East West University

Department of Computer Science and Engineering

Lab Experiment: Process Synchronization (Race Condition)

Objective:

- To understand the concept of **process synchronization**, **race conditions** and **shared memory**.
- To implement a simple example of race conditions using **shared memory**, **processes** and **thread** in **C**.
- To observe the behavior of the system when processes share a common resource without proper synchronization.
- To experiment with process execution order and its effect on the outcome.

Equipment and Tools:

- A C compiler (e.g., **gcc** on Linux).
- A Linux/Unix-based system (for inter-process communication and shared memory).

Pre-Lab Instructions:

- Familiarize yourself with **C programming** concepts such as pointers, processes, and memory management.
- Understand the concepts of **shared memory**, **forking**, **thread**, and **race conditions**.
- Learn how to use the `fork()` system call to create child processes and how shared memory is created using `shmget()` and `shmat()`.

Introduction:

The **race condition** occurs when multiple processes attempt to modify shared data concurrently without proper synchronization, leading to unpredictable results. The **Producer-Consumer** problem is a classic example of where race conditions can arise when two or more processes work with a common resource.

In this lab, you will work with shared memory between processes and experiment with race conditions by incrementing and decrementing a shared variable.

Independent process: An independent process in an operating system is one that does not affect or impact any other process of the system.

Cooperating Process: A cooperating process in the operating system is a process that gets affected by other processes under execution or can affect any other process under execution. These processes may share resources, data, variables, etc.

- A race condition in an operating system occurs when multiple processes or threads access and manipulate shared data concurrently, and the outcome depends on the timing or order of their execution. This can lead to unpredictable behavior, bugs, and data corruption.
- Race conditions are challenging to debug because they may not manifest consistently—often depending on the system's timing, thread scheduling, or other non-deterministic factors.

- In a word, a race condition in an operating system occurs when two or more processes or threads can access shared data, and they try to change it at the same time.

Theory:

- **Race Conditions:** A race condition happens when the outcome of a program depends on the sequence or timing of uncontrollable events, like process scheduling.
- **Shared Memory:** Shared memory allows processes to communicate by accessing a common memory space. Multiple processes can read from and write to the shared memory, but synchronization is required to avoid conflicts.
- **Forking:** The `fork()` system call creates a new process (child process) by duplicating the calling process (parent process).
- **System V Shared Memory:** It is a mechanism for allowing processes to share memory. Functions like `shmget()` create shared memory, and `shmat()` attaches it to the process.

Lab Procedure:

Step 1: Understanding Shared Memory

- **Shared Memory** is a region of memory that can be accessed by multiple processes. In this lab, we will use **System V shared memory** to create a shared buffer for processes to communicate.

Step 2: Code Overview

The following C code demonstrates how race conditions can arise when two processes interact with shared memory:

```
#include <stdio.h>

#include <stdlib.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <unistd.h>

#include <wait.h>

void *incrementFunction(int *shared);

void *decrementFunction(int *shared);

int main() {

    // Step 2.1: Create shared memory

    key_t key = ftok("shmfile", 65); // Generate a unique key

    int shmid = shmget(key, sizeof(int), 0666 | IPC_CREAT); // Create shared
memory segment

    if (shmid == -1) {

        perror("shmget failed");
```

```
        return 1;
    }
    // Step 2.2: Attach to shared memory
    int *shared = (int *)shmat(shmid, NULL, 0); // Attach shared memory to
the process
    if (shared == (int *)-1) {
        perror("shmat failed");
        return 1;
    }
    // Step 2.3: Initialize shared memory variable
    *shared = 1;
    // Step 2.4: Create two processes using fork
    pid_t pid1 = fork(); // Create the first process (process1)
    if (pid1 == 0) {
        // Child process 1 executes incrementFunction
        incrementFunction(shared);
        exit(0);
    }
    pid_t pid2 = fork(); // Create the second process (process2)
    if (pid2 == 0) {
        // Child process 2 executes decrementFunction
        decrementFunction(shared);
        exit(0);
    }
    // Step 2.5: Wait for both processes to finish
    wait(NULL);
    wait(NULL);
    // Step 2.6: Print final value of shared variable
    printf("Final value of shared is %d\n", *shared);
    // Step 2.7: Detach from shared memory
    shmdt(shared);
    // Step 2.8: Remove shared memory
```

```
    shmctl(shmid, IPC_RMID, NULL);
    return 0;
}
void *incrementFunction(int *shared) {
    int x;
    x = *shared; // process 1 reads the value of shared variable
    printf("Process1 reads the value of shared variable as %d\n", x);
    sleep(1);   // process 1 sleeps for 1 second
    x++; // process 1 increments its value
    printf("Local updation by Process1: %d\n", x);
    sleep(1);   // process 1 sleeps again to simulate context switching
    *shared = x; // process 1 updates the value of shared variable
    printf("Value of shared variable updated by Process1 is: %d\n", *shared);
}
void *decrementFunction(int *shared) {
    int y;
    y = *shared; // process 2 reads the value of shared variable
    printf("Process2 reads the value of shared variable as %d\n", y);
    sleep(1);   // process 2 sleeps for 1 second
    y--; // process 2 decrements its value
    printf("Local updation by Process2: %d\n", y);
    sleep(1);   // process 2 sleeps again to simulate context switching
    *shared = y; // process 2 updates the value of shared variable
    printf("Value of shared variable updated by Process2 is: %d\n", *shared);
}
```

Step 3: Explanation of Code Segments

1. **Shared Memory Creation and Attachment:**
    - `key_t key = ftok("shmfile", 65);` – Generates a unique key using `ftok()`. The "`shmfile`" is a file path, and 65 is a project ID.
    - `int shmid = shmget(key, sizeof(int), 0666 | IPC_CREAT);` – Creates shared memory with `shmget()` and attaches it to the process.

2. **Forking Processes:**
    - `pid_t pid1 = fork();` – Creates the first process. If `pid1 == 0`, it's the child process that runs incrementFunction.
    - `pid_t pid2 = fork();` – Creates the second process. If `pid2 == 0`, it's the second child process that runs decrementFunction.
3. **Race Condition:**
    - Both `incrementFunction()` and `decrementFunction()` read, modify, and update the shared variable without synchronization. The use of `sleep(1)` simulates context switching, allowing both processes to access and modify the shared variable concurrently.
4. **Waiting for Processes:**
    - `wait(NULL);` ensures that the parent process waits for both child processes to complete before printing the final result.
5. **Detaching and Cleaning Up Shared Memory:**
    - `shmdt(shared);` – Detaches shared memory.
    - `shmctl(shmid, IPC_RMID, NULL);` – Removes shared memory.

Step 4: Observations and Results

- **Race Condition Behavior:**
    - After executing the program, you will observe that the final value of the shared variable is unpredictable. The variable might not reflect the intended result of incrementing and decrementing due to race conditions.
- **Changing Process Order:**
    - Modify the order of `fork()` calls to see how the execution order impacts the final value of the shared variable. This demonstrates how the race condition's outcome depends on process execution order.

Tasks:

Task 1: Simulate Race Condition Using `sleep(1)`

- The `sleep(1)` calls allow processes to run for 1 second before modifying the shared memory, which introduces a delay that simulates the race condition. By observing the printed statements, you can see that the shared variable is modified by both processes without synchronization, leading to unpredictable results.

Task 2: Change Process Order

- Swap the order of `fork()` calls in the main function:
- `pid_t pid2 = fork();` // Process2 first
- `pid_t pid1 = fork();` // Process1 second

- Run the program again and compare the results with the original order. Changing the order can result in different final values of the shared variable, demonstrating the impact of process execution sequence on race conditions.

Discussion Questions:

1. **What is the effect of changing the order of the processes?**
2. **How do race conditions arise in this example?**

Example of a Race Condition in C using Thread:

Let's illustrate a race condition with a simple example in C where two threads increment a shared counter without proper synchronization. In this example, **we will not use the sleep function to provide a clearer understanding of what happens in real scenarios.**

Using Threads to Better Understand Race Conditions Compared to Processes

Why Threads Instead of Processes for Illustrating Race Conditions?

While both **threads** and **processes** can demonstrate race conditions, threads are a better choice for understanding and resolving them because of the following advantages:

1. **Shared Memory Space**:
    - **Threads** share the same memory space within a single process. This makes it easier to demonstrate race conditions, as variables (like counter in our example) are naturally accessible to all threads.
    - **Processes**, on the other hand, operate in separate memory spaces. To share data between processes, you need to use **Inter-Process Communication (IPC)** mechanisms (e.g., shared memory, message passing), which introduces additional complexity.
2. **Lightweight and Faster Context Switching**:
    - **Threads** are more lightweight than processes since they share the same code, data, and resources of the parent process. Context switching between threads is significantly faster because they operate in the same memory space.
    - **Processes**, however, require more overhead due to their independent resources and memory. Context switching between processes involves saving and restoring the state of each process, which consumes more time and resources.
3. **Ease of Synchronization**:
    - Using **mutexes** (as shown in the example) for synchronization is straightforward with threads because they work within the same memory space. You can directly lock and unlock shared resources.
    - Synchronization between **processes** requires more complex mechanisms like **semaphores**, file locks, or named mutexes, which adds additional implementation effort.
4. **Performance Benefits**:
    - Threads perform better for tasks that involve shared data or frequent communication since they avoid the overhead of process creation and memory duplication.

- Processes incur higher overhead for communication and execution, making them less efficient for such demonstrations.

5. **Practicality**:
    - Race conditions are often more prevalent in **multi-threaded** programs (e.g., web servers, multi-core computations) where multiple threads operate on shared resources.
    - Threads allow for a more realistic and practical understanding of concurrency problems like race conditions, which are commonly encountered in real-world software development.

Key Comparisons Between Threads and Processes:

Aspect	**Threads**	**Processes**
Memory Space	Shared memory space within a single process	Separate memory space for each process
Context Switching	Faster and lightweight	Slower due to overhead in switching resources
Communication	Direct access to shared variables	Requires IPC mechanisms like pipes or sockets
Resource Consumption	Lower resource consumption	Higher resource consumption
Synchronization	Easier to implement (e.g., mutexes)	More complex (e.g., named semaphores or mutexes)
Real-World Use Case	Common in multi-threaded applications	Used for isolated tasks with minimal interaction

Example Code: Race Condition

```
#include <stdio.h>
#include <pthread.h>
int counter = 0;  // Shared resource
void* increment_counter(void* arg) {
    for (int i = 0; i < 1000; i++) {
        counter++;  // Increment the shared counter
    }
    return NULL;
}
int main() {
    pthread_t thread1, thread2;
    // Create two threads
    pthread_create(&thread1, NULL, increment_counter, NULL);
```

```
    pthread_create(&thread2, NULL, increment_counter, NULL);
    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Final counter value: %d\n", counter);
    printf("Expected counter value:2000\n");
    return 0;
}
```

Explanation of the Race Condition:

Consider what happens when both threads attempt to execute counter++:

1. **Read counter value**: Each thread reads the current value of the counter (e.g., 100).
2. **Increment counter**: Each thread increments the value (e.g., 101).
3. **Write back the value**: Each thread writes the incremented value back to counter.

If both threads execute these steps simultaneously, they may both read 100, increment to 101, and then write 101 back, effectively losing one of the increments.

Task:

- Increase the number of iterations in the for loop used in the increment_counter function and observe the output.

Expected vs. Actual Outcome:

- **Expected Outcome**: If each thread increments the counter 100,000 times, the final value of the counter should be **200,000**.
- **Actual Outcome**: Due to the race condition, the final value of the counter is **likely to be less than 200,000** and will **vary between runs**. This happens because the two threads may be simultaneously reading and writing to the counter, leading to **lost updates**.

Discussion Questions:

1. **What would happen if you added synchronization mechanisms, such as mutexes, to this code?**
2. **Can this race condition be avoided using a single process?**
3. **How can this lab help you understand real-world scenarios?**
4. **What are the potential consequences of race conditions in software systems?**

Fixing the Race Condition:

To prevent a race condition, you need to synchronize access to the shared resource (in this case, the counter). This can be done using various synchronization mechanisms like **mutexes** (mutual exclusion locks).

Example Code: Using a Mutex to Prevent the Race Condition

```
#include <stdio.h>
#include <pthread.h>
int counter = 0;  // Shared resource
pthread_mutex_t lock;  // Mutex for synchronization
void* increment_counter(void* arg) {
    for (int i = 0; i < 100000; i++) {
        pthread_mutex_lock(&lock);  // Lock the mutex
        counter++;  // Increment the shared counter
        pthread_mutex_unlock(&lock);  // Unlock the mutex
    }
    return NULL;
}
int main() {
    pthread_t thread1, thread2;
    pthread_mutex_init(&lock, NULL);  // Initialize the mutex
    // Create two threads
    pthread_create(&thread1, NULL, increment_counter, NULL);
    pthread_create(&thread2, NULL, increment_counter, NULL);
    // Wait for both threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    pthread_mutex_destroy(&lock);  // Destroy the mutex
    printf("Final counter value: %d\n", counter);
    printf("Expected counter value:200000\n");
    return 0;
}
```

Explanation of the Fix:

1. **Mutex Initialization**:

2. `pthread_mutex_init(&lock, NULL);`

    - A mutex is initialized before the threads are created.

3. **Locking and Unlocking**:

    - **`pthread_mutex_lock(&lock);`:** Before modifying the shared counter, the thread locks the mutex, ensuring that no other thread can access the counter simultaneously.

    - **`pthread_mutex_unlock(&lock);`:** After modifying the counter, the thread unlocks the mutex, allowing other threads to proceed.

4. **Mutex Destruction**:

5. `pthread_mutex_destroy(&lock);`

    - After all threads are done, the mutex is destroyed to free resources.

Output with Synchronization:

With the mutex in place, the output will consistently be:

Final counter value: 200000

Summary:

Race conditions occur when the correctness of a program depends on the relative timing of threads or processes accessing shared data. They can lead to **unpredictable** and **incorrect** results. By using synchronization mechanisms like **mutexes**, you can serialize access to shared resources and prevent race conditions, ensuring that your program behaves **consistently** and **correctly**.