Deadlock

David Ferry
CSCI 2510 – Principles of Computing Systems
Saint Louis University
St. Louis, MO 63103

SAINT LOUIS UNIVERSITY™

Blocking Operations

Recall- Some operations cause a program to wait.

Quiz: Which of the following operations could cause blocking?

A. Locking a mutex
B. Opening a file
C. Reading a file
D. Reading from a pipe
E. Writing to a pipe

Blocking Operations

Recall- Some operations cause a program to wait.

Quiz: Which of the following operations could cause blocking?

A. Locking a mutex (already locked)
B. Opening a file (disk I/O)
C. Reading a file (disk I/O)
D. Reading from a pipe (pipe is empty)
E. Writing to a pipe (pipe is full)

Deadlock

A specific hazard occurs with blocking operations:

- Could an adversary scheduler break our code?

```
mutex a, b;
```

Thread 1:

```
lock(a)
lock(b)
//Critical section
unlock(b)
unlock(a)
```

Thread 2:

```
lock(b)
lock(a)
//Critical section
unlock(a)
unlock(b)
```

Deadlock

A specific hazard occurs with blocking operations:

- Could an adversary scheduler break our code?

mutex a, b;

Thread 1:

1. lock(a)
4. lock(b)

    //Critical section
    unlock(b)
    unlock(a)

Infinite wait

Thread 2:

2. lock(b)
3. lock(a)

    //Critical section
    unlock(a)
    unlock(b)

Classic Deadlock Example: Dining Philosophers

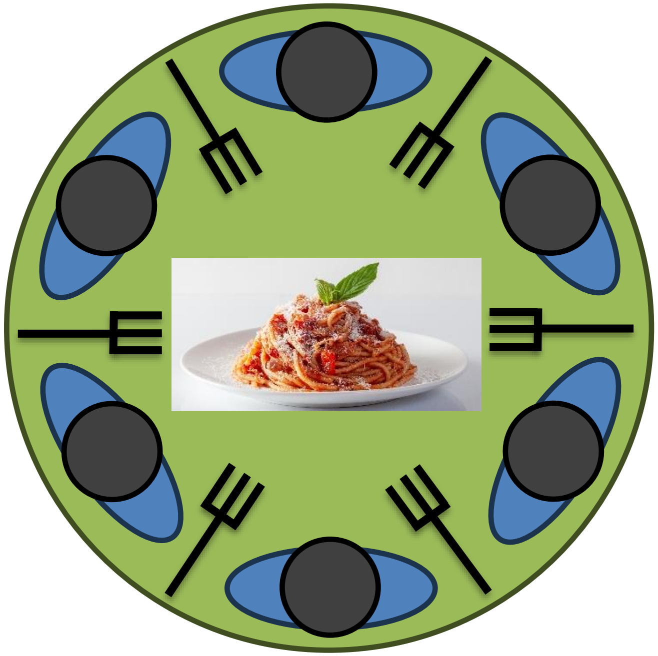
A group of philosophers are sharing a delicious spaghetti meal.

Six thinkers, six forks, each needs two forks to eat.

Each philosopher follows the procedure:

while(1){
1. Ponder mysteries for a while
2. Grab fork to their left *(or wait)*
3. Grab fork to their right *(or wait)*
4. Eat
5. Release forks

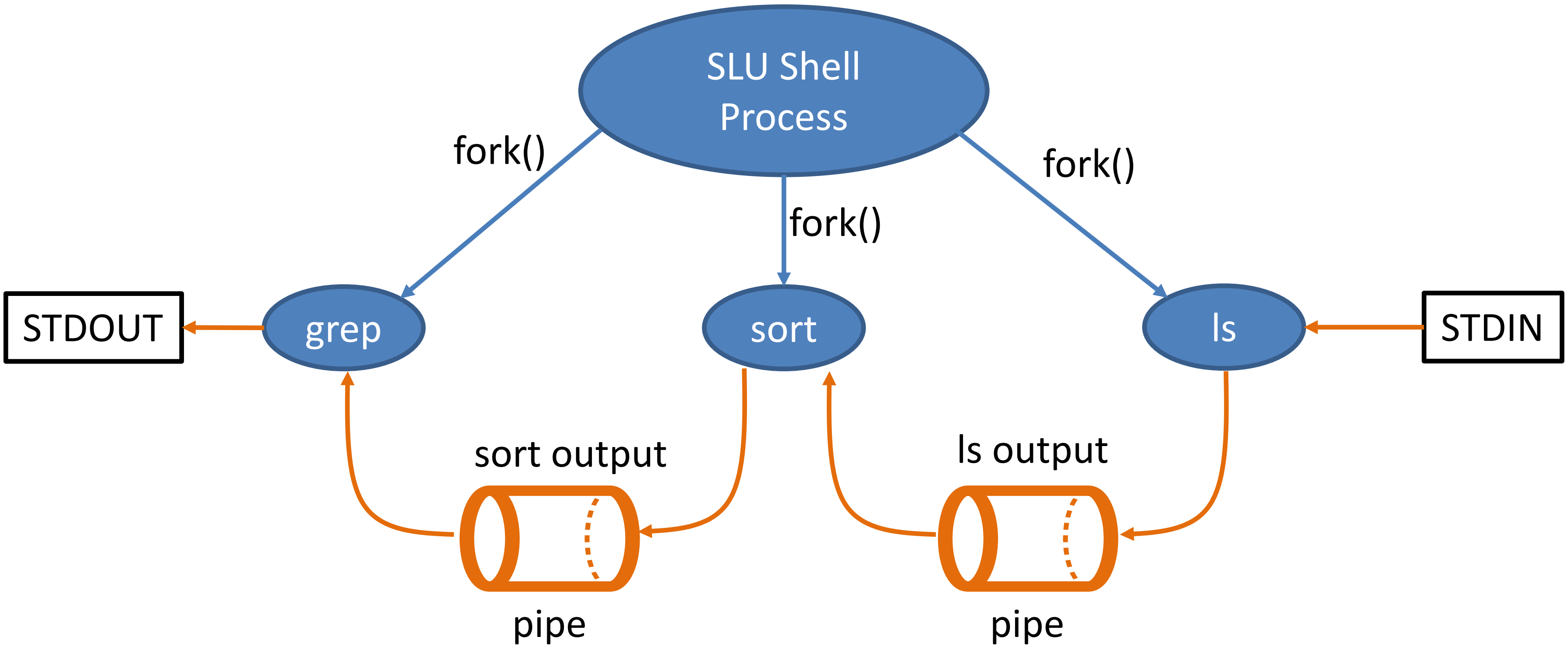
}



Spaghetti: Powerpoint Stock Photos

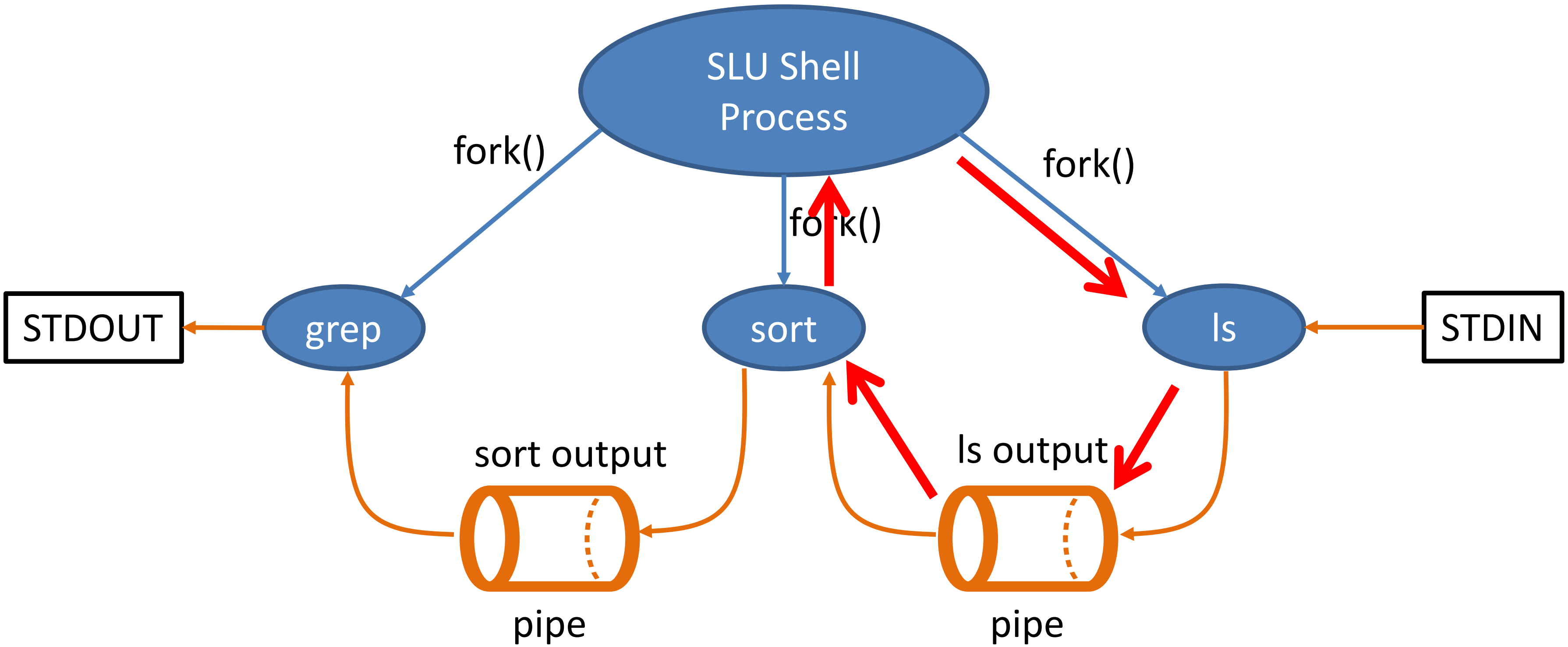
Recall – Lab 2

How could deadlock occur if the parent shell process *waits* on each child after forking instead of just the last?



Recall – Lab 2

How could deadlock occur if the parent shell process *waits* on each child after forking instead of just the last?



Necessary Conditions for Deadlock

Which of the following elements occurred in all three examples?

Philosophers	Locking
Graph cycles	Mutex variables
Processes	Waiting
Threads	Pipes
Resource exclusion	Spaghetti

Necessary Conditions for Deadlock

Which of the following elements occurred in all three examples?

Philosophers

Graph cycles

Processes

Threads

Resource exclusion

Locking

Mutex variables

Waiting

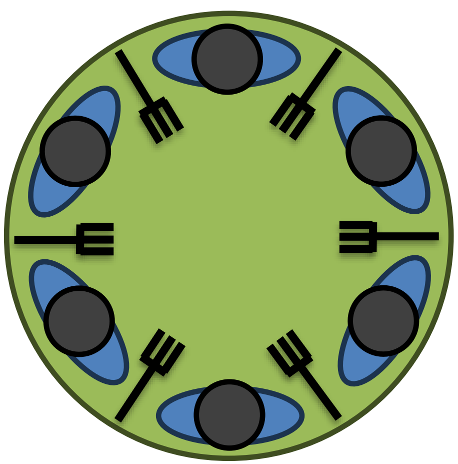
Pipes

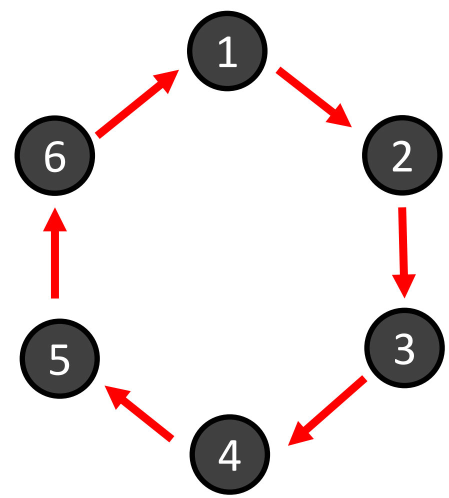
Spaghetti

Deadlock is not just a property of threads, locks, and mutexes- it is a general systems problem.

Deadlock Conditions

1. Resource exclusion – Processes can hold resources in a non-shareable state
2. Hold-while-waiting – Processes can hold a resource while waiting for another resource
3. No preemption – Only the process holding a resource can release it
4. Circular wait – There can exist a set of waiting processes $P_1 \ldots P_N$ such that P_1 waits on P_2, P_2 waits on P_3, … P_{N-1} waits on P_N, and P_N waits on P_1



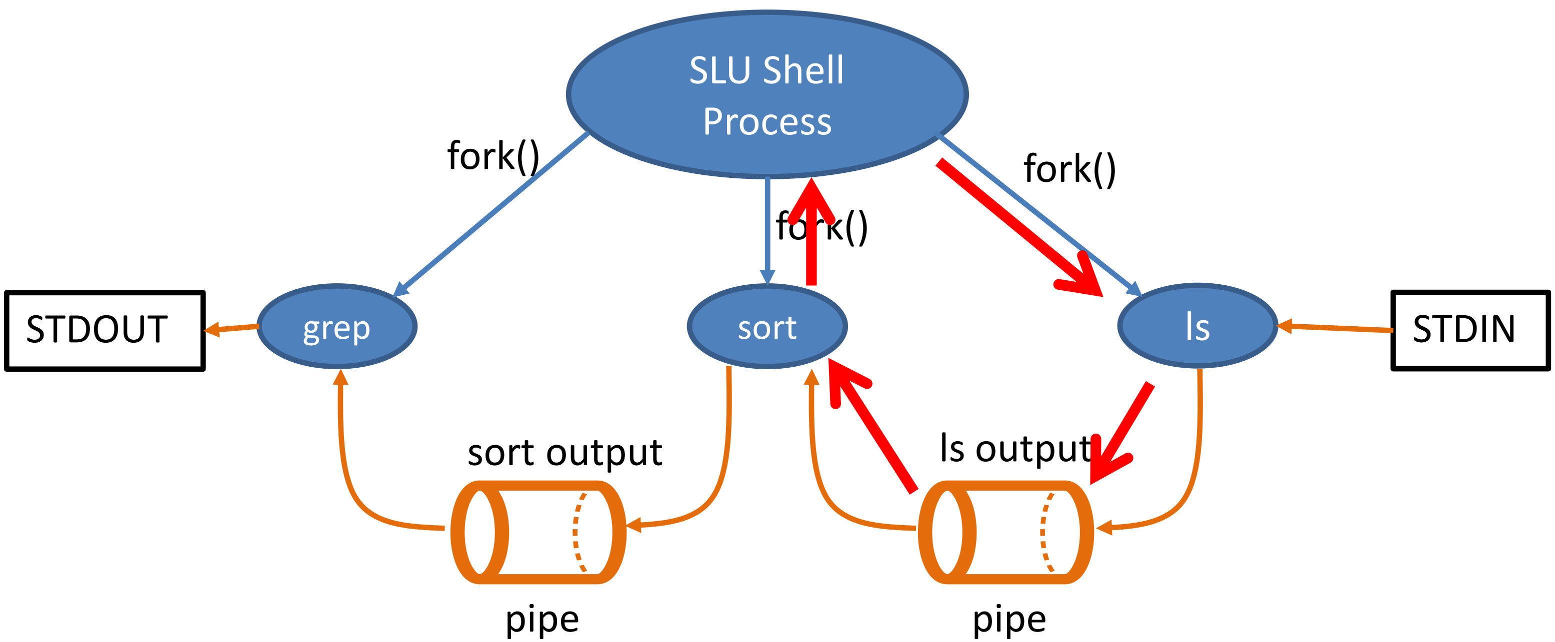


E. G. Coffman. System Deadlocks. ACM Comput. Surv. 3, 2 (June 1971), 67-78.

Deadlock does not require explicit resource acquisition

The shell process calls wait() on the first child process, the child process calls write() on the pipe. The pipe is full. No process ever explicitly acquires a resource.

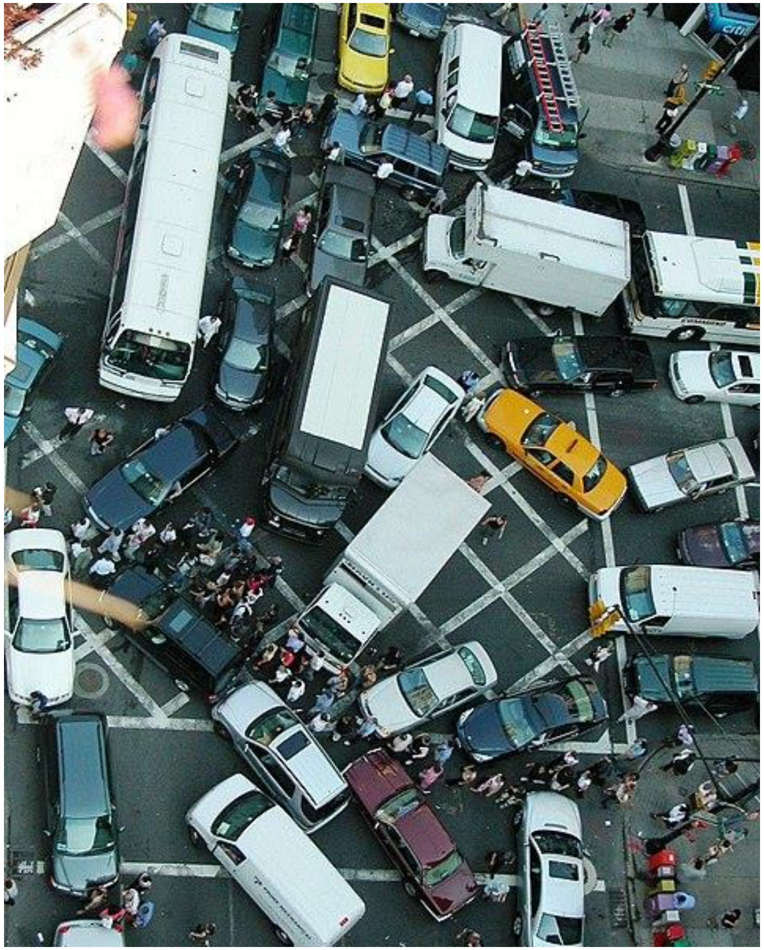
- Possibility of deadlock is an inherent property of a design
- That said, most of the time we talk about explicit acquisition



Real-World Example: Gridlock

Is this deadlock?

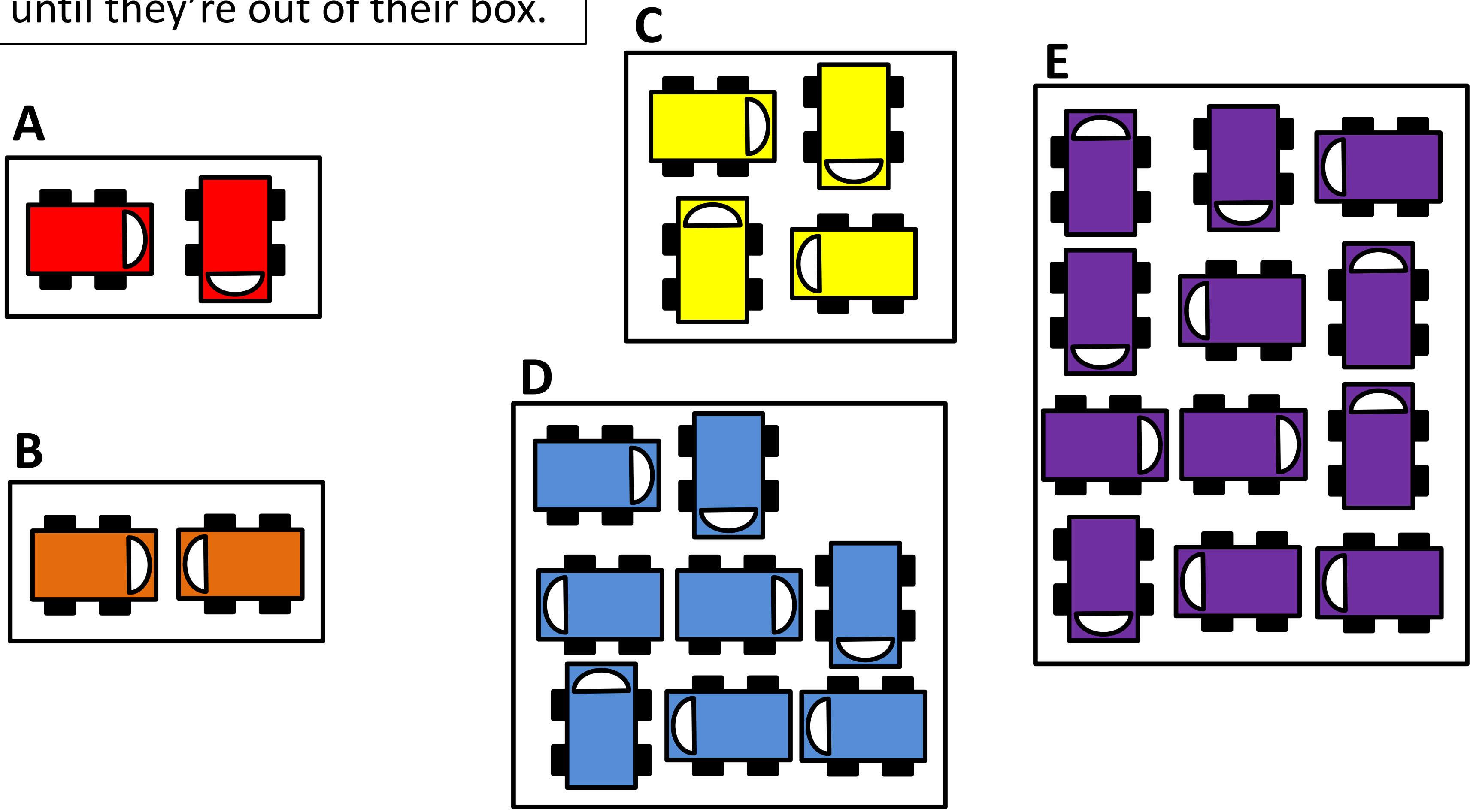
- Resource exclusion
- Hold-while-waiting
- No preemption
- Circular wait



Attribution: Rgoogin at the English Wikipedia

Which of these contain Deadlock?

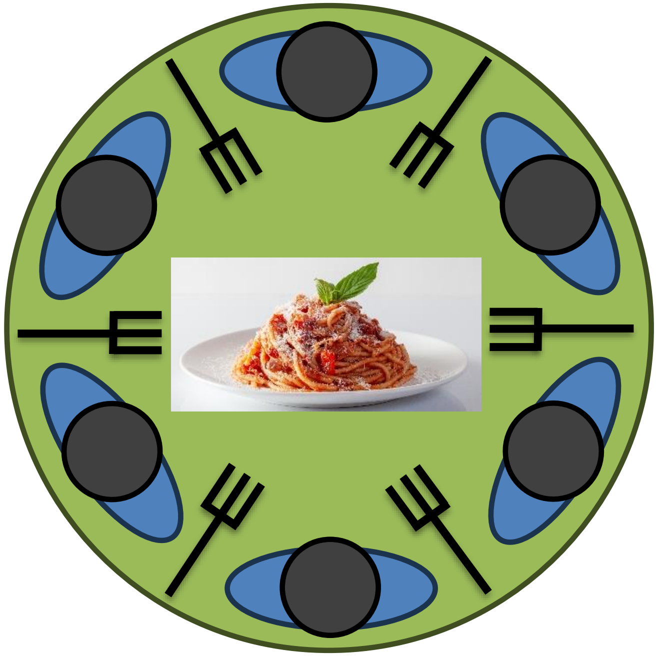
Rule: Cars must drive in a straight line until they're out of their box.

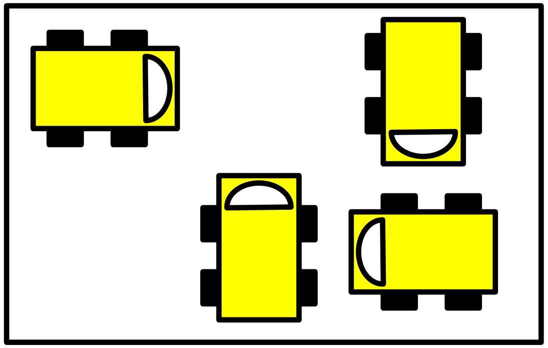


Can Deadlock is not Will Deadlock

Deadlock may or may not occur during any specific execution

- Usually results from a race, so depends on timing
- Just like race conditions, thorough testing is not guarantee of finding deadlock
- Developer must reason about the system and convince themselves the system is deadlock-free
- E.g. Philosophers don't always deadlock





Exercise: Give an execution for Dining Philosophers that doesn't deadlock

Starvation

Related concept, will only introduce here:

- Starvation – Individual processes may be "unlucky" and unable to progress
- Example: Threads race and 1 never gets to lock(a)

```
mutex a, b
Thread 1
   lock(a)
   lock(b)
   //process data
   unlock(b)
   unlock(a)

Thread 2
   lock(a)
   lock(b)
   //process data
   unlock(b)
   unlock(a)

Thread 3
   lock(a)
   lock(b)
   //process data
   unlock(b)
   unlock(a)
```

Livelock

Related concept, will only introduce here:

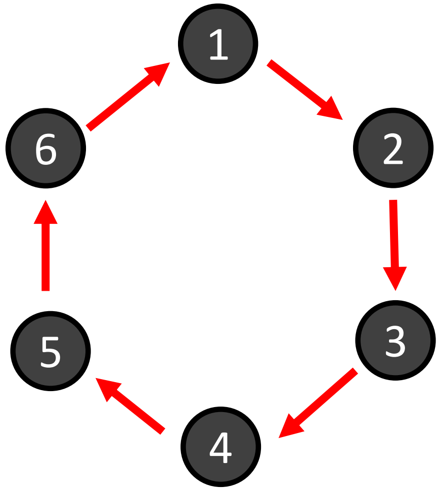
- Livelock – Threads aren't blocked/waiting but also never make progress
- Example: Suppose we have "polite threads" that release their first lock if not able to acquire the second lock.

Thread 1: Lock a
Thread 2: Lock b
Thread 1: Try to lock b and fail
Thread 2: Try to lock a and fail
Thread 1: Release a
Thread 2: Release b
Repeat

Deadlock Analysis

Deadlock scenarios are usually analyzed with a *directed graph* called a *resource allocation graph:*

Recall: A directed graph G = (V,E) is a set of vertices V and directed edges E, such that an edge a→b does not imply an edge b→a.



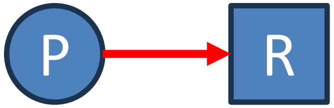
Example:
V = {1, 2, 3, 4, 5, 6}
E = {1→2, 2→3, 3→4,
4→5, 5→6, 6→1}

Resource Allocation Graph

The RAG has two sets of vertices-

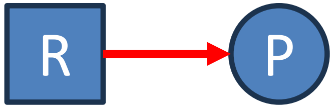
- A set processes/threads drawn as circles
- A set of resources drawn as squares

Edge from P to R: P is trying to acquire R



P is waiting for R

Edge from R to P: P is holding R



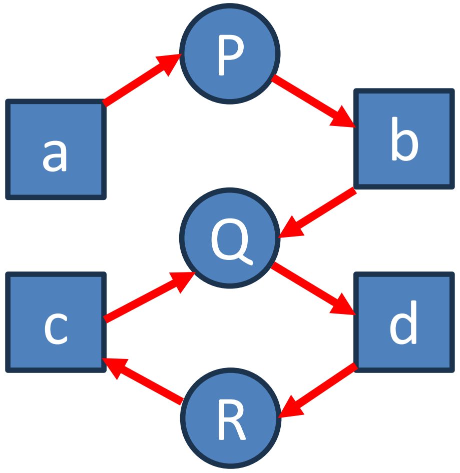
R is held by P

Deadlock Detection

Our first three deadlock conditions reflect semantics of the system which do not change.

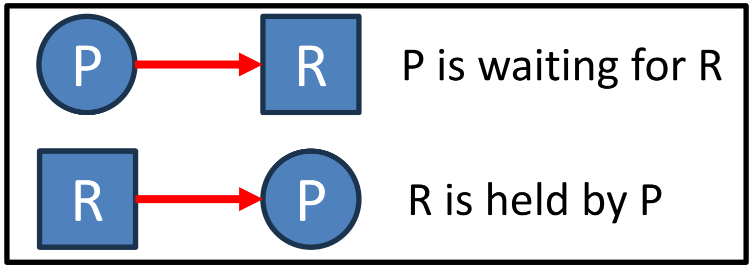
Whether a circular wait exists is a property of the runtime state of a system and does change.

- A circular chain of dependences in the RAG indicates deadlock
- Edges will alternate between processes and resources
- Never have edges between pairs of processes or pairs of resources
- Will never have more than one edge originating at a resource



The Deadly Embrace

A cycle in the RAG indicates deadlock!



mutex a, b;

Thread P:

1. lock(a)
4. lock(b)

Infinite wait

Thread Q:

2. lock(b)
3. lock(a)

RAG:

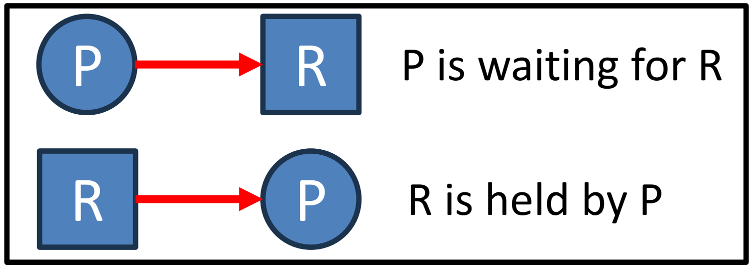
Processes =

Resources =

Edges =

The Deadly Embrace

A cycle in the RAG indicates deadlock!



mutex a, b;

Thread P: Thread Q:

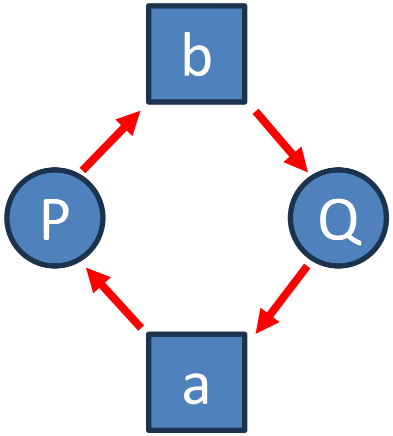
1. lock(a) — *Infinite wait* — 2. lock(b)

4. lock(b) — 3. lock(a)

RAG:

Processes = { P, Q } Resources = {a, b}

Edges = { $a \rightarrow P$, $b \rightarrow Q$, $Q \rightarrow a$, $P \rightarrow b$ }

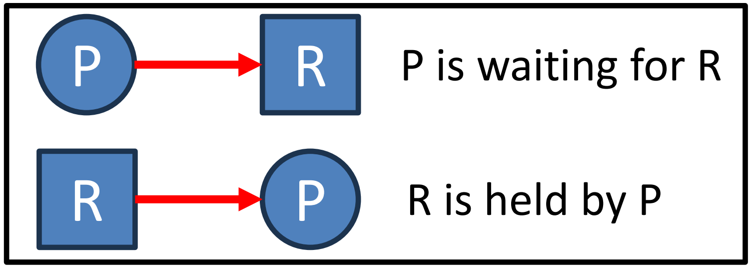


Exercise

Does the following sequence result in deadlock? If so, at which step does the system deadlock?

Processes = { P, Q, R} Resources = {a, b, c, d, e}

1 P.acquire(a)
2 Q.acquire(c)
3 R.acquire(b)
4 P.acquire(c)
5 Q.acquire(d)
6 R.acquire(a)
7 Q.acquire(b)
8 R.acquire(e)

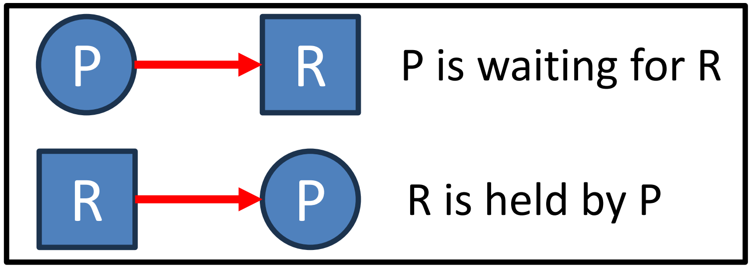


Lab 4

Does the following sequence result in deadlock? If so, at which step does the system deadlock?

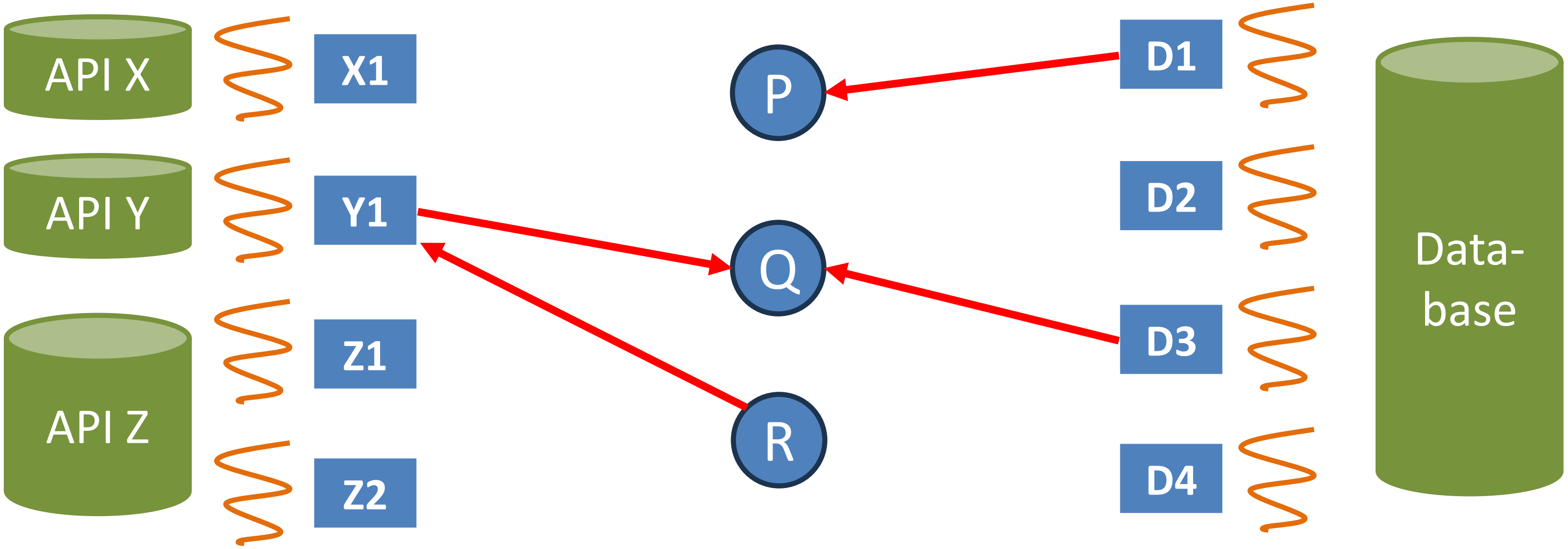
Processes = {0, 1, 2} Resources = {0, 1, 2, 3, 4}

1: 0 a 0
2: 1 a 2
3: 2 a 1
4: 0 a 2
5: 1 a 3
6: 2 a 0
7: 1 a 1
8: 2 a 4



Single Resource vs. Multi-Resource

- So far, we've only considered the case where we have exactly one of each type of resource.
- In general, we may have multiple resources of a given type, and each can be claimed separately.
- Example: Web services with different numbers of runners.



Multi-Resource Deadlock Avoidance: Banker's Algorithm

Never let the system get into an *unsafe state.*

Intuition from banking:

- Never let the bank loan out so much money they couldn't give everyone their deposits back if everyone came back at once.

Intuition for systems:

- Never grant resource requests if it means we might not be able to satisfy all possible resource requests.

Banker's Algorithm

1. Processes declare the maximal set of all resources they might need before executing.
2. When an acquisition request occurs, the system checks to see whether the system would become *unsafe*.
   - The system is *safe* if there is *some* execution path that allows all processes to finish.
   - Must be able to satisfy maximum request of *some* process.
3. If unsafe, denies the request or blocks requesting process. If safe, grants it.

Banker's Algorithm: Example

Left matrix: How many resources each process currently has
Right matrix: Maximum number of additional resources may be needed

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

Currently Has

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

May Need

Total Resources = { 7, 3, 4 }
Available = { 1, 0, 2 }
Possessed = { 6, 3, 2 }

Q: Is the system in a *safe* state?

The system is currently *safe* because we can find a sequence of completions that satisfy all processes:

- Look for some row that can be satisfied by currently available resources

- P4 can finish, so Available = { 2, 1, 2 }
- P2 can finish, so Available = { 3, 2, 2 }
- P1 can finish, so Available = { 6, 2, 3 }
- P3 can finish, and all processes are done

Banker's Algorithm: Example

Left matrix: How many resources each process currently has
Right matrix: Maximum number of additional resources may be needed

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

Currently Has

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

May Need

Total Resources = { 7, 3, 4 }
Available = { 1, 0, 2 }
Possessed = { 6, 3, 2 }

Q: Is it safe for P2 to acquire an R3?

- Yes- the available resources would then be { 1, 0, 1 }, so P4 could still execute and the solution from the last slide is still valid.

Q: Is it safe for P3 to acquire an R1?

- No, then the available resources would be { 0, 0, 2 }, which cannot satisfy any row in the May Need matrix. Deadlock may result.

Deadlock Prevention

Avoidance not practical! Even if we could forecast all future possible needs, likely to be very conservative!

We can prevent deadlock by breaking one of the four necessary conditions.

- Resource Exclusion
- Wait-and-hold
- No preemption
- Circular wait

Take a moment to consider how we could remove these from the system…

Deadlock Prevention

Resource Exclusion

- Not much we can do here…

Wait-and-hold

- Require processes to acquire resources as a set

No preemption

- If a process would wait, it releases all currently held resources and tries again

Circular wait

- Impose a linear order on the sequence that resources are requested

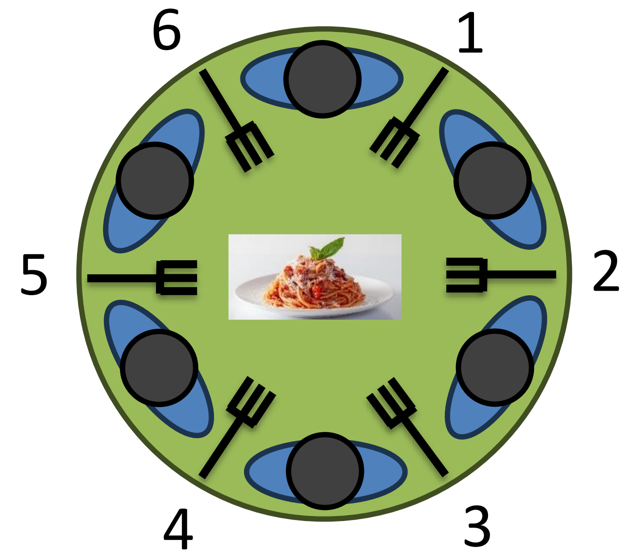
These strategies may not be workable in real systems!

E. G. Coffman. System Deadlocks. ACM Comput. Surv. 3, 2 (June 1971), 67-78.

Deadlock Prevention

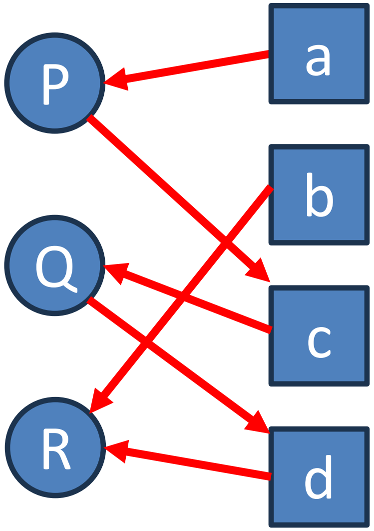
What would these prevention algorithms mean for the dining philosophers?

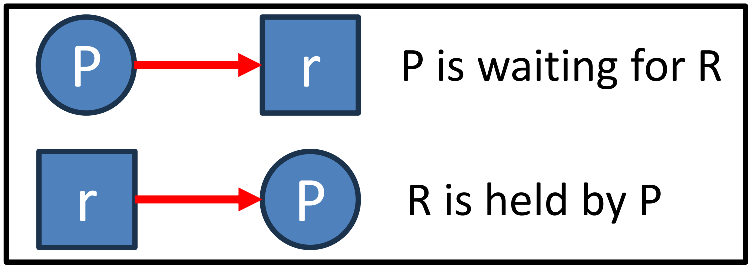
- Require resources to be acquired as a set
- If a process would wait, release resources and try again
- Impose a linear sequence on the order of resource acquisition
  1 -> 2 -> 3 -> 4 -> 5 -> 6



Linear Ordering

- Suppose resources {a, b, c, d} can only be acquired in the order a->b->c->d.
- If a process only needs a and c, it doesn't have to request b.
- How does this break circular wait?
- The process with the highest-ordered resource is always guaranteed to be able to make progress.
- How can this RAG untangle itself?





Deadlock Recovery

1. Periodically run an algorithm to detect deadlock after it has occurred
2. Take corrective action
   – Preempt processes involved
   – Roll back to a known good checkpoint
   – Kill the processes involved and restart

Hard problem in general. Preventing deadlocks may be too conservative or computationally intensive. Recovering from deadlocks after they occur is messy.

Conclusion

- Deadlock occurs as a consequence of system design & system semantics
- Four necessary criteria:
  - Resource exclusion
  - Hold-and-wait
  - No Preemption
  - Circular Wait
- Deadlocks can be detected with RAG search
- Deadlocks can be avoided with Banker's Algorithm
- Deadlocks might be prevented by changing system design or semantics
- Sometimes, the best we can do is to detect deadlocks and try to fix after the fact