

LAB NO: 7

# Classical Problems of Synchronization

Operating Systems • Thread Safety • Semaphores

---

Mutex Locks • Semaphores • Producer-Consumer • Readers-Writers • Dining Philosophers



# Learning Objectives

01

## Synchronization with Semaphores

Learn how to coordinate multiple processes using POSIX semaphores to avoid race conditions and ensure safe shared memory access.

02

## Inter-Process Communication

Understand how named pipes enable communication between two or more processes in a structured and synchronized manner.

03

## Classical Sync Problems

Explore foundational concurrency problems: Producer-Consumer, Readers-Writers, Bounded Buffer, and Dining Philosophers.

# What is a Data Race?

A data race occurs when multiple threads use the same data item and one or more of those threads are updating it — leading to unpredictable results.

## Scenario: Two Threads

```
*a = 10 initially  
Thread 1 reads: 10  
Thread 2 reads: 10  
Thread 1 writes: 14  
Thread 2 writes: 14  
Expected: 18 ✘ Got: 14
```

## Root Causes

- ▶ Shared mutable state
- ▶ No mutual exclusion
- ▶ Context switching mid-operation
- ▶ Non-atomic read-modify-write

# Race Condition in C

```
// race.c
#include <pthread.h>

int counter = 0;

void * func(void * params) {
    counter++; // Non-atomic: READ → MODIFY → WRITE
}

void main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, 0, func, 0);
    pthread_create(&thread2, 0, func, 0);
    pthread_join(thread1, 0);
    pthread_join(thread2, 0);
    // counter may be 1 instead of 2!
}
```

## ⚠ Why It Breaks

### Non-atomic op

counter++ compiles to 3 assembly instructions

### Context switch

OS can interrupt between read & write

### Lost update

One thread's write overwrites another's

### Detection

Use Helgrind:  
valgrind --tool=helgrind ./a.out

# Detecting Data Races with Helgrind

Helgrind is part of the Valgrind suite and is specifically designed to detect synchronization errors in C/C++ programs.

## Step 1

Compile with debug symbols

```
$ gcc -g race.c -lpthread -o race
```

## Step 2

Run under Helgrind

```
$ valgrind --tool=helgrind ./race
```



Helgrind reports: thread IDs, conflicting access locations, and the exact lines causing the race condition.

# Mutex Locks

A mutex lock is a mechanism that can be acquired by only one thread at a time. Other threads must wait until it is released.

## `pthread_mutex_init()`

Initialize mutex (dynamic or static via PTHREAD\_MUTEX\_INITIALIZER)

## `pthread_mutex_lock()`

Acquire lock — blocks if already held by another thread

## `pthread_mutex_unlock()`

Release the lock so waiting threads can proceed

## `pthread_mutex_destroy()`

Free resources when mutex is no longer needed

# Mutex Lock — Counter Example

```
pthread_mutex_t mutex;
volatile int counter = 0;

void * count(void * param) {
    for (int i = 0; i < 100; i++) {
        pthread_mutex_lock(&mutex);      // Acquire lock
        counter++;
        printf("Count = %i\n", counter);
        pthread_mutex_unlock(&mutex);   // Release lock
    }
}

int main() {
    pthread_mutex_init(&mutex, 0);
    pthread_create(&thread1, 0, count, 0);
    pthread_create(&thread2, 0, count, 0);
    pthread_join(thread1, 0);
    pthread_join(thread2, 0);
    pthread_mutex_destroy(&mutex);
}
```

## Key Points

- ✓ Lock wraps critical section only
- ✓ Unlock must always be called
- ✓ One thread in section at a time
- ✓ Counter will reliably reach 200
- ✓ Init before create, destroy after join

# Semaphores

A semaphore is a counting and signaling mechanism used to allow threads controlled access to a specified number of shared resources.

## Mutex

- Binary (0 or 1)
- Ownership concept
- Only locking thread can unlock
- Used for mutual exclusion
- One resource at a time

## Semaphore

- Counting (0 to N)
- No ownership
- Any thread can signal
- Used for signaling + counting
- Multiple resources

# POSIX Semaphore API

## `sem_init(sem, pshared, value)`

Initialize semaphore. pshared=0 (private to process), value=initial count.

## `sem_wait(sem)`

Decrement semaphore. If value is 0, blocks until nonzero, then decrements.

## `sem_post(sem)`

Increment the semaphore. Unblocks a waiting thread if any are queued.

## `sem_getvalue(sem, *val)`

Write current semaphore value into the integer variable pointed to by val.

## `sem_destroy(sem)`

Free resources. Call after all threads have finished using the semaphore.

# Semaphore for Thread Ordering

```
sem_t semaphore;

void *func1(void * param) {
    printf("Thread 1\n");
    sem_post(&semaphore); // Signal: done
}

void *func2(void * param) {
    sem_wait(&semaphore); // Wait for Thread 1
    printf("Thread 2\n");
}

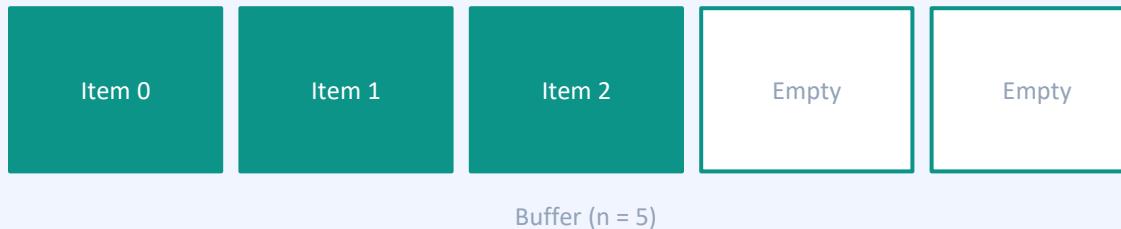
int main() {
    sem_init(&semaphore, 0, 1);
    pthread_create(&threads[0], 0, func1, 0);
    pthread_create(&threads[1], 0, func2, 0);
    pthread_join(threads[0], 0);
    pthread_join(threads[1], 0);
    sem_destroy(&semaphore);
}
```

## Execution Flow

- 1 Thread 1 starts
- 2 Prints 'Thread 1'
- 3 Posts (signals)
- 4 Thread 2 unblocks
- 5 Prints 'Thread 2'
- 6 Guaranteed order!

# The Bounded-Buffer Problem

A pool of  $n$  buffers, each holding one item. Synchronization ensures producers don't overfill and consumers don't read empty buffers.



## Semaphores Used:

**mutex**

1

Mutual exclusion for buffer access

**empty**

n (5)

Counts empty buffer slots

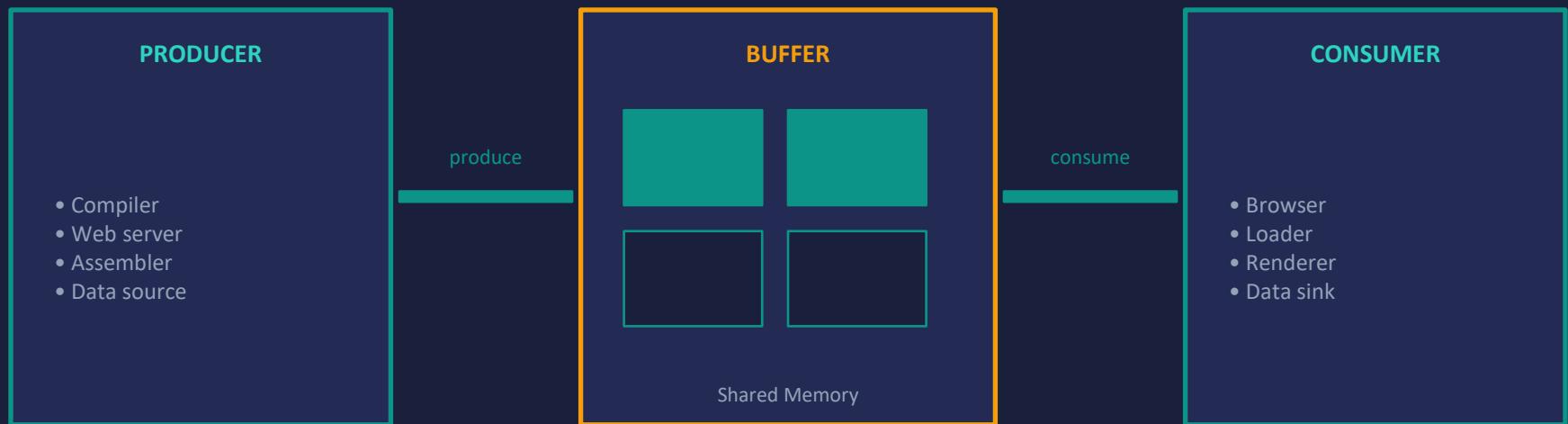
**full**

0

Counts filled buffer slots

# The Producer-Consumer Problem

A producer generates data; a consumer processes it. They share a buffer and must be synchronized to avoid overproduction or premature consumption.



# Producer Function

```
int buf[5], f, r;
sem_t mutex, full, empty;

void *produce(void *arg) {
    int i;
    for (i = 0; i < 10; i++) {
        sem_wait(&empty);    // Wait for empty slot
        sem_wait(&mutex);   // Lock buffer

        printf("produced item: %d\n", i);
        buf[(++r) % 5] = i; // Add to circular buf
        sleep(1);

        sem_post(&mutex);   // Unlock buffer
        sem_post(&full);    // Signal item available
    }
}
```

## Semaphore Logic

### sem\_wait(empty)

Decrement empty count. Block if buffer full.

### sem\_wait(mutex)

Lock buffer for exclusive write access.

### write item

Store item in circular buffer at position r%5.

### sem\_post(mutex)

Release the buffer lock.

### sem\_post(full)

Increment full count, wake consumer.

# Consumer Function

```
void *consume(void *arg) {
    int item, i;
    for (i = 0; i < 10; i++) {
        sem_wait(&full);      // Wait for item
        sem_wait(&mutex);     // Lock buffer

        item = buf[(++f) % 5]; // Read from circular buf
        printf("consumed item: %d\n", item);
        sleep(1);

        sem_post(&mutex);    // Unlock buffer
        sem_post(&empty);    // Signal slot freed
    }
}

int main() {
    sem_init(&mutex, 0, 1);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, 5);
    pthread_create(&tid1, NULL, produce, NULL);
    pthread_create(&tid2, NULL, consume, NULL);
}
```

## Init Values

**mutex → 1**

Binary: only 1 in buffer

**full → 0**

No items yet

**empty → 5**

All 5 slots free

Note: sem\_init with value  
• 0 = process-private  
• 1 = process-shared

# The Readers-Writers Problem

A shared database is accessed by both readers (read-only) and writers (read+write). The challenge is safe coordination.

Reader + Reader	ALLOWED
Multiple readers can access simultaneously with no conflict.	

Writer + Writer	BLOCKED
Two writers simultaneously causes data corruption — must be exclusive.	

Reader + Writer	BLOCKED
Reading while writing may see inconsistent data — must wait.	

Solo Writer	EXCLUSIVE
Writer needs exclusive access to the entire database.	

# Data Structures & Semaphore Roles

**mutex**

semaphore

= 1

Protects readcount variable updates. Ensures only one reader modifies readcount at a time.

**wrt**

semaphore

= 1

Shared by readers and writers. Controls write exclusivity and first/last reader access.

**readcount**

int

= 0

Tracks how many readers are currently in the critical section.

# Writer Process Structure

```
do {  
    wait(wrt);      // Acquire write lock  
  
    // ... writing is performed ...  
    // Exclusive access to database  
  
    signal(wrt);    // Release write lock  
  
    // ... non-critical section ...  
  
} while (TRUE);
```

## Writer Rules

- ▶ Must wait if any reader or writer is active
- ▶ Gets exclusive access (no readers, no writers)
- ▶ wrt semaphore ensures only 1 writer at a time
- ▶ Signal wrt when done to allow others
- ▶ Simple structure — only one semaphore needed

wrt = 1 initially → first writer acquires it. All others (readers + writers) must wait until signal(wrt).

# Reader Process Structure

```
do {  
    wait(mutex);  
    readcount++;  
    if (readcount == 1)  
        wait(wrt);      // First reader locks writers  
    signal(mutex);  
  
    // ... reading is performed ...  
  
    wait(mutex);  
    readcount--;  
    if (readcount == 0)  
        signal(wrt);   // Last reader unlocks writers  
    signal(mutex);  
  
} while (TRUE);
```

## Reader Logic

**readcount == 1**

First reader → block writers

**readcount > 1**

Subsequent readers enter freely

**readcount == 0**

Last reader → unblock writers

**mutex**

Protects readcount updates

# The Dining Philosophers Problem

## 5 Philosophers

Sit at a round table, alternating between thinking and eating.

## 5 Chopsticks

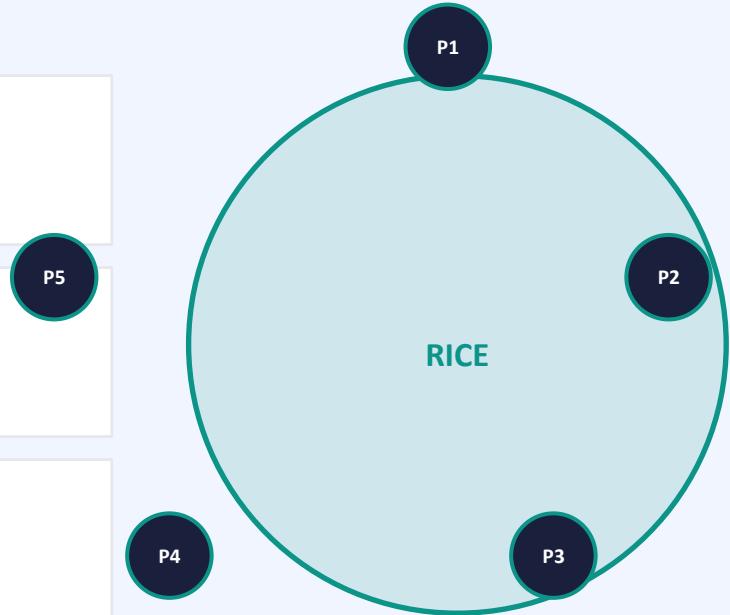
Each philosopher has one chopstick. Needs 2 to eat — must share with neighbor.

## Shared Resource

Chopsticks are shared resources. Grabbing both prevents eating by neighbors.

## Deadlock Risk

If all pick up left chopstick simultaneously → deadlock. No one can eat.



# Deadlock Scenario

Deadlock occurs when each philosopher holds one chopstick and waits forever for the second one that their neighbor holds.

1

## All philosophers get hungry

All 5 decide to eat at the same time.

2

## All pick up left chopstick

Each philosopher grabs the chopstick on their left.

3

## All wait for right chopstick

Each needs the right chopstick, which their right neighbor holds.

4

## Circular wait → Deadlock

No one can proceed. System is stuck indefinitely.

# Solutions to Avoid Deadlock

**1**

## Allow max N-1 philosophers to sit

Only 4 out of 5 allowed to sit at once. Guarantees at least one can eat.

**2**

## Asymmetric Chopstick Rule

Odd-numbered philosophers pick up left first; even pick up right first. Breaks circular wait.

**3**

## Both-or-Nothing Pickup

A philosopher only picks up chopsticks if both are available simultaneously (atomic action).

**4**

## Resource Hierarchy

Number resources globally. Always acquire lower-numbered resource first to prevent circular dependency.

# Dining Philosophers — Semaphore Solution

```
#define N 5
sem_t chopstick[N];

void philosopher(int i) {
    while(TRUE) {
        think();

        // Pick up chopsticks
        sem_wait(&chopstick[i]);          // Left
        sem_wait(&chopstick[(i+1)%N]); // Right

        eat();

        // Put down chopsticks
        sem_post(&chopstick[i]);
        sem_post(&chopstick[(i+1)%N]);
    }
}

int main() {
    for (int i = 0; i < N; i++)
        sem_init(&chopstick[i], 0, 1);
    // Create 5 philosopher threads
}
```

## Design

### chopstick[i]

Semaphore per chopstick, init=1

### sem\_wait left

Acquire left chopstick

### sem\_wait right

Acquire right (or block)

### sem\_post ×2

Release both after eating

## Caution

May still deadlock if all grab left — add asymmetric rule or limit philosophers

# Modify Producer-Consumer: Bounded Production Limit

Task: Modify the Producer-Consumer program so a producer can produce at most 10 items more than the consumer has consumed.

1

## Use `sem_getvalue()`

Read current value of the 'full' semaphore to know how many items are pending consumption.

2

## Add check before produce

Before producing, check if (`full_count >= 10`). If so, block the producer by calling `sem_wait` on a custom 'limit' semaphore.

3

## Consumer signals producer

After consuming, if pending count was at limit (`==10`), call `sem_post(wrt)` to wake the producer.

4

## Test thoroughly

Verify that the buffer count never exceeds 10 items ahead. Log produced/consumed counts to confirm.

# Code Hint — Bounded Production

```
// In consumer function – after consuming item:  
sem_post(&mutex);  
sem_post(&empty);  
  
// Check if producer was blocked at limit  
int val;  
sem_getvalue(&full, &val);  
if (val < 10)  
    sem_post(&wrt); // Wake up producer  
  
// In producer function – before producing:  
int pending;  
sem_getvalue(&full, &pending);  
if (pending >= 10) {  
    sem_wait(&wrt); // Block until consumer catches up  
}  
sem_wait(&empty);  
sem_wait(&mutex);  
// ... produce item ...
```

# First Readers-Writers Problem in C/C++ using Semaphores



## Declare semaphores and readcount

sem\_t mutex, wrt; int readcount = 0; — Initialize both semaphores to 1.



## Writer thread function

Call wait(wrt) before writing. Call signal(wrt) after. Ensure exclusive write access.



## Reader thread function

Increment readcount with mutex protection. First reader waits on wrt; last signals wrt.



## Main function setup

Initialize semaphores, create N reader threads and M writer threads, then join all.



## Handle starvation (optional)

First readers-writers problem can starve writers. Add a 'turn' semaphore if needed.

# Dining Philosophers in C/C++

Implement the complete Dining Philosophers solution using POSIX semaphores, avoiding deadlock.



## Setup

Declare chopstick[5] semaphores. Initialize each to 1 (one chopstick per seat).



## Acquire Chopsticks

sem\_wait left chopstick, then right chopstick. Use asymmetric rule for philosopher 0 to break deadlock.



## Release Chopsticks

sem\_post both chopsticks. Allow neighbors to proceed.



## Think Phase

philosopher() function calls think() (sleep or print) before trying to eat.



## Eat Phase

Call eat() (sleep or print) while holding both chopsticks.



## Main Function

Create 5 philosopher threads passing philosopher index. Join all threads. Verify no deadlock occurs.

# Classic Synchronization Problems — Summary

Problem	Parties	Key Semaphores	Challenge	Solution
Producer-Consumer	2 threads	mutex, full, empty	Buffer overflow / underflow	Bounded buffer with 3 semaphores
Readers-Writers	N threads	mutex, wrt, readcount	Writer starvation	First/second readers-writers variants
Dining Philosophers	5 threads	chopstick[N]	Deadlock, starvation	Asymmetric pickup / limit seats
Bounded Buffer	2 threads	mutex, empty (n)	N-slot shared pool	Same as producer-consumer

# Key Concepts Recap

## Data Race

Two or more threads access shared data concurrently, at least one writes, without synchronization.

## Critical Section

Code segment that accesses shared resources. Must be executed by only one thread at a time.

## Mutex

Binary lock mechanism. Only one thread can hold it at a time. Ensures mutual exclusion.

## Semaphore

Counting mechanism supporting wait/post. Can manage access to N instances of a resource.

## Deadlock

All threads blocked waiting for resources held by each other. Circular wait → no progress.

## Starvation

A thread is perpetually denied access to resources it needs, even though the system is not deadlocked.

# Common Mistakes & Best Practices

## Common Mistake

Forgetting to unlock mutex

Using sem\_wait without sem\_post

Destroying semaphore while in use

Not initializing semaphores

## Best Practice

Always call `pthread_mutex_unlock()` in the same code path as `lock()`

Every wait must have a corresponding post — otherwise threads block forever

`sem_destroy()` only after all threads have finished using the semaphore

Always call `sem_init()` or `pthread_mutex_init()` before creating threads

# Lab 7 Complete!

Classical Problems of Synchronization

5

Sync Concepts

3

Lab Problems

7

POSIX APIs

$\infty$

Bugs Prevented

## Topics Covered

Data Races • Mutex Locks • Semaphores • Producer-Consumer • Readers-Writers • Dining Philosophers • Deadlock Prevention

Operating Systems Laboratory — Lab No. 7