Lab 4: Multithreading

01-09-2025

Aim

Study and implement multithreading, and explore synchronization primitives (mutex, condition variables, semaphores).

Learning Outcomes

By the end, students can:

- Differentiate processes vs. threads.
- Create, join, and manage threads (`pthread_create`, `pthread_join`).
- Pass arguments and return values from threads.
- Identify race conditions and protect critical sections with a mutex.
- Coordinate threads using condition variables and semaphores.

Prerequisites

- Linux/WSL with GCC.
- Basic C (pointers, functions, heap allocation).
- Compile with: `gcc file.c -pthread -o file`

1 Introduction

Modern operating systems support *concurrency* - running multiple tasks seemingly at the same time.[1]

Concurrency mechanisms

- **Processes**: independent execution units with their own memory.
- **Threads**: lightweight execution units that share memory within a process.

Why threads ? Responsiveness & parallelism. Typical examples:

- **Browser**: separate threads for rendering, networking, JavaScript.
- **Editor**: one thread for user input, another for background formatting.

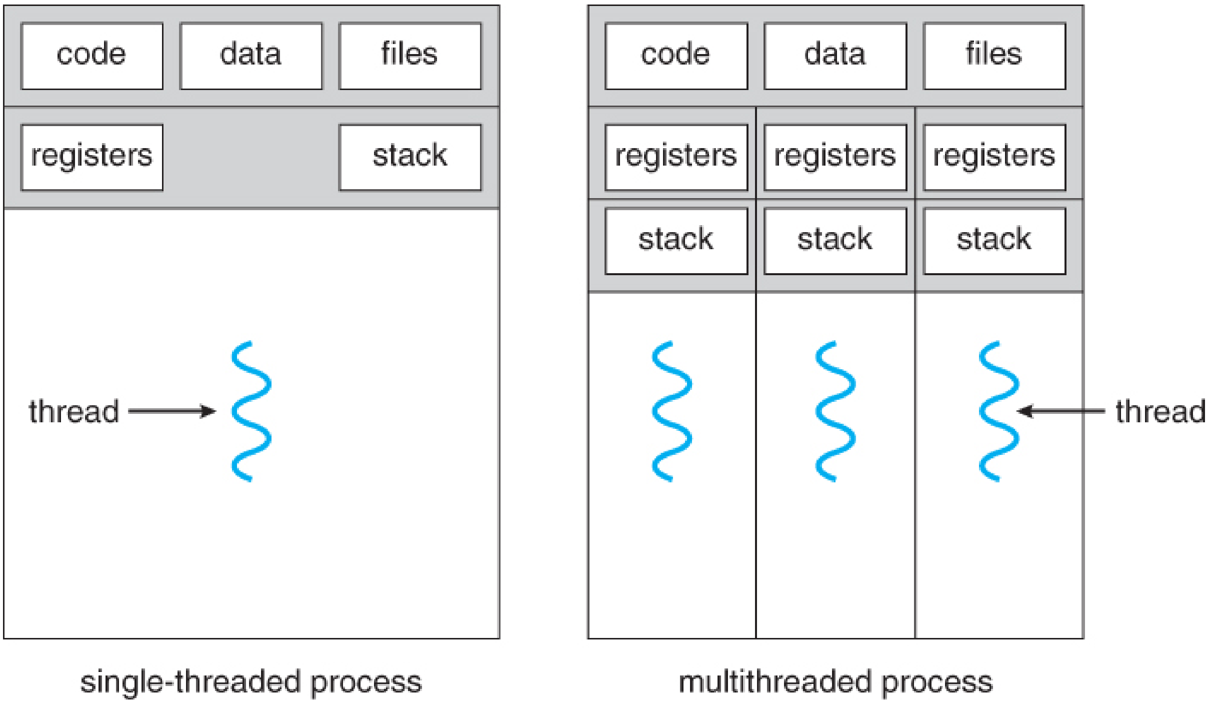

Figure 1: Processes vs. Threads: separate address spaces vs. shared address space (own stacks/registers).

2 Process vs Thread

- **Process**: program in execution with its own virtual address space.
- **Thread**: execution context within a process; threads share code/data/files but have *their own* stacks and CPU registers.

Feature	**Process**	**Thread**
Memory	Independent address space	Shares parent's address space
Communication	IPC (pipes, sockets, shared mem)	Direct via shared memory
Overhead	Heavyweight (context switch costly)	Lightweight
Creation/Termination	Slower (OS involvement)	Faster
Crash Impact	Other processes unaffected	May crash entire process

[1]On a single CPU, the scheduler time-slices; on multi-core systems, tasks may run truly in parallel.

3 POSIX Threads (pthreads) - Quick Reference

Headers & compile `#include <pthread.h>` (semaphores: `#include <semaphore.h>`)
Compile/link with: `gcc file.c -pthread -o app`

Core APIs

- `int pthread_create(pthread_t*, const pthread_attr_t*, void*(*)(void*), void*);`
- `int pthread_join(pthread_t, void**)` (collect optional return value);
- `void pthread_exit(void*)` (terminate calling thread);

Synchronization primitives

- `pthread_mutex_t` - mutual exclusion (protect critical sections).
- `pthread_cond_t` - condition variable (wait/signal for conditions).
- `sem_t` - semaphore (binary or counting) for resource control.

4) Basic Thread Programs

This section introduces how to create and manage threads, pass arguments, return values, and understand why output order may vary.

4.1 Creating a Thread

Theory: A thread runs a function concurrently with `main`. Use `pthread_create` to start it and `pthread_join` to wait for it to finish. Always join threads you create unless they are intentionally detached.

Code:

Listing 1: Hello from a thread

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

void* hello(void* arg) {
    printf("Hello from thread!\n");
    return NULL;
}

int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, hello, NULL);
    pthread_join(t1, NULL);
    printf("Thread finished.\n");
    return 0;
}
```

Expected Output:

```
[202463010@paramshavak ~]$ nano hello_thread.c
[202463010@paramshavak ~]$ gcc -pthread hello_thread.c -o hello_thread
[202463010@paramshavak ~]$ ./hello_thread
Hello from thread!
Thread finished.
```

Student Task: Create 3 threads, each printing a different message (e.g., “from T1/T2/T3”).

4.2 Passing Arguments

Theory: Pass a pointer as the fourth argument of `pthread_create`. Inside the thread, cast it to the right type. **Pitfall:** do not pass the address of a loop variable that keeps changing; use a per-thread stable storage (e.g., an array).

Code:

Listing 2: Passing an integer argument to a thread

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

void* print_num(void* arg) {
    int num = *(int*)arg;
    printf("Thread received: %d\n", num);
    return NULL;
}

int main() {
    pthread_t t1;
    int value = 42;
    pthread_create(&t1, NULL, print_num, &value);
    pthread_join(t1, NULL);
    return 0;
}
```

Expected Output:

```
[202463010@paramshavak ~]$ nano pass_int.c
[202463010@paramshavak ~]$ gcc -pthread pass_int.c -o pass_int
[202463010@paramshavak ~]$ ./pass_int
Thread received: 42
```

Student Task: Launch multiple threads, pass each its array index via a stable `int ids[N];` array, and print that index in the thread.

4.3 Returning Values

Theory: A thread can return a pointer via `pthread_exit`. Because the thread's stack disappears when it ends, return data must live beyond the thread (typically heap-allocated with `malloc`). The creator retrieves it with `pthread_join`.

Code:

Listing 3: Thread computes and returns a square

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

void* square(void* arg) {
    int num = *(int*)arg;
    int *result = malloc(sizeof(int));
    *result = num * num;
    pthread_exit(result);
}

int main() {
    pthread_t t1;
    int value = 7;
    int *res = NULL;

    pthread_create(&t1, NULL, square, &value);
    pthread_join(t1, (void**)&res);

    printf("Square of %d = %d\n", value, *res);
    free(res);
    return 0;
}
```

Expected Output:

```
[202463010@paramshavak ~]$ nano square_thread.c
[202463010@paramshavak ~]$ gcc -pthread square_thread.c -o square_thread
[202463010@paramshavak ~]$ ./square_thread
Square of 7 = 49
```

Student Task: Spawn 5 threads; each computes the factorial of its assigned number and returns it via `pthread_exit`. In `main`, print and free each result.

4.4 Multiple Threads (Non-deterministic order)

Theory: Threads run concurrently; the scheduler decides who runs when. Therefore, print statements from different threads may interleave in different orders across runs. This is normal and called *non-determinism of scheduling.*

Code:

Listing 4: Three workers running concurrently

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

void* worker(void* arg) {
    int id = *(int*)arg;
    printf("Thread %d is running\n", id);
    return NULL;
}

int main() {
    pthread_t threads[3];
    int ids[3] = {1, 2, 3};

    for (int i = 0; i < 3; i++)
        pthread_create(&threads[i], NULL, worker, &ids[i]);

    for (int i = 0; i < 3; i++)
        pthread_join(threads[i], NULL);

    printf("All threads finished.\n");
    return 0;
}
```

Expected Output (order varies):

```
[202463010@paramshavak ~]$ nano three_threads.c
[202463010@paramshavak ~]$ gcc -std=c99 -pthread -O2 three_threads.c -o three_threads
[202463010@paramshavak ~]$ ./three_threads
Thread 1 is running
Thread 2 is running
Thread 3 is running
All threads finished.
```

Student Task: Create 10 threads; each prints whether its ID is even or odd (e.g., “ID 6 is even”).

5) Concurrency Problems

5.1 Race Condition (Problem)

What it is: When two or more threads read/modify the same data *at the same time* without coordination, their steps can interleave in a harmful way. Example: both threads read the old value of a counter before either writes the new value, so one increment is "lost." The final result becomes *unpredictable* and often *incorrect*.

Unsynchronized increment (shows the bug):

Listing 5: Race condition: two threads increment a shared counter without protection

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

volatile int counter = 0;

void* inc(void* arg) {
    int i;
    for (i = 0; i < 1000000; i++) {
        counter++;
        if ((i & 1023) == 0) {
            sched_yield();
        }
    }
    return NULL;
}

int main(void) {
    pthread_t t1, t2;

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

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

    printf("Final Counter = %d (expected %d)\n", counter, 2000000);
    return 0;
}
```

Expected Output:

```
[202463010@paramshavak ~]$ nano race_counter.c
[202463010@paramshavak ~]$ gcc -std=c11 -pthread -O2 race_counter.c -o race_counter
[202463010@paramshavak ~]$ ./race_counter
Final Counter = 1142105 (expected 2000000)
```

Fix idea: Guard the update with a `pthread_mutex_t` (see Critical Section below).

5.2 Critical Section

What it is: A region of code that accesses shared data and must not be executed by more than one thread at a time.
How to protect it: Use a **mutex** (mutual exclusion lock) to ensure one-at-a-time entry.
Mutex-protected increment (fixes the race):

Listing 6: Protecting a critical section with a mutex

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

#define ITERS 1000000

int counter = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void* inc(void* arg) {
    int i;
    for (i = 0; i < ITERS; i++) {
        pthread_mutex_lock(&m);
        counter++;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}

int main(void) {
    pthread_t t1, t2;

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

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

    printf("Final Counter = %d (expected %d)\n", counter, 2*ITERS);

    pthread_mutex_destroy(&m);
    return 0;
}
```

Expected Output:

```
[202463010@paramshavak ~]$ nano counter_mutex.c
[202463010@paramshavak ~]$ gcc -pthread -O2 counter_mutex.c -o counter_mutex
[202463010@paramshavak ~]$ ./counter_mutex
Final Counter = 2000000 (expected 2000000)
```

5.3 Deadlock

What it is: Two (or more) threads each hold a resource and wait forever for the other to release theirs. No one can make progress.
Common cause: Locks taken in different orders in different places.

Example with reversed lock order (may hang):

Listing 7: Deadlock: each thread holds one lock and waits for the other

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

pthread_mutex_t lock1, lock2;

void* task1(void* arg) {
    pthread_mutex_lock(&lock1);
    sleep(1);
    pthread_mutex_lock(&lock2);
    printf("Task 1 finished\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}

void* task2(void* arg) {
    pthread_mutex_lock(&lock2);
    sleep(1);
    pthread_mutex_lock(&lock1);
    printf("Task 2 finished\n");
    pthread_mutex_unlock(&lock1);
    pthread_mutex_unlock(&lock2);
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);

    pthread_create(&t1, NULL, task1, NULL);
    pthread_create(&t2, NULL, task2, NULL);

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

    // this program may never reach here due to deadlock
    return 0;
}
```

What to expect: The program may *hang* (both threads wait forever).

```
[202463010@paramshavak ~]$ nano deadlock.c
[202463010@paramshavak ~]$ gcc -pthread -O2 deadlock.c -o deadlock
[202463010@paramshavak ~]$ ./deadlock
^C
```

Prevention Tips:

- Impose a global lock order (e.g., always acquire `lock1` then `lock2`).
- Keep critical sections short; avoid holding multiple locks if possible.
- Consider `pthread_mutex_trylock` or timeouts/backoff for recovery strategies.

Fix (global lock order):

Listing 8: Fix: enforce a single global lock order (lock1 then lock2)

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

pthread_mutex_t lock1, lock2;

void* task1(void* arg) {
    pthread_mutex_lock(&lock1);
    sleep(1);
    pthread_mutex_lock(&lock2);
    printf("Task 1 finished\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}

void* task2(void* arg) {
    pthread_mutex_lock(&lock1);
    sleep(1);
    pthread_mutex_lock(&lock2);
    printf("Task 2 finished\n");
    pthread_mutex_unlock(&lock2);
    pthread_mutex_unlock(&lock1);
    return NULL;
}

int main(void) {
    pthread_t t1, t2;
    pthread_mutex_init(&lock1, NULL);
    pthread_mutex_init(&lock2, NULL);

    pthread_create(&t1, NULL, task1, NULL);
    pthread_create(&t2, NULL, task2, NULL);

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

    pthread_mutex_destroy(&lock1);
    pthread_mutex_destroy(&lock2);
    return 0;
}
```

Expected Output:

```
[202463010@paramshavak ~]$ ./deadlock_fixed
Task 1 finished
Task 2 finished
```

6) Synchronization Mechanisms

6.1 Mutex (Mutual Exclusion)

Idea: Protect a critical section so only one thread executes it at a time.
Code: See Section 3 (*Critical Section*).

6.2 Condition Variables

What it is: A thread *waits* until a condition becomes true; another thread *signals* when ready. Useful in producer–consumer.

Code (simple signal/wait):

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

int ready = 0;
pthread_mutex_t lock;
pthread_cond_t cond;

void* consumer(void* arg) {
    pthread_mutex_lock(&lock);
    while (!ready) pthread_cond_wait(&cond, &lock);
    printf("Consumer: Resource ready!\n");
    pthread_mutex_unlock(&lock);
    return NULL;
}

void* producer(void* arg) {
    pthread_mutex_lock(&lock);
    ready = 1;
    printf("Producer: Sending signal.\n");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_mutex_init(&lock, NULL);
    pthread_cond_init(&cond, NULL);
    pthread_create(&t1, NULL, consumer, NULL);
    pthread_create(&t2, NULL, producer, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_mutex_destroy(&lock);
    pthread_cond_destroy(&cond);
    return 0;
}
```

Expected Output:

```
[202463010@paramshavak ~]$ nano pc_oneflag.c
[202463010@paramshavak ~]$ gcc -std=c11 -pthread -O2 pc_oneflag.c -o pc_oneflag
[202463010@paramshavak ~]$ ./pc_oneflag
Producer: Sending signal.
Consumer: Resource ready!
```

6.3 Semaphores (Binary / Counting)

What it is: A semaphore is a counter controlling how many threads may access a region. Binary (value 1) behaves like a mutex; counting (value N) allows up to N concurrent entrants.

Code (counting semaphore, allow 2 at a time):

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

sem_t sem;

void* worker(void* arg) {
    int id = *(int*)arg;
    sem_wait(&sem);
    printf("Thread %d entered\n", id);
    sleep(1);
    printf("Thread %d leaving\n", id);
    sem_post(&sem);
    return NULL;
}

int main() {
    pthread_t t[5];
    int ids[5];
    sem_init(&sem, 0, 2);
    for (int i = 0; i < 5; i++) {
        ids[i] = i + 1;
        pthread_create(&t[i], NULL, worker, &ids[i]);
    }
    for (int i = 0; i < 5; i++) pthread_join(t[i], NULL);
    sem_destroy(&sem);
    return 0;
}
```

Expected Output (order varies):

```
[202463010@paramshavak ~]$ nano semaphore.c
[202463010@paramshavak ~]$ gcc -std=c11 -pthread -O2 semaphore.c -o semaphore
[202463010@paramshavak ~]$ ./semaphore
Thread 1 entered
Thread 2 entered
Thread 1 leaving
Thread 4 entered
Thread 2 leaving
Thread 3 entered
Thread 4 leaving
Thread 3 leaving
Thread 5 entered
Thread 5 leaving
```

ASSIGNMENT NOTICE: The assignment for this lab will be given by the TAs during the lab session.