Theory of Operating Systems

Week 4 Part 2: Deadlocks

Faye Chi Zhang, Richard Cai, and Dr. Ping Ji

CUNY Hunter College

February 25, 2026

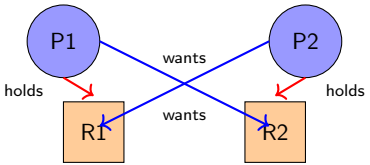
Agenda

1. Deadlocks
   - Resource Allocation Graph
   - Deadlock Handling Strategies
2. Summary

The Deadlock Problem

Definition

A set of processes is **deadlocked** if each process is waiting for an event that only another process in the set can cause.



Real-world analogy:

- Two cars at a narrow bridge
- Neither can proceed
- Neither will back up
- Traffic jam forever

Deadlock Example in Code

```
pthread_mutex_t lock_a = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock_b = PTHREAD_MUTEX_INITIALIZER;

void *thread1(void *arg) {
    pthread_mutex_lock(&lock_a);    // Acquire A
    sleep(1);                        // Simulate work
    pthread_mutex_lock(&lock_b);    // Wait for B (held by thread2)
    // ... critical section ...
    pthread_mutex_unlock(&lock_b);
    pthread_mutex_unlock(&lock_a);
}

void *thread2(void *arg) {
    pthread_mutex_lock(&lock_b);    // Acquire B
    sleep(1);                        // Simulate work
    pthread_mutex_lock(&lock_a);    // Wait for A (held by thread1)
    // ... critical section ...
    pthread_mutex_unlock(&lock_a);
    pthread_mutex_unlock(&lock_b);
}
```

Result: Both threads wait forever!

The Four Coffman Conditions (1971)

Deadlock can occur if and only if ALL four hold simultaneously:

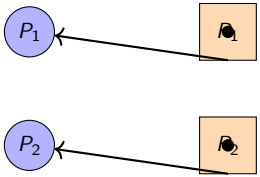
1. **Mutual Exclusion**
   - At least one resource is non-sharable
2. **Hold and Wait**
   - Process holds resource(s) while waiting for others
3. **No Preemption**
   - Resources cannot be forcibly taken away
4. **Circular Wait**
   - Circular chain: $P_0 \to P_1 \to \ldots \to P_n \to P_0$

Key Insight

Break **any one** condition $\Rightarrow$ deadlock is impossible.

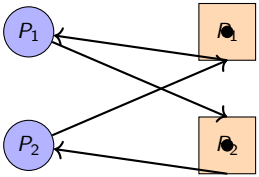
Resource Allocation Graph (RAG)

No Cycle = No Deadlock



Both have resources, no waiting.

Cycle = Possible Deadlock



Cycle detected = Deadlock!

Important

Single-instance resources: Cycle $\Leftrightarrow$ Deadlock.

Multi-instance resources: Cycle $\Rightarrow$ **Possible** Deadlock (not guaranteed).

Approaches to Handling Deadlock

1. **Prevention**: Ensure at least one Coffman condition cannot hold
2. **Avoidance**: Dynamically avoid unsafe states (Banker's algorithm)
3. **Detection & Recovery**: Allow deadlock, then detect and fix
4. **Ignorance** (Ostrich algorithm): Pretend it doesn't happen

Strategy	Overhead	Restriction
Prevention	Low	High
Avoidance	Medium	Medium
Detection	High	None
Ignorance	None	None

In practice: Most general-purpose OS (Linux, Windows, macOS) use the Ostrich algorithm – deadlock is rare enough that rebooting is cheaper than prevention.

Prevention: Lock Ordering

Most practical prevention: Break **circular wait** via total ordering

```
// Define global ordering: lock_a < lock_b
pthread_mutex_t lock_a, lock_b;

void *thread1(void *arg) {
    pthread_mutex_lock(&lock_a);  // Always acquire a first
    pthread_mutex_lock(&lock_b);
    // Critical section
    pthread_mutex_unlock(&lock_b);
    pthread_mutex_unlock(&lock_a);
}

void *thread2(void *arg) {
    pthread_mutex_lock(&lock_a);  // Same order!
    pthread_mutex_lock(&lock_b);
    // Critical section
    pthread_mutex_unlock(&lock_b);
    pthread_mutex_unlock(&lock_a);
}
```

No deadlock possible: Both threads acquire in same order.

Banker's Algorithm (Dijkstra)

Avoidance: Before granting a request, check if it leads to a **safe state**

Data structures (n processes, m resource types):

- `Available[m]`: Available instances of each resource
- `Max[n][m]`: Maximum demand of each process
- `Allocation[n][m]`: Currently allocated
- `Need[n][m]`: Remaining need (`Max - Allocation`)

Safety check: Find a sequence where each process can finish using available + released resources.

- Safe $\Rightarrow$ Grant request
- Unsafe $\Rightarrow$ Deny and wait

Banker's Algorithm Example

System: 3 resource types (A, B, C) with 10, 5, 7 instances

	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	7	4	3
P_1	2	0	0	3	2	2	1	2	2
P_2	3	0	2	9	0	2	6	0	0
P_3	2	1	1	2	2	2	0	1	1
P_4	0	0	2	4	3	3	4	3	1

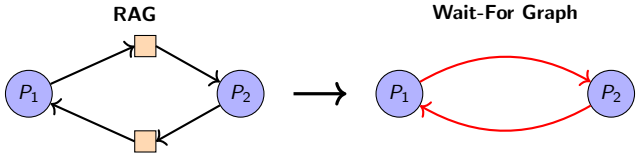
Available: (3, 3, 2)

Safe sequence: $\langle P_1, P_3, P_4, P_2, P_0 \rangle$

- P_1: Need(1,2,2) $\leq$ Avail(3,3,2) ✓ $\rightarrow$ Avail = (5,3,2)
- P_3: Need(0,1,1) $\leq$ Avail(5,3,2) ✓ $\rightarrow$ Avail = (7,4,3)
- Continue until all finish...

Detection and Recovery

Detection: Use Wait-For Graph (simplify RAG by removing resource nodes)



Cycle in WFG $\Leftrightarrow$ **Deadlock**. Detection via DFS: $O(n^2)$.

Recovery options:

- Terminate deadlocked processes (one at a time or all)
- Preempt resources and rollback

Practice Exercise 6: Coffman Conditions

Question 6

For each scenario, identify which Coffman condition is violated (making deadlock impossible):

1. A system that uses read-write locks where data is mostly read-only.
2. A protocol requiring processes to release all locks before requesting new ones.
3. A system where the OS can forcibly reclaim memory from processes.
4. All threads acquire locks in alphabetical order by lock name.

Answers:

1. **Mutual Exclusion** violated – reads are sharable
2. **Hold and Wait** violated – must release before requesting
3. **No Preemption** violated – OS can preempt
4. **Circular Wait** violated – total ordering prevents cycles

Practice Exercise 7: Banker's Algorithm

Question 7

Given 2 resource types (A, B) with Available = (1, 1):

	Alloc		Max		Need	
	A	B	A	B	A	B
P_0	1	0	2	1	1	1
P_1	0	1	1	2	1	1

Is this state safe? If P_0 requests (0, 1), should it be granted?

Answer: Current state is safe: $\langle P_0, P_1 \rangle$ works.
If P_0 requests (0,1): Avail becomes (1,0). Need_0=(1,0) $\leq$ (1,0) ✓.
After P_0: Avail = (1,0)+(1,1) = (2,1). Need_1=(1,1) $\leq$ (2,1) ✓.
Safe $\Rightarrow$ Grant the request.

Key Takeaways

1. **Threads**: Lightweight execution units sharing address space
   - pthreads API: create, join, detach, exit
2. **Synchronization Primitives**:
   - Mutex: Mutual exclusion (owner unlocks)
   - Semaphore: Counting + signaling (no ownership)
   - Condition Variable: Wait for condition (use with mutex + while loop)
3. **Classic Problems**: Producer-Consumer, Dining Philosophers
4. **Deadlock**: Four Coffman conditions must all hold
   - Prevention: Lock ordering (most practical)
   - Avoidance: Banker's algorithm
   - Detection: Wait-for graph cycle detection

Synchronization Quick Reference

Primitive	Purpose	Key Operations	Use Case
Mutex	Mutual exclusion	lock, unlock	Protect shared data
Semaphore	Counting/signaling	wait, signal	Resource pools, signaling
Cond Var	Wait for condition	wait, signal, broadcast	Producer-consumer
RW Lock	Reader-writer	rdlock, wrlock, unlock	Read-heavy workloads
Barrier	Sync point	wait	Parallel phases
Spinlock	Short CS	lock, unlock	Kernel, short waits

Decision Guide

- Need exclusive access? $\rightarrow$ Mutex
- Need to limit concurrent access? $\rightarrow$ Counting semaphore
- Need to wait for a condition? $\rightarrow$ Condition variable + mutex
- Many readers, few writers? $\rightarrow$ Read-write lock

This Week's Tasks

- **Quiz 4**: Threads, synchronization, deadlock conditions
- **Reading**: Textbook chapters on threads and deadlocks
- **Practice**:
  - Write a multithreaded program with race condition, then fix it
  - Implement producer-consumer with semaphores
  - Apply Banker's algorithm by hand on sample problems
  - Draw RAGs and identify deadlocks

Questions?