

# Lecture 9: Concurrency Control: Synchronization

(Producer-Consumer, Condition Variables, and Semaphores)

Xin Liu

Florida State University  
xl24j@fsu.edu

COP 4610 Operating Systems  
<https://xinliulab.github.io/cop4610.html>  
September 29, 2024

# Outline

- Producer-Consumer Problem
- Condition Variables
- Semaphores

## Spinlocks

- A spinlock is a simple lock where a thread constantly checks for lock availability.
- Imagine a single key to a critical section. The first thread to acquire the key can enter.
- Hardware instructions ensure atomic key exchange.

# Understanding Spinlocks Thoroughly

The single atomic instruction (xchg) ensures no race condition.

```
// 0 means unlocked, 1 means locked
```

```
int lock = 0;
```

```
void acquire_lock(int *lock) {
```

```
    while (*lock != 0) {}
```

```
    *lock = 1;
```

```
}
```

```
void release_lock(int *lock) {
```

```
    *lock = 0;
```

```
}
```

```
void *foo(void *arg) {
```

```
    acquire_lock(&lock);
```

```
    // Critical section: Do work here ...
```

```
    release_lock(&lock);
```

```
    return NULL;
```

```
}
```

```
// 0 means unlocked, 1 means locked
```

```
int lock = 0;
```

```
int xchg(int *addr, int newval) {
```

```
    int result;
```

```
    asm volatile (
```

```
        "lock xchg %0, %1"
```

```
        : "+m" (*addr), "=a" (result)
```

```
        : "1" (newval)
```

```
        : "cc"
```

```
    );
```

```
    return result;
```

```
}
```

```
void acquire_lock(int *lock) {
```

```
    while (xchg(lock, 1)) {}
```

```
}
```

```
void release_lock(int *lock) {
```

```
    xchg(lock, 0);
```

```
}
```

```
void *foo(void *arg) {
```

```
    acquire_lock(&lock);
```

```
    // Critical section: Do work here ...
```

```
    release_lock(&lock);
```

```
    return NULL;
```

```
}
```

# Rules for Acquiring a Lock

- **Grab first, verify later:** Don't bother checking if the lock is free. Just grab it and verify its status later.
- **Be fast:** Grab that lock as quickly as possible before anyone else does.

# Recap: Spinlocks

## Spinlocks

- A spinlock is a simple lock where a thread constantly checks for lock availability.
- Imagine a single key to a critical section. The first thread to acquire the key can enter.
- Hardware instructions ensure atomic key exchange.

## Performance Issue

- Spinlocks can cause inefficiency, especially if many threads compete for the same lock, leading to frequent context switches (Grab first, verify later).
- If a thread holding the lock is swapped out, all other threads continue busy-waiting, wasting CPU resources, because the CPU still considers them active (either in the Running or Ready to Run state).

# Recap: Mutexes and Futexes

## Mutexes

- The lock is managed by the OS kernel.
- When a thread attempts to acquire a mutex that is already locked, the OS puts the thread to sleep (blocked state) instead of busy-waiting.
- The kernel wakes up the thread when the lock becomes available, preventing it from wasting CPU time while waiting for the lock.

## Futexes

- A futex is a combination of spinlocks and mutexes.
- It starts with spinning and escalates to a kernel-based mutex when needed.
- This hybrid approach improves performance by reducing both busy-waiting in user space and context switches to the kernel.

# Example: Mutex with 3 Threads (Sleep and Wake-up)

- 1 **Thread X** acquires the lock first and enters the critical section.
  - 2 **Thread Y** and **Thread Z** attempt to acquire the lock but go into a sleep (blocked state) since the lock is already held by **X**.
  - 3 Once **X** finishes and releases the lock, the OS wakes up **Y**, typically following a first-come, first-served policy (FIFO) or priority-based scheduling.
  - 4 After Thread **Y** completes its critical section and releases the lock, the OS wakes up **Z**, which then acquires the lock.
- The waking mechanism is managed by the OS, which monitors the release of the lock and uses it as the signal to wake the next waiting thread.



# Building on Previous Experience

- We started with the thread library (`#include <pthread.h>`) and implemented simple threads and spinlocks.

```
pthread_t t1, t2;

pthread_create(&t1, NULL, foo, NULL);
pthread_create(&t2, NULL, foo, NULL);

pthread_join(t1, NULL);
pthread_join(t2, NULL);
```

```
int lock = 0; // Spinlock variable

// Atomic exchange function to swap *addr with newval.
// Not provided by pthread.h, so defined here.
int xchg(int *addr, int newval) {
    int result;
    asm volatile (
        "lock_xchg_%0,_%1"
        : "+m" (*addr), "=a" (result)
        : "1" (newval)
        : "cc"
    );
    return result;
}

void acquire_lock(int *lock) {while (xchg(lock, 1)) {}}

void release_lock(int *lock) {xchg(lock, 0);}

void *foo(void *arg) {
    acquire_lock(&lock);
    // Critical section: Do work here ...
    release_lock(&lock);
    return NULL;
}
```

- This provided your first steps into concurrent programming.

# Concurrency: What We Have Learned So Far

Date	Topic
9/4 (W)	L3: Introduction to Concurrency (OS State Machine, Process & Thread, Amdahl's Law)
9/9 (M)	L4: CPU Scheduling (OS Boot, Process Creation, Address Space, Scheduling)
9/11 (W)	L5: Independent & Cooperating Threads (Race Condition and Loss of Atomicity)
9/16 (M)	L6: Concurrency Control: Mutual Exclusion (Lock Failures, Peterson's Algorithm, and Spin Locks)
9/18 (W)	L7: UNIX Shell / Project 1 Hints
9/23 (M)	L8: Concurrency Control: Advanced Mutual Exclusion (Mutex & Futex Locks)

# The Essence of Collaborative Relationships

- Collaborative relationships are a combination of Competition Relationships and Dependency Relationships

# Competition Relationships

- Involves access and modification of shared resources within threads
- When threads are independent
  - The main concern is to avoid Competition Relationships
  - Use synchronization mechanisms like **Spinlocks** and **Mutex Locks**
  - Ensure only one thread accesses the shared resource at a time
  - Avoid data inconsistency and race conditions
- Focus on safe access within threads

# Dependency Relationships

- Involves execution order and causal relationships between threads
- When one thread must complete before another can execute
  - Use mechanisms like **Condition Variables** and **Semaphores**
  - Control the execution order of threads
  - Satisfy logical dependency requirements
- Focus on correct coordination between threads

## Core Question

- How do you coordinate multiple threads to handle tasks efficiently in real-world systems?

## Example: E-commerce Platform Order Processing System

- **Order Validation:** Check product inventory, user balance, and coupon validity.
- **Payment Processing:** Deduct from user accounts or process third-party payments.
- **Inventory Update:** Deduct product stock to prevent overselling.
- **Logistics Arrangement:** Generate shipping orders and arrange delivery.
- **Notify Users:** Send confirmation emails or SMS to users.

## Challenges and Solutions

- **Managing Shared Resources (Competition):**
  - Multiple threads updating inventory or user balances may cause race conditions.
  - **Solution:** Use Mutex locks to ensure only one thread modifies shared resources at a time.
  - Also, use transactions to roll back in case of failures, ensuring data consistency.
- **Managing Dependencies Between Threads (Dependency):**
  - Notification threads must wait until order processing is complete.
  - **Solution:** Use condition variables or semaphores to signal thread progress and control execution order.
  - Task queues can be used to arrange execution based on dependencies.

## Objective of this Lecture

- By the end of this lecture, you'll know how to utilize multiple CPUs for running parallel algorithms.



## Producer-Consumer Problem

- A fundamental synchronization problem that allows you to solve 99.9% of real-world concurrency issues.

## Dining Philosophers Problem

- Another classic problem that demonstrates how multiple entities share limited resources (like CPUs).

## Condition Variables

- A flexible synchronization primitive that allows threads to wait until a specific condition is met.

## Semaphores

- A more rigid mechanism used to control access to shared resources by multiple threads.

# Producer-Consumer Problem

## Producer "O" and Consumer "X"

### Producer:

- Produces an item ("O")
- Waits if storage is full
- Must be synchronized with the consumer

### Consumer:

- Consumes an item ("X")
- Waits if no item is available
- Synchronization ensures no consumption before production

- We need to ensure that the symbols ("O" and "X") are printed in a valid sequence:
- Example:
  - $n = 3$ , 000XXOXX000 (valid)
  - $n = 3$ , 0000XXXX, 00XXX (invalid)

# Why Producer-Consumer is Widely Representative

- Involves two types of threads: Producers (generate data) and Consumers (process data)
- Producers don't overflow the buffer and consumers don't try to consume data that's not yet available.

## Challenges:

- Synchronization and mutual exclusion
- Managing dependencies and inter-thread communication

# Initial Attempt

- Ensure the condition is met using mutex locks.
  - Link of Code: [Producer-Consumer Example Code](#)
- Stress Testing
  - Link of Code: [Stress Test Checker Code](#)
  - Command: `./a.out 2 | python3 pc_checker.py 2`
- **Bad News:**
  - After running the program for several hours, it actually failed!
  - The issue is difficult to reproduce and to fix.
  - Concurrent programming is highly challenging.
- **Good News:**
  - The problem occurred while it was in your hands.
  - Avoid taking shortcuts and always stick to the most reliable methods.

# **Condition Variables: A Universal Synchronization Method**

# The Essence of Synchronization

- The essence of synchronization is ensuring that multiple threads or processes reach a **known state** at the same time, so that they can proceed in coordination.

## Example:

- Imagine two people (threads) trying to meet for dinner (a task).
- One is playing a game (task A), and the other is fixing a bug (task B).
- They can't start dinner (synchronized task) until both have finished their tasks (**known state**).
- Even if one person finishes earlier, they must wait for the other.

## Core Concept

- The core of synchronization is waiting for all necessary conditions to be met before proceeding together.

# Synchronization Example

- From the very beginning when you started working with threads, you were already using synchronization.
- Can you find which part is synchronization?

```
pthread_t t1, t2;  
  
pthread_create(&t1, NULL, foo, NULL);  
pthread_create(&t2, NULL, foo, NULL);  
  
pthread_join(t1, NULL);  
pthread_join(t2, NULL);
```



# Synchronization Example (Cont.)

- From the very beginning when you started working with threads, you were already using synchronization.
- Can you find which part is synchronization?
  - `pthread_join` ensures that the main thread waits for the other threads to finish before continuing.
  - This is a form of synchronization because it guarantees that all threads reach a known state (completion) before the program proceeds.

```
pthread_t t1, t2;  
  
pthread_create(&t1, NULL, foo, NULL);  
pthread_create(&t2, NULL, foo, NULL);  
  
pthread_join(t1, NULL);  
pthread_join(t2, NULL);
```

# Problems with Initial Attempt

```
void *Tproduce(void *arg) {
    while (1) {
retry:
        pthread_mutex_lock(&lk);
        if (count == n) {
            pthread_mutex_unlock(&lk);
            goto retry;
        }
        count++;
        printf("O");
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

```
void *Tconsume(void *arg) {
    while (1) {
retry:
        pthread_mutex_lock(&lk);
        if (count == 0) {
            pthread_mutex_unlock(&lk);
            goto retry;
        }
        count--;
        printf("X");
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

# Problems with Initial Attempt (Cont.)

```
void *Tproduce(void *arg) {
    while (1) {
retry:
        pthread_mutex_lock(&lk);
        if (count == n) {
            pthread_mutex_unlock(&lk);
            goto retry;
        }
        count++;
        printf("O");
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

```
void *Tconsume(void *arg) {
    while (1) {
retry:
        pthread_mutex_lock(&lk);
        if (count == 0) {
            pthread_mutex_unlock(&lk);
            goto retry;
        }
        count--;
        printf("X");
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

- **Busy Waiting:** Both producer and consumer continuously retry when the buffer is full or empty. This leads to a waste of CPU resources.
- **Resource Contention:** Multiple threads constantly lock and unlock the same mutex without meaningful progress when conditions are not met, causing unnecessary contention.
- **High CPU Utilization:** The goto retry causes the threads to remain in a tight loop, consuming CPU cycles even when they should be waiting.

## **"Haste makes waste."**

*Constant spinning and busy waiting lead to errors. Slowing down with condition variables reduces mistakes.*

- Link of Code: [Condition Variables Example Code](#)

# Tip 1: pthread\_cond\_wait

```
void *Tproduce(void *arg) {
    while (1) {
        pthread_mutex_lock(&lk);
        while (count == n) {
            pthread_cond_wait(&not_full, &lk);
        }
        count++;
        printf("O");
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

```
void *Tconsume(void *arg) {
    while (1) {
        pthread_mutex_lock(&lk);
        while (count == 0) {
            pthread_cond_wait(&not_empty, &lk);
        }
        count--;
        printf("X");
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

- pthread\_cond\_wait: A thread goes to sleep and releases the mutex while waiting for a condition (e.g., buffer not empty/full).
- **Important:** pthread\_cond\_wait must be used with a mutex.
  - The thread must first acquire the mutex lock before calling pthread\_cond\_wait.
  - pthread\_cond\_wait only handles waiting for a condition to be met, it does not handle acquiring the lock.

## Tip 2: pthread\_cond\_signal

```
void *Tproduce(void *arg) {
    while (1) {
        pthread_mutex_lock(&lk);
        while (count == n) {
            pthread_cond_wait(&not_full, &lk);
        }
        count++;
        printf("O");
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

```
void *Tconsume(void *arg) {
    while (1) {
        pthread_mutex_lock(&lk);
        while (count == 0) {
            pthread_cond_wait(&not_empty, &lk);
        }
        count--;
        printf("X");
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

- pthread\_cond\_signal: Wake up one waiting thread when the condition is met (e.g., an item is produced or consumed).
  - Which thread is woken up?
    - If multiple threads are waiting, the OS decides which thread to wake up based on a scheduling policy, usually first-come, first-served (FIFO) or priority-based.

## Tip 3: You can also use pthread\_cond\_broadcast

```
void *Tproduce(void *arg) {
    while (1) {
        pthread_mutex_lock(&lk);
        while (count == n) {
            pthread_cond_wait(&not_full, &lk);
        }
        count++;
        printf("O");
        pthread_cond_broadcast(&not_empty);
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

```
void *Tconsume(void *arg) {
    while (1) {
        pthread_mutex_lock(&lk);
        while (count == 0) {
            pthread_cond_wait(&not_empty, &lk);
        }
        count--;
        printf("X");
        pthread_cond_broadcast(&not_full);
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

- pthread\_cond\_broadcast: Wake up all waiting threads when the condition is met.
  - When to use pthread\_cond\_broadcast?
    - Use pthread\_cond\_broadcast when a global state changes that affects all threads.

# The Most Important Tip: Two Condition Variables!

```
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;  
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
```

- Avoid waking the same type of thread:
  - Producers should not wake other producers, and consumers should not wake other consumers.
  - Producer thread:
    - Waits on `not_full` when the buffer is full.
    - Signals `not_empty` after producing an item, allowing consumers to wake up and consume.
  - Consumer thread:
    - Waits on `not_empty` when the buffer is empty.
    - Signals `not_full` after consuming an item, allowing producers to wake up and produce.
- Link of Code: [\*\*Single Condition Variable Example Code\*\*](#)



# Deadlock with Single Condition Variable Example

```
pthread_cond_t buffer_change = PTHREAD_COND_INITIALIZER;
```

- Scenario:
  - Buffer size (  $n = 1$  )
  - 2 producer threads (P1, P2) and 2 consumer threads (C1, C2)
  - The buffer is empty and C1 and C2 are sleeping
  - P2 is also sleeping due to the buffer being full previously.
- Process:
  - P1 produces an item, filling the buffer ( count = 1 ), then signals 'buffer\_change' (P1 is ready to run and not sleeping)
  - The signal wakes up P2
  - P2 is woken up, but finds the buffer is full, so P2 goes back to sleep without sending any signal
  - P1 is scheduled by the OS, but P1 also finds the buffer is full and goes to sleep without sending any signal
  - The OS may now try to schedule C1 or C2, but they are still sleeping, waiting for the signal that hasn't been sent
- Result:
  - All threads are now in a sleeping state, resulting in deadlock

# Cause of Single Condition Variable Deadlock

- All threads rely on a signal to wake up, rather than automatically waking when the condition becomes true.
- A single condition variable may wake up the same type of thread repeatedly.
- No further signals can be sent, leading to deadlock.
- Role of the Operating System:
  - Manages thread scheduling and CPU time allocation
  - Does not manage thread synchronization or signal passing
  - Cannot wake threads
- Thread Communication:
  - Synchronization happens through condition variables (signals) and mutexes
  - Signals must be explicitly sent and received between threads
  - Proper signal passing is critical for correct thread coordination

# Why Two Condition Variables Prevent Deadlock

- A producer's 'not\_empty' signal only wakes consumers.
- A consumer's 'not\_full' signal only wakes producers.
- At least one thread type can always proceed and change the buffer state
- Eliminates the possibility of all threads waiting at the same time

# Why Two Condition Variables Prevent Deadlock

- A producer's 'not\_empty' signal only wakes consumers.
- A consumer's 'not\_full' signal only wakes producers.
- At least one thread type can always proceed and change the buffer state
- Eliminates the possibility of all threads waiting at the same time

# Limitations of Condition Variables

- Imagine a buffer with 5 slots, initially empty / full.
- If 5 producer / consumer threads want to produce / consume 'O', a condition variable only allows one thread to produce / consume at a time.
- But what if we want multiple threads to produce / consume 'O' concurrently?

# Semaphores

- **Semaphore** is a synchronization mechanism used to control access to shared resources in concurrent systems.
- It acts as an integer counter that tracks the availability of a limited number of resources.
- It can allow multiple threads to enter the critical section simultaneously.
  - However, you must ensure that there are no race conditions when multiple threads are in the critical section. If there are no such issues, semaphores can be used effectively.

# Semaphore Operations

- Semaphores were first introduced by **Edsger W. Dijkstra** in the 1960s.
- Semaphores operate similarly to **condition variables**, allowing threads to wait and be signaled based on certain conditions.

## Semaphores have two primary operations:

- **P operation** (from Dutch *proberen*, meaning "to try"):
  - Decreases the semaphore's value by 1.
  - If the value becomes negative, the thread performing the P operation is blocked until the semaphore's value becomes positive.
- **V operation** (from Dutch *verhogen*, meaning "to increment"):
  - Increases the semaphore's value by 1.
  - If there are any blocked threads, the V operation wakes up one of them.

# Code Comparison

```
// Condition Variables
void *Tproduce(void *arg) {
    while (1) {
        pthread_mutex_lock(&lk);
        while (count == n) {
            pthread_cond_wait(&not_full, &lk);
        }
        count++;
        printf("O");
        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}

void *Tconsume(void *arg) {
    while (1) {
        pthread_mutex_lock(&lk);
        while (count == 0) {
            pthread_cond_wait(&not_empty, &lk);
        }
        count--;
        printf("X");
        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&lk);
    }
    return NULL;
}
```

```
// Semaphores
void *Tproduce(void *arg) {
    while (1) {
        P(&empty_sem);

        pthread_mutex_lock(&mutex);
        printf("O");
        pthread_mutex_unlock(&mutex);

        V(&full_sem);
        return NULL;
    }
}

void *Tconsume(void *arg) {
    while (1) {
        P(&full_sem);

        pthread_mutex_lock(&mutex);
        printf("X");
        pthread_mutex_unlock(&mutex);

        V(&empty_sem);
    }
    return NULL;
}
```

- Link of Code: [Semaphores Example Code](#)



# Semaphores vs Condition Variables: Key Differences

- **Resource Management:**

- Semaphores have a built-in counter to manage resource availability.
- Condition variables do not track resource availability. The programmer must manage resource state manually.

- **Wait/Wake Mechanism:**

- Semaphores use the **P (wait)** and **V (signal)** operations to automatically handle the blocking and unblocking of threads.
- Condition variables use **pthread\_cond\_wait()** to put a thread to sleep and **pthread\_cond\_signal()** or **pthread\_cond\_broadcast()** to wake up waiting threads.

- **Mutex Usage:**

- Semaphores can be used with or without a mutex, allowing multiple threads to access the critical section simultaneously based on the semaphore's value.
- Condition variables must be used with a mutex, typically allowing only one thread in the critical section at a time, even if multiple threads are woken up.

# Semaphores without Mutex

```
void *Tproduce(void *arg) {  
    while (1) {  
        P(&empty_sem);  
        printf("O");  
        V(&full_sem);  
    }  
    return NULL;  
}
```

```
void *Tconsume(void *arg) {  
    while (1) {  
        P(&full_sem);  
        printf("X");  
        V(&empty_sem);  
    }  
    return NULL;  
}
```

- Link of Code: [Semaphores without Mutex Example Code](#)

## Considerations for Semaphore Usage

- If you plan to implement more complex buffer operations (e.g., actually storing data instead of just printing characters), you will need to use a mutex to avoid race conditions.
- While semaphores may seem convenient, they become less effective as more rules are added, making them harder to manage.
- It's often better to use **condition variables** for complex synchronization needs.

# Takeaways

- Most of the synchronization problems you will face are just variations of the **Producer-Consumer problem**.
- Mastering **condition variables** is enough to handle most real-world scenarios.
- The rest is just icing on the cake.