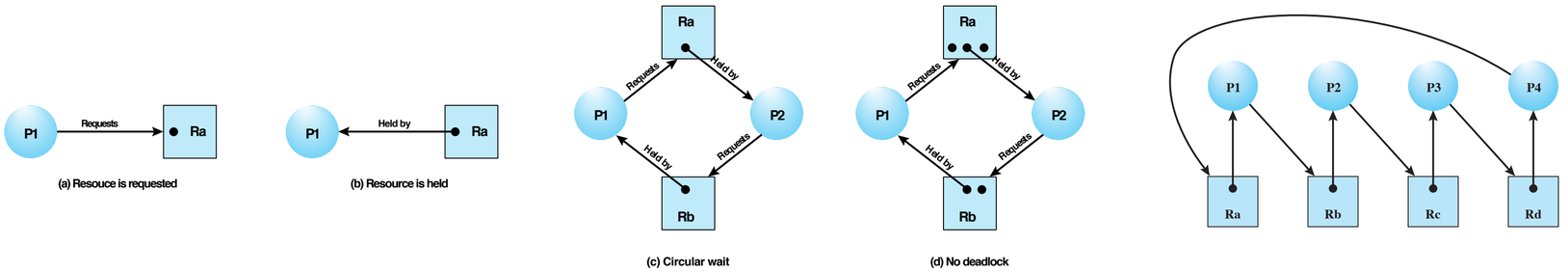
3. No Preemption

   No resource can be forcibly removed from a process holding it.

4. Circular Wait

   A closed chain of processes exists s.t. each process holds at least one resource needed by the next process in the chain.

***Indirect** (1-3) vs. **Direct** (4)*



Dining Philosophers Problem

Recall: Deadlock Conditions

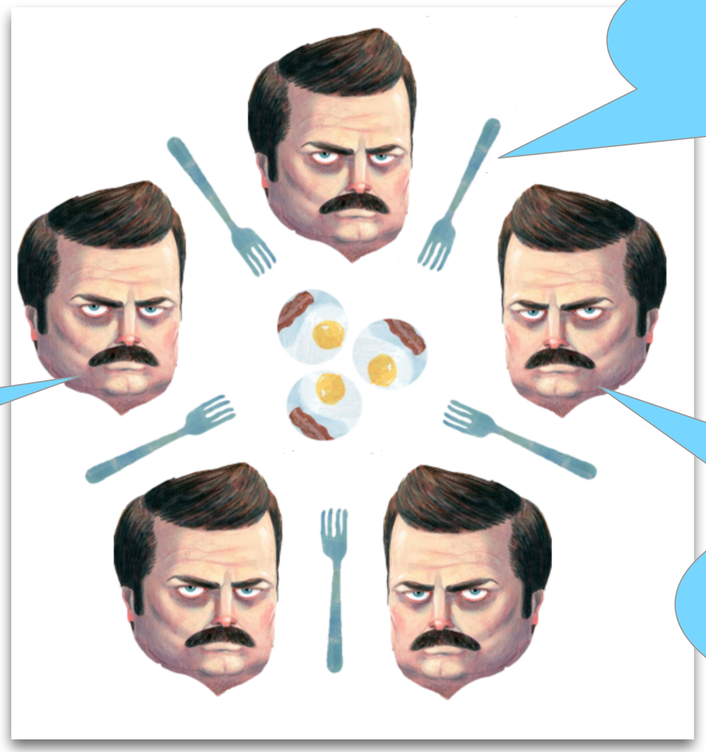
For deadlock to occur, all 4 must hold…

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

Rules

1. No two Philosophers (Rons) can use the same fork at the same time…
2. No Philosopher (Ron) should starve to death…

```
/* program     diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

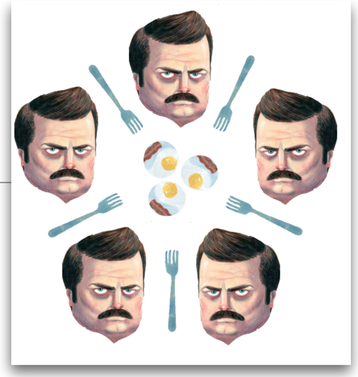


Each Fork:
Who is holding me?

Each Ron:
I'm Hungry…
Give me bacon and eggs…

Each Ron (Philosopher):
Which fork(s) are mine?

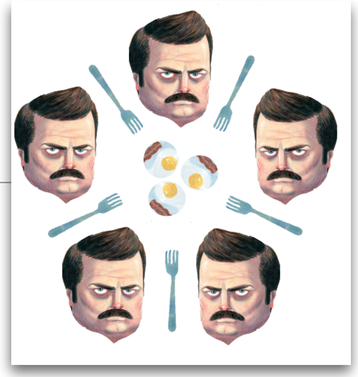
Deadlock **Prevention** — *disallow/prevent one of the deadlock conditions*



- Can we "prevent"…
  1. Mutual Exclusion?
  2. Hold and Wait?
  3. No Preemption?
  4. Circular Wait?

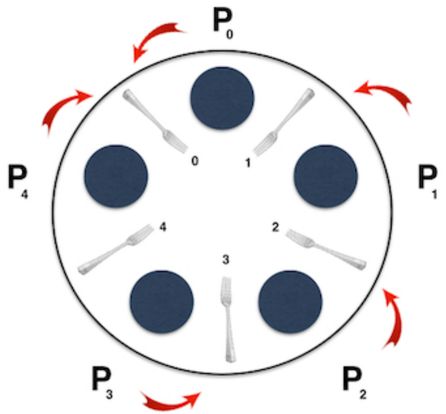
  Why or Why Not? Any problems? Advantages? Disadvantages?
- i.e., How to "break" one of these conditions?

Deadlock **Prevention** — *disallow/prevent one of the deadlock conditions*



- Can we "prevent"…
  1. Mutual Exclusion?
  2. Hold and Wait?
  3. No Preemption?
  4. Circular Wait?

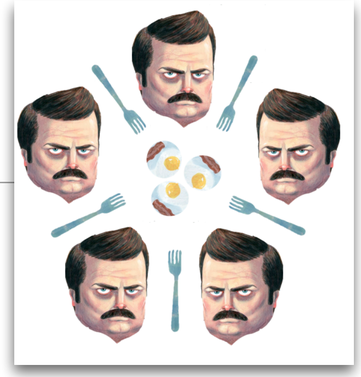
  Why or Why Not? Any problems? Advantages? Disadvantages?

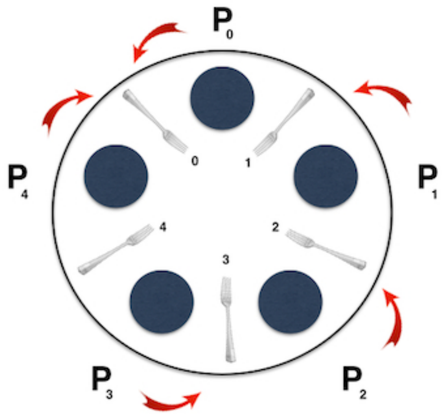


- i.e., How to "break" one of these conditions?
- **Ex.** Make *Circular Wait* (requirement 4) impossible (impose a linear ordering).

  Each Ron must pick up the lowest numbered fork first.
  - *Advantages? Disadvantages?*

Deadlock **Prevention** — *disallow/prevent one of the deadlock conditions*

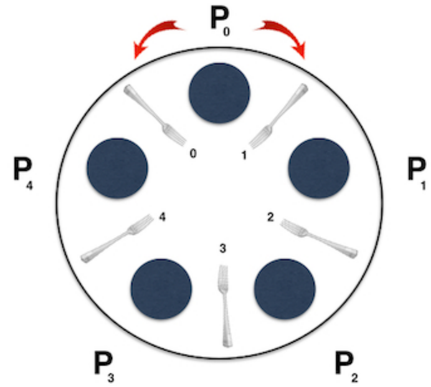




- Can we "prevent"…
  1. Mutual Exclusion?
  2. Hold and Wait?
  3. No Preemption?
  4. Circular Wait?

  Why or Why Not? Any problems? Advantages? Disadvantages?
- i.e., How to "break" one of these conditions?
- **Ex.** Make *Circular Wait* (requirement 4) impossible (impose a linear ordering).

  Each Ron must pick up the lowest numbered fork first.
  - *Advantages? Disadvantages?*



- **Ex.** Make *Hold and Wait* (requirement 2) impossible (use a monitor).

  Each Ron must ask the "dining-hall manager" to pick up both forks; otherwise, Ron must wait until he can pick up both forks.

  (i.e., picking up forks is essentially an atomic operation thanks to the dining-hall monitor)
  - *Advantages? Disadvantages?*

Deadlock **Avoidance*** *— do not grant resource request if it might lead to deadlock*

- What does it mean to "avoid"…
  1. Mutual Exclusion?
  2. Hold and Wait?
  3. No Preemption?
  4. Circular Wait?

➔***Allow necessary conditions for deadlock, BUT don't allocate resources if they may lead to deadlock***

NOTE: *Avoidance is indeed similar to prevention…*

Deadlock **Avoidance*** *— do not grant resource request if it might lead to deadlock*

- What does it mean to "avoid"…
  1. Mutual Exclusion?
  2. Hold and Wait?
  3. No Preemption?
  4. Circular Wait?

➔***Allow necessary conditions for deadlock, BUT don't allocate resources if they may lead to deadlock***

- Approaches:
  - ***Don't start a process*** *if its demands will lead to deadlock…*
    - *not ideal (must know ALL resources before starting);*
    - *not optimal (assumes the WORST about a process's use of resources)*
  - ***Don't grant incremental requests*** *for more resources if requests will lead to deadlock…*
    - *Banker's Algorithm*

NOTE: *Avoidance is indeed similar to prevention…*

Deadlock **Avoidance*** *— do not grant resource request if it might lead to deadlock*

Banker's Problem

*A bank has a limited reserve of money to lend **(resources)** and a list of customers **(processes)**, each with a line of credit. A banker must decide how to lend money.*

***State** = current allocations*

vs.

***Safe State** = there exists at least 1 sequence that does not result in deadlock*

vs.

***Unsafe Sate** = a state that is not safe ;)*

A Strategy: The Banker's Algorithm

*When a process makes a request for a set of resources, **assume** the request is granted, **update** the system state accordingly, then **determine** if the result is a safe state.*

***Yes?** Grant the request!*

***No?** Block the process until it is safe to grant the request*

Deadlock **Avoidance*** *— do not grant resource request if it might lead to deadlock*

A Strategy: The Banker's Algorithm

*When a process makes a request for a set of resources, **assume** the request is granted, **update** the system state accordingly, then **determine** if the result is a safe state.*

***Yes?** Grant the request!*
***No?** Block the process until it is safe to grant the request*

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Advantages? Disadvantages?

Deadlock **Detection** — *grant resource requests when possible; periodically check for deadlock + take action to recover*

- ***Allow*** all necessary conditions for deadlock,
  BUT periodically try to detect if deadlock has occurred (and fix if necessary)
- What does it mean to "detect" deadlock?
  - Specifically, detect ***Circular Wait***…

Deadlock **Detection** — *grant resource requests when possible; periodically check for deadlock + take action to recover*

- ***Allow*** all necessary conditions for deadlock,
  BUT periodically try to detect if deadlock has occurred (and fix if necessary)
- What does it mean to "detect" deadlock?
  - Specifically, detect ***Circular Wait***…

Deadlock Detection Algorithm

- Intuition:
  - Account for all possibilities of sequences of the tasks that remain to be completed.
  - "Mark" processes not contained in a "deadlock set"
  - Use ***Allocation Matrix (A)*** & ***Availability Vector (V)***
  - Also, ***Request Matrix (Q)***
  - Deadlock exists iff there are unmarked processes at the end of the algorithm

```
1.  Remove each proc i where if A(i,*) == 0
2.  Init W = V
3.  Find i s.t. i is not marked and
          Q_ik <= W_k,
             for 1 <= k <= m
    if i == NULL
       TERMINATE
    else
        mark i
        W_k = W_k + A_ik,
           for 1 <= k <= m
        repeat step 3.
```

Deadlock **Detection** — *grant resource requests when possible; periodically check for deadlock + take action to recover*

- ***Allow*** all necessary conditions for deadlock,
  BUT periodically try to detect if deadlock has occurred (and fix if necessary)
- What does it mean to "detect" deadlock?
  - Specifically, detect ***Circular Wait***…

Deadlock Detection Algorithm

- Intuition:
  - Account for all possibilities of sequences of the tasks that remain to be completed.
  - "Mark" processes not contained in a "deadlock set"
  - Use ***Allocation Matrix (A)*** & ***Availability Vector (V)***
  - Also, ***Request Matrix (Q)***
  - Deadlock exists iff there are unmarked processes at the end of the algorithm

```
1.  Remove each proc i where if A(i,*) == 0
2.  Init W = V
3.  Find i s.t. i is not marked and
          Q_ik <= W_k,
             for 1 <= k <= m
    if i == NULL
       TERMINATE
    else
        mark i
        W_k = W_k + A_ik,
           for 1 <= k <= m
        repeat step 3.
```

How to RECOVER if deadlock is detected?

Advantages? Disadvantages?

Deadlock - **Hybrid/Integrated Strategies**

- Group resources into different ***resource classes***
- Use linear ordering to prevent circular wait ***between classes***
- Use appropriate strategy/algorithm ***within each class***

Example:

Interclass Strategy: Group resources into classes & assign resources in order of classes

- Class 1 = Swappable Space
  - ***Intraclass Strategy:*** require all resources to be allocated at one time (hold-and-wait prevention strategy)
- Class 2 = Process Resources
  - ***Intraclass Strategy:*** use deadlock avoidance algorithm
- Class 3 = Main Memory
  - ***Intraclass Strategy:*** prevention by preemption; swap process to secondary memory if memory resources need to be freed up
- Class 4 = Internal Resources
  - ***Intraclass Strategy:*** prevention by means of resource ordering

Summary of Approaches for Addressing Deadlock

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	•Works well for processes that perform a single burst of activity •No preemption necessary	•Inefficient •Delays process initiation •Future resource requirements must be known by processes
		Preemption	•Convenient when applied to resources whose state can be saved and restored easily	•Preempts more often than necessary
		Resource ordering	•Feasible to enforce via compile-time checks •Needs no run-time computation since problem is solved in system design	•Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	•No preemption necessary	•Future resource requirements must be known by OS •Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	•Never delays process initiation •Facilitates online handling	•Inherent preemption losses

See Also…

UNIX/Linux Concurrency Mechanisms *(Sections 6.7-6.8)*

- Pipes
- Messages
- Shared Memory
- Semaphores *(Binary, Counting, and Reader-Writer Semaphores)*
- Signals *(Simple Signals vs. Real-Time Signals)*
- Atomic Operations on Integers & Bitmaps
- Spinlocks *(a variety of flavors…)*
- Memory Barriers *(Prevent compiler and/or processor from reordering instructions)*
- RCU (Read-Copy-Update)
  - *Writer copies the shared resource, edits it, and assigns pointer to the updated version AFTER all readers are done*
  - *Useful when reads are frequent and writes are rare*
- …