# Theory of Operating Systems

## Week 5 Part 2: Synchronization Primitives & Deadlocks

Faye (Chi Zhang)

CUNY Hunter

February 25, 2026

# Today's Agenda

1. Mutex (Review & Depth)
2. Semaphores
3. Condition Variables
4. Classic Synchronization Problems
5. Deadlocks
6. Deadlock Prevention & Detection

# Mutex Recap

**Binary lock:** locked or unlocked

**Rules:**

- Only **one** thread holds it at a time
- Only the **owner** can unlock
- Other threads **block** on lock()

```
1 pthread_mutex_t m =
2   PTHREAD_MUTEX_INITIALIZER;
3
4 pthread_mutex_lock(&m);
5 // critical section
6 pthread_mutex_unlock(&m);
```

## Key Property

Provides **mutual exclusion** — at most one thread in the critical section.

# Mutex Types

**NORMAL** Default. Deadlocks if same thread locks twice.

**ERRORCHECK** Returns error on double-lock (good for debugging).

**RECURSIVE** Same thread can lock multiple times; must unlock same number.

```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr,
    PTHREAD_MUTEX_ERRORCHECK);
pthread_mutex_t m;
pthread_mutex_init(&m, &attr);
```

# Semaphores

**Invented by Dijkstra (1965).**
A semaphore is an integer $S \geq 0$ with two atomic operations:

wait(S) / P() If $S > 0$, decrement $S$.
Otherwise block.

signal(S) / V() Increment $S$.
Wake one blocked thread.

**Binary semaphore** ($S \in \{0, 1\}$)
Acts like a mutex

**Counting semaphore** ($S \geq 0$)
Controls access to $N$ resources

**vs. Mutex**

Any thread can signal a semaphore (no ownership).

# POSIX Semaphores

```c
#include <semaphore.h>

sem_t sem;
sem_init(&sem, 0, 3);  // initial value = 3

sem_wait(&sem);   // P: decrement or block
// critical section (up to 3 threads)
sem_post(&sem);   // V: increment, wake waiter

sem_destroy(&sem);
```

- `sem_init(sem, pshared, value)` — 0 = thread-shared
- `sem_wait` blocks if value is 0
- `sem_post` never blocks
- `sem_trywait` returns error instead of blocking

# Semaphore Use Cases

**Mutual Exclusion** (binary, init=1)

1. `sem_wait(&s)` — enter CS
2. Critical section
3. `sem_post(&s)` — leave CS

**Resource Counting** (init=N)

- Database connection pool
- Bounded buffer slots

**Signaling / Ordering**

- Init semaphore to 0
- Thread A does work, then `sem_post`
- Thread B calls `sem_wait` — blocks until A signals

This enforces "A before B" ordering without a mutex.

# Condition Variables

**Problem:** How does a thread wait for an *arbitrary condition* (not just a lock)?

**Condition variable** = a queue of threads waiting for a condition to become true.

pthread_cond_wait(cv, mtx) Atomically: unlock mutex, sleep on cv.
On wakeup: re-acquire mutex.

pthread_cond_signal(cv) Wake **one** waiting thread.

pthread_cond_broadcast(cv) Wake **all** waiting threads.

**Always use with a mutex + while loop**

Spurious wakeups can occur — always recheck the condition.

# Condition Variable Pattern

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cv  = PTHREAD_COND_INITIALIZER;
int ready = 0;

// Waiter
pthread_mutex_lock(&mtx);
while (!ready)                      // WHILE, not IF
    pthread_cond_wait(&cv, &mtx); // unlock + sleep
// condition is true, mutex held
pthread_mutex_unlock(&mtx);

// Signaler
pthread_mutex_lock(&mtx);
ready = 1;
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mtx);
```

# Producer-Consumer (Bounded Buffer)

**Setup:**

- Shared buffer of size $N$
- Producers add items
- Consumers remove items
- Block when full/empty

**Synchronization:**

- `mutex` — protect buffer
- `empty` — sem init $N$
- `full` — sem init 0

**Producer:**

1. `sem_wait(&empty)`
2. `mutex_lock`
3. Add item to buffer
4. `mutex_unlock`
5. `sem_post(&full)`

**Consumer:**

1. `sem_wait(&full)`
2. `mutex_lock`
3. Remove item from buffer
4. `mutex_unlock`
5. `sem_post(&empty)`

# Readers-Writers Problem

**Multiple readers OR one writer.**

- Readers can share (concurrent reads OK)
- Writer needs exclusive access
- No reader while writing
- No writer while reading

**Solution with read-write lock:**

- `pthread_rwlock_rdlock`
  Shared (multiple readers)
- `pthread_rwlock_wrlock`
  Exclusive (one writer)
- `pthread_rwlock_unlock`

**Starvation Risk**

Reader-priority: writers may starve.
Writer-priority: readers may starve.

# Dining Philosophers

- 5 philosophers at a round table
- 5 forks between them
- To eat: pick up **both** adjacent forks
- Think $\rightarrow$ Hungry $\rightarrow$ Eat $\rightarrow$ Think

**Naïve solution:**

1. Pick up left fork
2. Pick up right fork
3. Eat
4. Put down both

$\Rightarrow$ **Deadlock** if all pick up left!

**Solutions:**

1. Allow at most 4 to sit
2. Pick up both forks atomically
3. Asymmetric: odd picks left first, even picks right first
4. Use a waiter (central mutex)

**Key Lesson**

Circular wait + hold-and-wait = deadlock.
Breaking either condition prevents it.

# Primitives Comparison

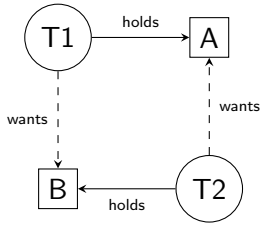
| | **Mutex** | **Semaphore** | **Cond Var** |
|---|---|---|---|
| Purpose | Mutual excl. | Counting/signal | Wait for condition |
| Ownership | Yes | No | N/A (uses mutex) |
| Value | 0 or 1 | $\geq 0$ | N/A |
| Can signal w/o hold? | No | Yes | Must hold mutex |
| Typical use | Protect data | Limit concurrency | Event notification |

# What is a Deadlock?

**Deadlock:** A set of threads where each is waiting for a resource held by another thread in the set. No thread can make progress.

```
// Thread 1
lock(A);
lock(B);  // blocks: T2 holds B

// Thread 2
lock(B);
lock(A);  // blocks: T1 holds A
```



# Four Necessary Conditions (Coffman)

**All four** must hold simultaneously for deadlock:

1. **Mutual Exclusion** — Resource held exclusively by one thread.
2. **Hold and Wait** — Thread holds resource(s) while waiting for more.
3. **No Preemption** — Resources cannot be forcibly taken.
4. **Circular Wait** — $T_1 \to T_2 \to \cdots \to T_n \to T_1$

## Key Insight

Prevent deadlock by breaking **any one** of these four conditions.

# Deadlock Prevention

Break one of the four conditions:

**Break Mutual Exclusion** Use shareable resources (read-only data, lock-free structures). Often impractical.

**Break Hold and Wait** Request **all** resources at once before starting. Low utilization.

**Break No Preemption** If a thread can't get a resource, release what it holds and retry. Works for some resources (CPU, memory).

**Break Circular Wait** **Lock ordering** — assign a global order to all locks; always acquire in ascending order.

## Most practical: Lock Ordering

Number your locks. Always acquire lock $i$ before lock $j$ if $i < j$. Circular wait becomes impossible.

# Lock Ordering Example

**Deadlock-prone:**

```
// Thread 1
lock(B);  // order: B, A
lock(A);

// Thread 2
lock(A);  // order: A, B
lock(B);
```

**Fixed (always A before B):**

```
// Thread 1
lock(A);  // order: A, B
lock(B);

// Thread 2
lock(A);  // order: A, B
lock(B);
```

No circular wait $\Rightarrow$ no deadlock.

# Deadlock Avoidance: Banker's Algorithm

**Idea:** Before granting a resource, check if the system stays in a *safe state*.

**Safe state:** There exists an ordering of threads such that each can finish with currently available + resources held by preceding threads.

**Banker's Algorithm (Dijkstra):**

1. Thread requests resources
2. OS *pretends* to grant them
3. Runs safety check: can all threads finish?
4. If safe $\rightarrow$ grant. If unsafe $\rightarrow$ thread waits.

**Limitation**

Requires knowing max resource needs in advance. Rarely used in practice but important conceptually.

# Deadlock Detection

**Allow deadlocks to occur, then detect and recover.**

**Resource Allocation Graph:**

- Nodes: threads and resources
- Edge $T_i \rightarrow R_j$: thread $i$ requests resource $j$
- Edge $R_j \rightarrow T_i$: resource $j$ assigned to thread $i$
- **Cycle = deadlock** (for single-instance resources)

**Recovery options:**

1. Terminate one or more deadlocked threads
2. Preempt resources from a thread (rollback)
3. Terminate all deadlocked threads (drastic)

# Deadlock Strategies Summary

| Strategy | Approach | Overhead |
|---|---|---|
| Prevention | Break a Coffman condition | Low–Med |
| Avoidance | Banker's algorithm | High |
| Detection | Find cycles in wait graph | Medium |
| Ignorance | "Ostrich algorithm" | None |

## In Practice

Most OSes (Linux, Windows) use **prevention** (lock ordering in kernel) + **ignorance** (let user-space deal with it). Detection is used in databases (transaction rollback).

# Key Takeaways

1. **Mutex** — binary lock with ownership; protects critical sections
2. **Semaphore** — counting primitive; controls concurrent access to $N$ resources
3. **Condition Variable** — wait for arbitrary conditions; always use with mutex + while loop
4. **Classic Problems** — Producer-Consumer, Readers-Writers, Dining Philosophers illustrate synchronization patterns
5. **Deadlock** — four Coffman conditions must all hold
6. **Prevention** — lock ordering is the most practical defense

## Next Week

Memory Management Fundamentals: address spaces, segmentation, fragmentation.