Lab 07
Process Synchronization
Chapters 6 & 7

Section 1: Summary – Self-study before entering the lab

This summary is based on the Week 6 lecture materials, which cover selected topics from Chapter 6 (Synchronization Tools) and Chapter 7 (Synchronization Examples) of the textbook

1. Core Concepts and Requirements

- Processes/threads execute concurrently and can be interrupted at any time, leading to data inconsistency when accessing shared data.
    - Race Condition: Occurs when multiple threads access shared data, and the outcome depends on the unpredictable order of execution (interleaving of low-level instructions).
- Critical-Section Problem: The goal is to design a protocol for processes to share data by ensuring that the critical section (where shared data is accessed) is executed with mutual exclusion.
- Requirements for a Solution:
    1. Mutual Exclusion: Only one process/thread may be in the critical section at any time.
    2. Progress: Selection of a process to enter the critical section cannot be postponed indefinitely.
    3. Bounded Waiting: A limit exists on the number of times other processes can enter the critical section after one has requested entry.

2. Synchronization Tools

- Hardware Solutions: Necessary because instruction reordering can break software solutions (like Peterson's).
    - Memory Barriers: Instructions that force memory changes to be visible to all processors.
    - Atomic Operations (CAS/Test-and-Set): Perform a read, modify, and write on a memory location as a single, uninterruptible instruction.
    - Atomic Variables: Provide atomic updates for basic data types, such as integer counters.

- Software Primitives:
    - Mutex Locks: The simplest tool; threads use acquire() before and release() after the critical section.
        - Spinlocks (Busy Waiting): A thread continuously loops, consuming CPU time while waiting for the lock to be released. Preferred for very short critical sections on multiprocessors.
    - Semaphores: An integer variable for synchronization, accessed via atomic wait() (decrement) and signal() (increment).
        - Binary Semaphore: Value is 0 or 1 (acts like a mutex).
        - Counting Semaphore: Value is 0 to N (used to control access to N instances of a resource).
        - Waiting Queue: Semaphores can suspend (sleep()) threads instead of busy waiting, moving the thread to a queue until signaled to wake up.
    - Monitors: A high-level language construct that provides implicit mutual exclusion (only one thread active inside at a time).
        - Condition Variables: Used within monitors to allow threads to voluntarily wait (wait()) for a condition to be met and be notified (signal()).

3. Liveness and Evaluation

- Liveness Failure: Occurs when a process is waiting indefinitely.
    - Deadlock: Two or more threads are waiting for a resource held by another thread in the same set.
    - Priority Inversion: A high-priority thread is blocked by a low-priority thread, which is then preempted by a medium-priority thread, violating priority order.
- Tool Evaluation: The choice of tool depends on lock contention.
    - Low/Moderate Contention: Lock-free or CAS-based approaches are generally faster.
    - High Contention: Traditional locks that suspend waiting threads are preferred to avoid excessive CPU waste from spinning.

4. Synchronization Examples

- Bounded-Buffer: Uses mutex (for exclusive access), empty (to count available slots), and full (to count items) semaphores to coordinate the producer and consumer.
- Readers-Writers: Uses a lock (rw_mutex) and a counter (read_count) to allow multiple readers but enforce exclusive writing access.
- Dining-Philosophers: A classic resource allocation problem used to demonstrate synchronization and deadlock avoidance.

5. Operating System Implementations

- Linux:
    - Uses atomic variables, spinlocks (for SMP short sections), and semaphores/mutexes (for longer sections).
    - Avoids spinlocks on single-core systems by disabling kernel preemption instead.
- POSIX: Provides APIs for:
    - Mutex Locks (pthread_mutex_t).
    - Semaphores (Named and Unnamed) (sem_t).
    - Condition Variables (pthread_cond_t), which must be explicitly associated with a mutex for protection.

Section 2: Discussion – selected questions only (1 Hour)

1. What three conditions must be satisfied in order to solve the critical section problem?

2. Race conditions are possible in many computer systems. Consider a banking system with the following two functions: deposit(amount) and withdraw(amount). These two functions are passed the amount that is to be deposited or withdrawn from a bank account. Assume a shared bank account exists between a husband and wife and concurrently the husband calls the withdraw() function and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

3. Briefly explain Peterson's solution to the critical-section problem.

4. Write two short methods that implement the simple semaphore wait() and signal() operations on global variable, S.

5. What is the meaning of the term busy waiting? Can busy waiting be avoided altogether? Explain your answer.

6. The following program segment is used to manage a finite number of instances of an available resource. The maximum number of resources and the number of available resources are declared as follows:

```
#define MAX_RESOURCES 5
int available_resources = MAX_RESOURCES;
```

When a process wishes to obtain a number of resources, it invokes the

```
/* decrease available_resources by count resources */
/* return 0 if sufficient resources available, */
/* otherwise return -1 */
int decrease_count(int count) {
        if (available_resources < count)
              return -1;
         else {
               available_resources -= count;
               return 0;
         }
}
```

When a process wants to return a number of resources, it calls the increase_count() function:

```
/* increase available_resources by count */

int increase_count(int count) {
        available_resources += count;
        return 0;
}
```

The preceding program segment produces a race condition. Do the following:

A. Identify the data involved in the race condition.
B. Identify the location (or locations) in the code where the race condition occurs.
C. Using a semaphore, fix the race condition.

7. Describe the dining-philosophers problem and how it relates to operating systems.

Section 3: Practical exercises

Exercise 1: [Producer/Consumer problem] The following code demonstrate the producer/consumer problem. This exercise shows the problems with implementing this concept and the need for synchronization.

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

int buffer[5];
int count = 0;

void* producer()
{
    for (int i = 0; i < 10; i++) {
        if (count < 5) {
            buffer[count] = i;
            printf("PR: Produced: %d\n", i);
            count++;
            printf("PR: Counter: %d\n", count);
        }
        //sleep(1);
    }
    return NULL;
}

void* consumer()
{
    for (int i = 0; i < 10; i++) {
        if (count > 0) {
            count--;
            printf("CN: Consumed: %d\n", buffer[count]);
            printf("CN: Produced: %d\n", count);
        }
        //sleep(1);
    }
    return NULL;
}

int main()
{
    pthread_t p, c;
    pthread_create(&p, NULL, producer, NULL);
    pthread_create(&c, NULL, consumer, NULL);
    pthread_join(p, NULL);
    pthread_join(c, NULL);
    return 0;
}
```

1. Compile and run the program. Explain the output of the program.
2. What are the shared resources in this example?

3. Identify the critical section in this code.

4. Uncomment the sleep() function calls in the program, then compile and run the program. Try different combinations by uncommenting the first sleep(), then uncomment the second sleep() alone, and finally uncomment both sleep() functions. Discuss the results and record your observations.

Exercise 2: The Peterson's Algorithm

Peterson's Algorithm is a classic software-based solution for achieving mutual exclusion between two processes/threads. It was developed by Gary Peterson in 1981 and solves the critical section problem without requiring special hardware instructions.

Key Characteristics

1. Software-only solution (no special hardware needed)
2. Works for exactly 2 processes/threads
3. Guarantees mutual exclusion, progress, and bounded waiting
4. Uses only shared memory variables

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

// Shared variables for Peterson's Algorithm
bool flag[2] = {false, false};   // Interest of each thread
int turn = 0;                    // Whose turn is it

int counter = 0;  // Shared resource

// Thread 0
void* process0(void* arg) {
    for (int i = 0; i < 5; i++) {
        // Entry section (Peterson's Algorithm)
        flag[0] = true;           // I want to enter
        turn = 1;                 // Give priority to other thread
        while (flag[1] && turn == 1) {
            // Busy wait
        }

        // ===== CRITICAL SECTION =====
        printf("Thread 0: Entering critical section\n");
        counter++;
        printf("Thread 0: Counter = %d\n", counter);
        sleep(1);
        printf("Thread 0: Exiting critical section\n");
        // ============================

        // Exit section
        flag[0] = false;          // I'm done

        sleep(1);  // Remainder section
    }
    return NULL;
```

```
}

// Thread 1
void* process1(void* arg) {
    for (int i = 0; i < 5; i++) {
        // Entry section (Peterson's Algorithm)
        flag[1] = true;           // I want to enter
        turn = 0;                 // Give priority to other thread
        while (flag[0] && turn == 0) {
            // Busy wait
        }

        // ===== CRITICAL SECTION =====
        printf("Thread 1: Entering critical section\n");
        counter++;
        printf("Thread 1: Counter = %d\n", counter);
        sleep(1);
        printf("Thread 1: Exiting critical section\n");
        // ============================

        // Exit section
        flag[1] = false;          // I'm done

        sleep(1);  // Remainder section
    }
    return NULL;
}

int main() {
    pthread_t t0, t1;

    printf("=== Peterson's Algorithm Demo ===\n\n");

    pthread_create(&t0, NULL, process0, NULL);
    pthread_create(&t1, NULL, process1, NULL);

    pthread_join(t0, NULL);
    pthread_join(t1, NULL);

    printf("\nFinal counter value: %d (Expected: 10)\n", counter);

    return 0;
}

// Compile: gcc -pthread peterson.c -o peterson
```

1. Compile and run the program.
2. Study how the Peterson algorithm synchronize the process of entering the critical section between 2 processes.
3. Discuss the results.
4. Is this solution acceptable for more than 2 processes?

Exercise 3: Using the code in exercise 1 and implementing mutex locking and unlocking to protect the critical section. One process can enter the critical section.

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

int buffer[5];
int count = 0;
pthread_mutex_t lock;  // Add mutex

void* producer()
{
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&lock);    // Lock critical section

        if (count < 5) {
            buffer[count] = i;
            printf("PR: Produced: %d\n", i);
            count++;
            printf("PR: Counter: %d\n", count);
        }

        pthread_mutex_unlock(&lock);  // Unlock critical section
        sleep(1);
    }
    return NULL;
}

void* consumer()
{
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&lock);    // Lock critical section

        if (count > 0) {
            count--;
            printf("CN: Consumed: %d\n", buffer[count]);
            printf("CN: Counter: %d\n", count);
        }

        pthread_mutex_unlock(&lock);  // Unlock critical section
        sleep(1);
    }
    return NULL;
}

int main()
{
    pthread_t p, c;

    pthread_mutex_init(&lock, NULL);  // Initialize mutex

    pthread_create(&p, NULL, producer, NULL);
    pthread_create(&c, NULL, consumer, NULL);
    pthread_join(p, NULL);
    pthread_join(c, NULL);
```

```
    pthread_mutex_destroy(&lock);     // Destroy mutex

    return 0;
}

// Compile: gcc -pthread producer_consumer.c -o pc
```

1. Compile and run the code a number of times
2. Discuss the results.
3. Uncomment the sleep() functions one by one and then comment both, do you think the problem is saved? What's wrong?

Exercise 4: Solving the problem of exercise 1 using semaphores.

Semaphores are synchronization primitives that use a counter to control access to shared resources. Unlike mutexes and spinlocks, semaphores can:

- Count available resources
- Block threads (no busy waiting)
- Solve complex synchronization problems

Types of Semaphores

a) Binary Semaphore: Acts like a mutex (value 0 or 1)
b) Counting Semaphore: Can have value > 1 (counts available resources)

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0;   // Index where producer puts item
int out = 0;  // Index where consumer takes item

// Semaphores
sem_t empty;  // Count of empty slots (initially BUFFER_SIZE)
sem_t full;   // Count of full slots (initially 0)
sem_t mutex;  // Binary semaphore for mutual exclusion (initially 1)

void* producer() {
    for (int i = 0; i < 10; i++) {
        // Produce an item
        int item = i;

        sem_wait(&empty);  // Wait if buffer is full (decrement empty)
        sem_wait(&mutex);  // Enter critical section

        // Add item to buffer
        buffer[in] = item;
        printf("PR: Produced %d at index %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;  // Circular buffer

        sem_post(&mutex);  // Exit critical section
        sem_post(&full);   // Increment count of full slots

        sleep(1);
    }
    return NULL;
}

void* consumer() {
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);   // Wait if buffer is empty (decrement full)
        sem_wait(&mutex);  // Enter critical section
```

```
        // Remove item from buffer
        int item = buffer[out];
        printf("CN: Consumed %d from index %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;  // Circular buffer

        sem_post(&mutex);  // Exit critical section
        sem_post(&empty);  // Increment count of empty slots

        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t p, c;

    printf("=== Producer-Consumer using Semaphores ===\n\n");

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);  // 5 empty slots initially
    sem_init(&full, 0, 0);              // 0 full slots initially
    sem_init(&mutex, 0, 1);             // Binary semaphore (unlocked)

    pthread_create(&p, NULL, producer, NULL);
    pthread_create(&c, NULL, consumer, NULL);

    pthread_join(p, NULL);
    pthread_join(c, NULL);

    // Destroy semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    printf("\nProduction and consumption completed!\n");

    return 0;
}

// Compile: gcc -pthread semaphore.c -o semaphore
```

1. Compile and run the program and observe the output.
2. Understand the sequence of defining, creating and using semaphores to synchronize the entering of the critical section.

Exercise 5: Solving the problem in exercise 4 using binary semaphores.

Binary Semaphores can only have values 0 or 1, acting like a lock. Let me show you how to solve the producer-consumer problem using only binary semaphores.
The challenge: We need to track both empty/full slots AND provide mutual exclusion, but binary semaphores can only count 0 or 1!

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

int buffer[5];
int count = 0;

// Binary Semaphores for strict alternation
sem_t producer_turn;  // 1 = producer can go
sem_t consumer_turn;  // 1 = consumer can go

void* producer() {
    for (int i = 0; i < 10; i++) {
        sem_wait(&producer_turn);  // Wait for my turn

        // Critical Section
        if (count < 5) {
            buffer[count] = i;
            printf("PR: Produced: %d (count=%d)\n", i, count + 1);
            count++;
        }

        sem_post(&consumer_turn);  // Give turn to consumer
        sleep(1);
    }
    return NULL;
}

void* consumer() {
    for (int i = 0; i < 10; i++) {
        sem_wait(&consumer_turn);  // Wait for my turn

        // Critical Section
        if (count > 0) {
            count--;
            printf("CN: Consumed: %d (count=%d)\n", buffer[count], count);
        }

        sem_post(&producer_turn);  // Give turn to producer
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t p, c;

    printf("=== Binary Semaphores - Strict Alternation ===\n\n");
```

```
    // Initialize: Producer goes first
    sem_init(&producer_turn, 0, 1);  // Producer can start (1)
    sem_init(&consumer_turn, 0, 0);  // Consumer must wait (0)

    pthread_create(&p, NULL, producer, NULL);
    pthread_create(&c, NULL, consumer, NULL);

    pthread_join(p, NULL);
    pthread_join(c, NULL);

    sem_destroy(&producer_turn);
    sem_destroy(&consumer_turn);

    printf("\nFinal count: %d\n", count);

    return 0;
}
```

1. Compile and run the program and observe the output.
2. Discuss the structure of binary semaphores in the program.

Exercise 6: Condition variables

Condition variables are synchronization primitives that allow threads to wait for a specific condition to become true. They work together with mutexes to provide efficient thread coordination without busy-waiting.

Key Concept
Instead of repeatedly checking a condition (busy-waiting), a thread can sleep until another thread signals that the condition has changed.

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int count = 0;  // Number of items in buffer

// Mutex and Condition Variables
pthread_mutex_t mutex;
pthread_cond_t cond_producer;  // Signals when space is available
pthread_cond_t cond_consumer;  // Signals when items are available

void* producer() {
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);

        // Wait while buffer is full
        while (count == BUFFER_SIZE) {
            printf("PR: Buffer full, waiting...\n");
            pthread_cond_wait(&cond_producer, &mutex);
            // Mutex released while waiting, reacquired when woken
        }

        // Produce item
        buffer[count] = i;
        printf("PR: Produced %d (count=%d)\n", i, count + 1);
        count++;

        // Signal consumer that item is available
        pthread_cond_signal(&cond_consumer);

        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    return NULL;
}

void* consumer() {
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);

        // Wait while buffer is empty
        while (count == 0) {
```

```
            printf("CN: Buffer empty, waiting...\n");
            pthread_cond_wait(&cond_consumer, &mutex);
            // Mutex released while waiting, reacquired when woken
        }

        // Consume item
        count--;
        int item = buffer[count];
        printf("CN: Consumed %d (count=%d)\n", item, count);

        // Signal producer that space is available
        pthread_cond_signal(&cond_producer);

        pthread_mutex_unlock(&mutex);
        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t p, c;

    printf("=== Producer-Consumer using Condition Variables ===\n\n");

    // Initialize mutex and condition variables
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond_producer, NULL);
    pthread_cond_init(&cond_consumer, NULL);

    pthread_create(&p, NULL, producer, NULL);
    pthread_create(&c, NULL, consumer, NULL);

    pthread_join(p, NULL);
    pthread_join(c, NULL);

    // Cleanup
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond_producer);
    pthread_cond_destroy(&cond_consumer);

    printf("\nFinal count: %d\n", count);

    return 0;
}

// Compile: gcc -pthread condition_var.c -o condvar
```

1. Study the example above and observe the structure of defining and using the condition variables with mutes.
2. Compile and run the program.
3. Discuss the results.

Exercise 7: Dining Philosopher problem.

The Dining Philosophers Problem is a classic synchronization problem that illustrates challenges in resource allocation and deadlock avoidance.

The Problem:

- 5 philosophers sit around a circular table
- 5 chopsticks (forks) placed between them (one between each pair)
- Each philosopher needs 2 chopsticks (left and right) to eat
- Philosophers alternate between thinking and eating

The Challenge

- Deadlock can occur if:
  - All philosophers pick up their left chopstick simultaneously
  - All wait forever for their right chopstick
  - Circular wait = Deadlock!

The following code solves the dining philosopher problem using 3 methods. This is part of the next week topics (Deadlocks).

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

#define N 5    // Number of philosophers

sem_t chopsticks[N];  // One semaphore per chopstick

// ============================================================
// SOLUTION 1: Simple approach (CAN DEADLOCK!)
// ============================================================
void* philosopher_simple(void* arg) {
    int id = *(int*)arg;
    int left = id;
    int right = (id + 1) % N;

    for (int i = 0; i < 3; i++) {
        // Think
        printf("Philosopher %d is thinking\n", id);
        sleep(1);

        // Pick up chopsticks
        printf("Philosopher %d is hungry\n", id);
        sem_wait(&chopsticks[left]);   // Pick up left chopstick
        printf("Philosopher %d picked up left chopstick %d\n", id, left);
        sem_wait(&chopsticks[right]);  // Pick up right chopstick
        printf("Philosopher %d picked up right chopstick %d\n", id, right);

        // Eat
        printf("Philosopher %d is EATING\n", id);
        sleep(2);

        // Put down chopsticks
        sem_post(&chopsticks[left]);
```

```
        sem_post(&chopsticks[right]);
        printf("Philosopher %d finished eating\n", id);
    }
    return NULL;
}

// ============================================================
// SOLUTION 2: Asymmetric (Prevents Deadlock)
// ============================================================
void* philosopher_asymmetric(void* arg) {
    int id = *(int*)arg;
    int left = id;
    int right = (id + 1) % N;

    for (int i = 0; i < 3; i++) {
        printf("Philosopher %d is thinking\n", id);
        sleep(1);

        printf("Philosopher %d is hungry\n", id);

        // ASYMMETRIC: Last philosopher picks up in reverse order
        if (id == N - 1) {
            sem_wait(&chopsticks[right]);  // Right first
            printf("Philosopher %d picked up right chopstick %d\n", id,
right);
            sem_wait(&chopsticks[left]);   // Then left
            printf("Philosopher %d picked up left chopstick %d\n", id,
left);
        } else {
            sem_wait(&chopsticks[left]);   // Left first
            printf("Philosopher %d picked up left chopstick %d\n", id,
left);
            sem_wait(&chopsticks[right]);  // Then right
            printf("Philosopher %d picked up right chopstick %d\n", id,
right);
        }

        printf("Philosopher %d is EATING\n", id);
        sleep(2);

        sem_post(&chopsticks[left]);
        sem_post(&chopsticks[right]);
        printf("Philosopher %d finished eating\n", id);
    }
    return NULL;
}

// ============================================================
// SOLUTION 3: Limit Diners (Prevents Deadlock)
// ============================================================
sem_t room;  // Limit philosophers at table

void* philosopher_limited(void* arg) {
    int id = *(int*)arg;
    int left = id;
    int right = (id + 1) % N;

    for (int i = 0; i < 3; i++) {
```

```
        printf("Philosopher %d is thinking\n", id);
        sleep(1);

        // Only allow N-1 philosophers at table
        sem_wait(&room);
        printf("Philosopher %d entered dining room\n", id);

        printf("Philosopher %d is hungry\n", id);
        sem_wait(&chopsticks[left]);
        printf("Philosopher %d picked up left chopstick %d\n", id, left);
        sem_wait(&chopsticks[right]);
        printf("Philosopher %d picked up right chopstick %d\n", id, right);

        printf("Philosopher %d is EATING\n", id);
        sleep(2);

        sem_post(&chopsticks[left]);
        sem_post(&chopsticks[right]);
        printf("Philosopher %d finished eating\n", id);

        sem_post(&room);
        printf("Philosopher %d left dining room\n", id);
    }
    return NULL;
}

int main() {
    pthread_t philosophers[N];
    int ids[N];

    // Initialize chopstick semaphores
    for (int i = 0; i < N; i++) {
        sem_init(&chopsticks[i], 0, 1);  // Each chopstick available
        ids[i] = i;
    }

    printf("=== Choose Solution ===\n");
    printf("1: Simple (CAN deadlock)\n");
    printf("2: Asymmetric (NO deadlock)\n");
    printf("3: Limited diners (NO deadlock)\n");
    printf("Using Solution 2 (Asymmetric)...\n\n");

    // Create philosopher threads
    for (int i = 0; i < N; i++) {
        pthread_create(&philosophers[i], NULL, philosopher_asymmetric,
&ids[i]);
    }

    // Wait for all philosophers
    for (int i = 0; i < N; i++) {
        pthread_join(philosophers[i], NULL);
    }

    // Cleanup
    for (int i = 0; i < N; i++) {
        sem_destroy(&chopsticks[i]);
    }
```

```
    printf("\nAll philosophers finished dining!\n");

    return 0;
}

// Compile: gcc -pthread dining.c -o dining
```

Study the three solutions and understand each solution.