Interprocess Communication

Jibon Naher[1] and Samnun Azfar[2]

[1]Lecturer, CSE
[2]Junior Lecturer,CSE

December 17, 2025

1 Race Condition

1.1 What Is a Race Condition?

A race condition occurs when two or more processes (or threads) access shared data concurrently, and the final result depends on the order of execution. Because the order of execution is unpredictable, the program can produce inconsistent or incorrect results.

1.2 Critical Section

The part of code accessing shared data — must be executed by only one process at a time. for example, line 8 of above code: *counter++;*

1.3 Example 1: Race Condition in C (Two Threads Incrementing a Variable)

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

int counter = 0;  // Shared resource

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        counter++;  // Critical section
    }
    return NULL;
}
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, increment, NULL);
    pthread_create(&t2, NULL, increment, NULL);

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

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

1.4 Explanation

The following section explains the function of key lines typically found in C programs that use the `pthreads` library for thread management.

1.4.1 Line 13: `pthread_t t1, t2;`

Declares two thread identifiers (`t1` and `t2`). These are like "thread handles," used by the main program to refer to the threads later (e.g., to wait for them to finish).

1.4.2 Line 14: `pthread_create(&t1, NULL, increment, NULL);`

This is the function call to **create a new thread** that runs the `increment` function concurrently.

- `&t1`: Address of the thread ID variable where the newly created thread's identifier will be stored.
- `NULL`: Specifies the thread attributes (e.g., stack size, priority). `NULL` means default attributes are used.
- `increment`: The function pointer that the new thread will begin execution in.
- `NULL`: The single argument to be passed to the `increment` function. (In this case, none is passed.)

Note: After this line, two threads are running concurrently: the main thread (continuing in `main`) and Thread `t1` (running `increment`).

1.4.3 Line 15: `pthread_create(&t2, NULL, increment, NULL);`

Similar to the above, this creates a second thread, `t2`. Now, **three threads exist** (`main`, `t1`, and `t2`). Threads `t1` and `t2` are now competing to modify the shared resource, leading to the race condition.

1.4.4 Line 17: `pthread_join(t1, NULL);`

This is a blocking call. The main thread will pause and wait for **thread** `t1` to finish its execution (i.e., exit the `increment` function). This is crucial for ensuring the main thread does not continue until the concurrent work is complete.

1.4.5 Line 18: `pthread_join(t2, NULL);`

The main thread waits for **thread** `t2` to finish. After both `pthread_join()` calls successfully return, both worker threads are guaranteed to be finished, and it is safe for the main thread to proceed, typically by reading the final result of the shared counter.

1.5 Execute the program

command: *gcc race.c -o race -pthread* and then `./race`

2 How to fix: using mutex

```c
#include <pthread.h>
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

void* increment(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

3 Another Fix: Using Semaphore

```c
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>  // Required for semaphores

int counter = 0;  // Shared resource

// Declare a semaphore (initialized later)
sem_t sem;

void *increment(void *) {
    for (int i = 0; i < 100000; i++) {
        sem_wait(&sem);    // Acquire the semaphore (enter critical
           section)
        counter++;         // Critical section (now safe!)
        sem_post(&sem);    // Release the semaphore (exit critical
           section)
    }
}

int main() {
    pthread_t t1, t2;

    // Initialize semaphore to 1 (binary semaphore = mutex behavior)
    sem_init(&sem, 0, 1);

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

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

    // Destroy the semaphore
    sem_destroy(&sem);

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