POSIX Threads (pthreads)

Eng. Ahmed Métwalli

December 12, 2024

1 Introduction

This document summarizes the concepts, functions, and reasoning behind a series of C code examples demonstrating the creation, management, and coordination of threads using the POSIX Threads (pthreads) library. The presented code snippets progressively introduce key concepts, from basic thread creation and joining, to passing arguments, handling concurrency with multiple threads, and imposing dependencies between threads.

2 Comparison of Threads and Alternatives

Feature	**Threads**	**Processes**	**Event-Driven**	**Async/Await**
Problem Solved	Concurrent tasks within a process.	Isolated tasks with separate memory.	Non-blocking I/O tasks.	Lightweight, non-blocking I/O.
Key Features	Shared memory, parallelism.	Isolation via separate memory spaces.	Event loop manages tasks.	Simple non-blocking syntax.
Performance	High for shared memory.	Higher overhead due to isolation.	High for I/O-bound tasks.	Efficient for I/O tasks, limited for CPU-bound tasks.
Complexity	Needs careful synchronization to avoid issues.	Easier debugging; IPC is complex.	Simplifies concurrency, no parallelism.	Easier but requires async-compatible libraries.
Advantages	Fast communication, low overhead.	Fault isolation, safe execution.	Avoids thread-related bugs.	Simplifies async workflows.
Disadvantages	Risk of race conditions and deadlocks.	High resource usage, slow IPC.	Not suitable for CPU-bound tasks.	No true parallelism for CPU-bound tasks.
Best Use Cases	Real-time systems, parallel computations.	Long-running isolated tasks (e.g., database servers).	GUI, networked applications (e.g., HTTP servers).	Lightweight I/O-bound applications.

Table 1: Comparison of Threads and Their Alternatives

3 Race Conditions and Deadlocks

3.1 Race Conditions

A **race condition** occurs when multiple threads or processes access shared data and try to modify it simultaneously, leading to unpredictable behavior.

Example:

```
blueint counter = 0;

bluevoid* increment() {
    bluefor (blueint i = 0; i < 1000; i++) {
        counter++; green!50!black//green!50!black green!50!blackSimultaneous
           green!50!black green!50!blackaccessgreen!50!black green!50!blackcan
           green!50!black green!50!blackcausegreen!50!black green!50!blackincorrect
           green!50!black green!50!blackresults
    }
    bluereturn NULL;
}
```

Without proper synchronization, the result of the ‘counter‘ may be incorrect.

Solution: Use **mutexes** (mutual exclusion) to ensure only one thread accesses the critical section at a time.

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);

pthread_mutex_lock(&lock);
green!50!black//green!50!black green!50!blackCriticalgreen!50!black
   green!50!blacksection
counter++;
pthread_mutex_unlock(&lock);
```

3.2 Deadlocks

A **deadlock** occurs when two or more threads are waiting for resources held by each other, creating a circular dependency where no thread can proceed.

Example:

- Thread 1 locks `Resource A` and waits for `Resource B`.
- Thread 2 locks `Resource B` and waits for `Resource A`.

Solution:

- Ensure consistent lock ordering.
- Avoid holding multiple locks simultaneously.
- Use **timed locks** to detect and recover from potential deadlocks.

—

By addressing these issues, threads can be used safely and effectively for concurrent programming.

4 Key Terms and Concepts

- **Thread:** A thread is an independent flow of control within a process. Multiple threads can run concurrently, sharing the same address space, which allows for parallelism and more efficient utilization of CPU resources.
- **POSIX Threads (pthreads):** A standardized C library (IEEE POSIX 1003.1c) providing functions for creating, synchronizing, and managing threads. The library includes data types and functions such as `pthread_t`, `pthread_create()`, `pthread_join()`, etc.
- **Thread Routine:** A function executed by a thread. It must have the signature:

  ```
  void *routine(void *arg)
  ```

  where `arg` points to arguments passed at thread creation time.
- **pthread_create():** A function that spawns a new thread. It takes a thread identifier, optional attributes, a routine to run, and a pointer to arguments: `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg)`
- **pthread_join():** A function that blocks the calling thread until the specified thread terminates. This ensures deterministic behavior and the proper sequencing of program events.
- **Synchronization and Dependencies:** Ensuring that certain threads finish before others start, or coordinating the order of execution, is critical in multithreaded programs. In these examples, `pthread_join()` is used to impose an order of execution (i.e., waiting on a prerequisite thread before proceeding).
- **Thread Arguments:** Passing data to threads can be done by providing a pointer to any data structure. Often, a custom `struct` is defined to bundle all necessary parameters, ensuring that the thread function has well-structured input.
- **Random Delays and Sleep:** Functions such as `sleep()` and `usleep()` (microsecond-level granularity) are used to simulate computation time or random delays, showcasing concurrency and interleaving of thread output.

5 Progression of the Examples

5.1 Basic Thread Creation and Execution

In the first code snippet, a single thread is created using `pthread_create()`. The main steps are:

1. Define a thread routine (e.g., `my_routine()`) that prints output and sleeps.
2. In `main()`, create a `pthread_t` variable, and call `pthread_create()` to start the thread executing `my_routine()`.
3. While the thread runs, `main()` also performs its own tasks (printing and sleeping).
4. Once the main loop is done, call `pthread_join()` to wait for the child thread to finish before terminating the program.

This demonstrates:

- How to create a thread and run a function in parallel.
- The use of `pthread_join()` to synchronize the completion of threads.

5.2 Passing Arguments to Threads

The second code snippet shows how to pass arguments to threads:

1. Instead of a fixed routine, pass a pointer to a variable (e.g., a character `char c`) that identifies the thread.
2. In the thread routine, cast the `void *` argument back to the appropriate type, and use its value.
3. Create multiple threads, each with different arguments (e.g., `'A'` and `'B'`), and observe interleaved execution.

This demonstrates:

- How to differentiate threads by passing arguments.
- The importance of properly casting and handling arguments in a thread-safe manner.

5.3 Multiple Threads with Random Delays

In the next snippets, multiple threads (e.g., 10 threads) are created, each labeled with consecutive characters (`'A', 'B', 'C', ...'`). Random delays are introduced via `usleep()` with a random argument, simulating unpredictable computation times and showcasing concurrency more realistically.

This demonstrates:

- How multiple threads run in parallel, each performing its own work.
- The concept of non-deterministic interleaving of thread output due to concurrent execution and random delays.

5.4 Using a `struct` to Bundle Thread Arguments

Another snippet introduces a `struct` (e.g., `Targs`) to group arguments such as a label and a limit. Each thread receives a pointer to its respective `Targs` structure, enabling:

- Cleaner argument passing: Instead of passing multiple arguments or using global variables, all necessary data is encapsulated in a single structure.
- Easier code maintenance and readability.

5.5 Imposing Dependencies Between Threads

In the final snippet, threads are arranged such that some depend on the completion of others. This is done by:

1. Storing the thread ID of a "prerequisite" thread in the argument struct.
2. Calling `pthread_join()` on that prerequisite thread inside the newly created thread before proceeding with its own work.

This demonstrates:

- How to impose an execution order in a multithreaded environment.
- The use of `pthread_join()` not just in the main thread, but also in other worker threads for synchronization.

6 General Approach and Best Practices

When starting to write multithreaded code, consider the following steps and guidelines:

1. **Identify concurrency opportunities:** Determine which parts of your computation can be performed in parallel.
2. **Define thread routines:** Write functions that encapsulate the work to be done by each thread. Keep them as independent and thread-safe as possible.
3. **Pass arguments carefully:** Use pointers to data structures to avoid global variable reliance. Ensure that any shared data is properly synchronized if needed.
4. **Create and start threads:** Use `pthread_create()` to spawn threads. Check return values for error handling.
5. **Synchronization:** When threads must cooperate or proceed in a certain order, use `pthread_join()` or other synchronization primitives (like mutexes, condition variables, or semaphores) as needed.
6. **Wait for completion:** In the main thread, call `pthread_join()` to wait for all worker threads to finish before the program exits. This ensures a clean shutdown and proper resource cleanup.
7. **Testing and Debugging:** Threaded code can be non-deterministic. Test multiple times, use debug statements, and consider tools such as `valgrind` or thread sanitizers to detect race conditions or memory issues.

7 Conclusion

By reviewing these code examples and the associated explanations, one gains a practical understanding of how to:

- Create and manage threads using `pthread_create()` and `pthread_join()`.
- Pass arguments to threads and handle multiple threads concurrently.
- Use structures to organize thread arguments.
- Impose order and dependencies among threads using synchronization primitives.

These building blocks are essential for developing more complex, efficient, and maintainable multithreaded programs.