

# **Operating Systems – COC 3071L**

**SE 5th A – Fall 2025**

---

## **Lab 9: Synchronization**

### **1. Introduction to Synchronization**

#### **1.1 What is Synchronization?**

**Synchronization** ensures that multiple threads or processes **do not interfere** with each other when accessing **shared resources** (like variables, files, or memory).

It prevents **race conditions**, where two or more threads access the same data concurrently and produce **unpredictable results**.

**Real-world analogy:**

Imagine two people editing the same document at the same time — one deletes a paragraph while the other changes it. The final result depends on *who finishes last!* Synchronization is like setting “edit permissions” so only one person can modify the file at a time.

---

#### **1.2 Critical Section**

A **critical section** is a block of code that:

- Accesses or modifies **shared resources**
- Must be executed by **only one thread at a time**

To protect a critical section, we use **synchronization mechanisms** like:

- **Peterson’s Algorithm (software-based)**
  - **Mutex (Mutual Exclusion Lock)**
  - **Semaphores**
  - **Condition Variables**
-

## 2. Mutex Locks

### 2.1 What is a Mutex?

A **mutex (mutual exclusion)** is a synchronization primitive that ensures only one thread can enter a **critical section** at a time.

If another thread tries to lock an already locked mutex, it must **wait** until it's released.

**Functions used:**

Function	Purpose
<code>pthread_mutex_init()</code>	Initialize a mutex
<code>pthread_mutex_lock()</code>	Acquire the lock
<code>pthread_mutex_unlock()</code>	Release the lock
<code>pthread_mutex_destroy()</code>	Free resources used by mutex

---

### Program 1: Fixing Race Condition using Mutex

```
#include <stdio.h>
#include <pthread.h>

int counter = 0; // Shared resource
pthread_mutex_t lock; // Declare mutex

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock); // Lock before accessing shared
variable
        counter++;
        pthread_mutex_unlock(&lock); // Unlock after done
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL); // Initialize mutex

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

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

```

    printf("Expected counter value: 200000\n");
    printf("Actual counter value: %d\n", counter);

    pthread_mutex_destroy(&lock); // Clean up
    return 0;
}

```

## Observation:

- Output is now **always 200000**.
  - Mutex ensures **one thread at a time** accesses the shared variable.
- 

## 3. Peterson's Algorithm

### 3.1 Concept

Before hardware-based synchronization tools (like mutexes and semaphores), synchronization was handled purely in **software**.

**Peterson's Algorithm** is a classic software-based solution for achieving **mutual exclusion** between two processes or threads.

It ensures:

1. **Mutual Exclusion** – Only one thread is in the critical section.
  2. **Progress** – No thread is unnecessarily delayed.
  3. **Bounded Waiting** – Every thread gets a fair chance eventually.
- 

### 3.2 How Peterson's Algorithm Works

It uses two shared variables:

- `flag[2]` : Indicates a thread's intent to enter the critical section.
  - `turn` : Indicates whose turn it is to enter.
- 

## Program 2: Peterson's Algorithm Simulation

```

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

```

```

int turn;
int flag[2] = {0, 0};
int counter = 0; // Shared resource

void* process(void* arg) {
    int id = *(int*)arg; // Thread ID: 0 or 1
    int other = 1 - id;

    for (int i = 0; i < 5; i++) {
        // Entry section
        flag[id] = 1;
        turn = other;
        while (flag[other] && turn == other);

        // Critical section
        counter++;
        printf("Thread %d in critical section | Counter = %d\n", id,
        counter);
        sleep(1);

        // Exit section
        flag[id] = 0;
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t t1, t2;
    int id1 = 0, id2 = 1;

    pthread_create(&t1, NULL, process, &id1);
    pthread_create(&t2, NULL, process, &id2);

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

    printf("Final Counter Value: %d\n", counter);
    return 0;
}

```

## 4. Introduction to Semaphores

### 4.1 What is a Semaphore?

A **semaphore** is a synchronization primitive that maintains an internal counter. It can be:

- **Decrement** (wait operation)
- **Increment** (signal operation)

### Types of Semaphores:

Type	Initial Value	Purpose
<b>Binary Semaphore</b>	0 or 1	Similar to a mutex; mutual exclusion
<b>Counting Semaphore</b>	$n > 1$	Access to multiple units of a shared resource

## 4.2 Semaphore Operations

### Wait (P operation):

- If counter > 0, decrement and proceed
- If counter = 0, block

### Signal (V operation):

- Increment counter
- Wake up a waiting thread

## 4.3 Binary Semaphore Example

This example demonstrates mutual exclusion using a **binary semaphore**. Initial value = 1, so only one thread can enter the critical section. A binary Semaphore works similarly to Mutex lock.

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex; // Binary semaphore
int counter = 0;

void* thread_function(void* arg) {
    int id = *(int*)arg;

    for (int i = 0; i < 5; i++) {
        printf("Thread %d: Waiting...\n", id);
        sem_wait(&mutex); // Acquire
    }
}
```

```

        // Critical section
        counter++;
        printf("Thread %d: In critical section | Counter = %d\n", id,
counter);
        sleep(1);

        sem_post(&mutex); // Release
        sleep(1);
    }

    return NULL;
}

int main() {
    sem_init(&mutex, 0, 1); // Binary semaphore initialized to 1

    pthread_t t1, t2;
    int id1 = 1, id2 = 2;

    pthread_create(&t1, NULL, thread_function, &id1);
    pthread_create(&t2, NULL, thread_function, &id2);

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

    printf("Final Counter Value: %d\n", counter);

    sem_destroy(&mutex);
    return 0;
}

```

## 4.4 Counting Semaphore Example

A **counting semaphore** with initial value = 3 allows up to 3 threads to access a resource simultaneously.

```

#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t resource_semaphore;

void* thread_function(void* arg) {
    int thread_id = *(int*)arg;

```

```

    printf("Thread %d: Waiting for resource...\n", thread_id);
    sem_wait(&resource_semaphore); // Wait: decrement counter

    printf("Thread %d: Acquired resource!\n", thread_id);
    sleep(2); // Use resource

    printf("Thread %d: Releasing resource...\n", thread_id);
    sem_post(&resource_semaphore); // Signal: increment counter

    return NULL;
}

int main() {
    sem_init(&resource_semaphore, 0, 3); // Allow 3 concurrent threads

    pthread_t threads[5];
    int ids[5];

    for (int i = 0; i < 5; i++) {
        ids[i] = i;
        pthread_create(&threads[i], NULL, thread_function, &ids[i]);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&resource_semaphore);
    return 0;
}

```

**Compile and run:**

```

gcc counting_semaphore.c -o counting_semaphore -lpthread
./counting_semaphore

```

**Explanation:**

- Semaphore initialized with value **3**.
- Threads 0, 1, 2 acquire resources immediately.
- Threads 3, 4 block until one of the first three finishes.
- When a thread releases, the next waiting thread acquires the resource.

## 4.5 Semaphore Use Cases

### 1. Mutual Exclusion (Binary Semaphore):

```
sem_t mutex = 1; // Binary semaphore  
  
sem_wait(&mutex);  
// Critical section  
sem_post(&mutex);
```

## 2. Resource Pool (Counting Semaphore):

```
sem_t pool = 5; // 5 available resources  
  
sem_wait(&pool); // Acquire a resource  
// Use resource  
sem_post(&pool); // Release resource
```

## 3. Producer-Consumer Synchronization:

- Empty buffer semaphore (N slots available)
  - Full buffer semaphore (0 items available)
  - Mutex semaphore (protect buffer access)
- 

## 5. Summary

1. **Race conditions** occur when shared data is accessed without synchronization.
  2. **Mutexes** enforce one-thread-at-a-time access.
  3. **Peterson's Algorithm** provides software-only mutual exclusion.
  4. **Semaphores** offer generic synchronization — binary semaphores for mutual exclusion, counting semaphores for multiple resources.
  5. Proper use of **wait()** and **signal()** prevents conflicts and ensures safe concurrency.
-