Chapter 5a: Multithreaded Programming and Mutexes

Gene Cooperman

THIS IS A WORK IN PROGRESS.

1 `dotask.c` Source Code

Let's look at our first multithreaded program.

```c
#include <pthread.h>
#define NUM_THREADS 5
int task_num = 0;
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

// The "start function" for a new thread is do_task.
void *do_task(void *arg) {
    pthread_mutex_lock(&mymutex);
    int mytask = task_num++;
    pthread_mutex_unlock(&mymutex);
    do_work(mytask);
    return NULL;
}
int main() {
    pthread_t thread_ids[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++) {
        // Create a new thread with the start function, do_task.
        pthread_create(&thread_ids[i], NULL, do_task, NULL);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        // Let the thread exit when the start function finishes.
        pthread_join(thread_ids[i], NULL);
    }
    return 0;
}
```

---

2 Thread-to-Process Analogy

The **pthreads** model shares key functional similarities with the traditional Unix process management model (`fork` and `execvp`), which helps in understanding their roles:

pthreads (Threads)	Unix Processes (Process)	Description
`pthread_create`	**`fork` combined with `execvp`**	Launches a new flow of control. `fork` creates a copy, and `execvp` replaces the code with a new program. `pthread_create` launches a new flow of control that runs a **start function** (`do_task`) within the **same program's memory space**.
`do_task` (**Start Function**)	**The `main()` function of the program launched by `execvp`**	This is the entry point where execution begins for the new entity (thread or process).
`pthread_t` (**Thread Descriptor**)	**`pid` (Process ID)**	A unique identifier used by the operating system and the parent to manage and refer to the newly created flow of control.
`pthread_join`	**`waitpid`**	A synchronous call where the calling thread/process pauses execution until the **start function returns (analogous to a child process exiting)**.

The full `man` pages for `pthread_create` and `pthread_join` show how to extend this example by specifying an argument inside `pthread_create` and passing the argument to the start function, `do_task`, and then passing the return value of the start function back to the calling thread through `pthread_join`. For the purpose of this document, we will focus only on the core functionality for creating, joining, and synchronizing threads, assuming the reader can discover these extensions on their own by consulting the man pages.

CAVEAT: By default, when the primary thread (original thread) of a process returns from its `main()` function, then the entire process will exit, *even if some secondary threads are still executing.* If you don't like this behavior, then investigate the man pages for `pthread_detach` and `pthread_attr_init`.

---

3 What Goes Wrong Without a Mutex? (Race Condition)

If the calls to `pthread_mutex_lock(&mymutex);` and `pthread_mutex_unlock(&mymutex);` were removed, and if the program were run many times, then the final value of `task_num` would be incorrect (less than 5) during some executions.

This type of bug is called a **race condition**. A race condition in a multithreaded program is a software bug in which two or more threads change the same piece of data at the same time. The schedule chosen by the opering system determines which thread wrote the data last, and therefore, which thread "won the race".

The problem arises because the operation `int mytask = task_num++;` is not a single, indivisible step. Instead, the CPU must perform at least three separate operations to complete it: **READ** (read the current value), **INCREMENT** (add 1), and **WRITE** (write the new value back to memory).

In a multithreaded environment without a mutex, the operating system can switch between threads at any point, interleaving these steps. This can result in two or more threads reading the same old value of `task_num` before either one writes its updated value back, causing one or more increments to be **lost**.

---

4 Understanding the Critical Section and Mutual Exclusion

A **mutex** is short for **Mutual Exclusion**. It is a synchronization tool that enforces the rule that **at most one thread can access a shared resource at any given time**.

4.1 The Critical Section

Any code that accesses shared resources (like the global variable `task_num`) and needs protection from concurrent access is called a **critical section**.

For your program, `dotask.c`, the lines between the lock and unlock calls define the critical section for this specific mutex:

```
pthread_mutex_lock(&mymutex);
int mytask = task_num++;
pthread_mutex_unlock(&mymutex);
```

- **Mutual Exclusion:** If Thread A locks `mymutex` and enters the critical section, any other thread (like Thread B) attempting to call `pthread_mutex_lock(&mymutex)` will be **blocked** (paused) until Thread A calls `pthread_mutex_unlock(&mymutex)`.

4.2 Critical Section Contiguity

It's important to note that the critical section defined by a mutex **need not be contiguous** in the source code. The critical section for a single mutex includes **all instances of the code protected by that same mutex's lock/unlock pair**, regardless of where they appear in the program. The mutex ensures that entry into one instance of the critical section prevents entry into any other instance protected by the same lock.

---

5 Performance and Limiting the Size of the Critical Section (Coarse-Grained vs. Fine-Grained Locking)

A core design principle for using mutexes is to ensure that:

threads shold spend only a minimal amount of time inside the critical section.

To understand why this is important, let's revisit our function `do_task()`, and consider what happens when we violate this principle. We could have written `do_task()` to keep together the initialization of the local variable `mytask` and the use of that local variable:

```
void *do_task_SIMPLE(void *arg) {
    pthread_mutex_lock(&mymutex);
    int mytask = task_num++;
    do_work(mytask);
    pthread_mutex_unlock(&mymutex);
    return NULL;
}
```

The function `do_task_SIMPLE()` will always produce the correct answer. Its problem is performance. Only one thread can be in the critical section at a time. If `do_work()` dominates the CPU time, then the effect is to serialize execution of `do_work()`. We can no longer use multiple CPUs in parallel.

Any CPU-intensive work should be done *outside* of the critical section. This maximizes concurrency by allowing other threads to run their compute work without being blocked by the mutex.

We see this principle applied in `do_task()`. The thread calculates the unique task identifier, `mytask`, **inside** the critical section (where `task_num` is shared and needs protection). However, the CPU-intensive work performed by `do_work(mytask)` is executed **outside** the critical section. This keeps the lock held for the shortest possible duration, improving overall performance.

```
void *do_task(void *arg) {
    // 1. Enter brief critical section to safely read/update shared data
    pthread_mutex_lock(&mymutex);
    int mytask = task_num++; // CRITICAL SECTION: Only a few CPU cycles
    pthread_mutex_unlock(&mymutex);

    // 2. Perform long-running, independent work outside of the lock
    do_work(mytask);          // NOT in critical section
    return NULL;
}
```

---

6 How a Mutex is Implemented Conceptually

Conceptually, a **mutex** can be implemented with just a **single bit** that tracks the lock's state, along with supporting logic to manage waiting threads. This implementation is designed to enforce the "at most one thread" rule with high efficiency.

6.1 1. The Core State

A mutex fundamentally maintains one of two states: **UNLOCKED** (≈ 0) or **LOCKED** (≈ 1).

6.2 2. The Atomic Operation

The challenge in implementing a mutex is avoiding a race condition in the mutex implementation itself. To address this, the `mutex_lock()` function must perform two critical steps **atomically** (as a single, indivisible hardware operation):

- **Test:** Test if the mutex state is **UNLOCKED**.
- **Set:** If it is UNLOCKED, change the state to **LOCKED** and allow the thread to proceed.

This combined **Test-and-Set** operation prevents two threads from simultaneously seeing the state as UNLOCKED and both attempting to acquire the lock. Modern CPUs provide special instructions (like `compare_and_swap` or `exchange`) to guarantee this atomicity.

The keyword **"and"** in instruction names like **Test-and-Set** or **Compare-and-Swap** signifies that two distinct operations (e.g., a read/test **and** a write/set) are executed **atomically**, meaning they occur without the possibility of interruption or interleaving by a different thread or processor. This concept of atomicity is also found in standard assembly instructions, such as the common **Jump-and-Link (JAL)** instruction in architectures like **MIPS or RISC-V**, where the Program Counter (PC) register is updated for the jump **and** the Return Address (RA) register is simultaneously set to the next sequential instruction's address, both actions guaranteed to complete without interruption.

6.3 3. Spinning vs. Sleeping

When a thread attempts to lock a mutex that is already LOCKED, the `mutex_lock()` function follows a combined strategy:

6.3.1 a. Spinlock (Busy Waiting)

The thread initially executes a **spinlock**, repeatedly testing if the mutex is still locked. This is fast if the lock is released *very quickly* (avoiding a context switch), but wastes CPU cycles (busy waiting) if the lock is held for long.

6.3.2 b. Sleeping (Kernel Intervention)

After some iterations of the spinlock, if the lock is not gained, the thread's state transitions to **SLEEPING** (or **BLOCKED**) and it is placed into a waiting queue for that mutex. This frees the CPU but incurs the high overhead of a **context switch**.

6.4 4. Mutex Unlock

When `mutex_unlock()` is called:

1. The mutex state is reset to **UNLOCKED**.
2. The function detects if any thread is **sleeping** on this mutex and, if so, **wakes it up** (changes its state to RUNNABLE).

---

7 The Danger of Hidden Race Conditions (The Leader Election Problem)

Race conditions are not limited to simple arithmetic operations; they can corrupt any shared logic that relies on the sequence of *reading* and *writing* a shared variable. A classic example is a simple **leader election** mechanism intended to ensure only one thread becomes the leader.

Leader election is an important algorithm in thread programming and in distributed systems. It allows a single thread leader to dynamically assign new tasks based on past history, instead of using a static, fixed sequence of tasks, as in the earlier example with the `do_task()` function.

The problem lies in the fact that the `if (condition) { set_variable; }` block is not atomic and can be interrupted, causing two threads to satisfy the condition simultaneously.

7.1 Pseudocode Example

This pseudocode attempts to ensure that if the other thread hasn't claimed leadership, the current thread will:

```
bool A_is_leader = 0;
bool B_is_leader = 0;
```

Thread A Code	Thread B Code
`if ( ! B_is_leader ) {`	`if ( ! A_is_leader ) {`
`A_is_leader = 1;`	`B_is_leader = 1;`
`};`	`};`

```
assert( ! (A_is_leader && B_is_leader) );
```

Outcome without a Mutex:

The race condition allows the assertion to fail because Thread A can read `B_is_leader` as `0` and Thread B can read `A_is_leader` as `0` before either thread writes its new value. This leads to the final state: `A_is_leader = 1` and `B_is_leader = 1` (two leaders). This shows that any **read-test-write sequence** on shared data must be wrapped in a mutex to form a single, atomic critical section.