Concurrent programming II

Raul P. Pelaez

January 13, 2026

Contents

1 Introduction

Thus far, the kind of concurrent code we have written has consisted on multiple threads executing independent tasks. However, in many occasions threads need to share data and cooperate to achieve a common goal. This introduces new challenges, as we need to ensure that the shared data is accessed in a controlled manner to avoid inconsistencies and unexpected behaviors.

For instance, what would happen if two threads try to increment the same counter variable at the same time? If both threads read the current value of the counter simultaneously, increment it, and then write back the new value, one of the increments will be lost. This is known as a race condition.

Advice

Debugging concurrent code
Concurrent code is non-deterministic by nature. We can issue threads and ensure the y finish (`join`) at specific points, but things like the order in which threads are scheduled, or the timing of their execution, is out of our control. This makes debugging concurrent code particularly challenging, as bugs may not manifest consistently. Moreover, issues related to the interactions between threads can often be intermittent and hard to reproduce (like disappearing when debugging is enabled).
It is thus crucial to adopt a defensive programming mindset when writing concurrent code, anticipating potential issues and implementing safeguards to mitigate them. Add tones of error handling!
Other than that, tools like thread sanitizers (e.g., `-fsanitize=thread` in GCC and Clang) can help detect data, and race conditions during runtime. We also have specialized debuggers and profilers that can help analyze thread interactions and performance. In my experience though, if you have to fire up a debugger to find a concurrency bug, you already lost.

Today, we will cover some common techniques to manage interactions between threads in concurrent programs.

2 Criticality

Info

Critical section
A portion of code that must be executed atomically, i.e. by only one thread at a time.

When dealing with shared data, we need to identify critical sections of code that access or modify the shared data. For instance, see the following code, in which several threads increment a shared counter:

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

constexpr int num_threads = 4;
constexpr int increments_per_thread = 10000;
void* increment_counter(void* arg) {
    int *counter = (int*)arg;
    for (int i = 0; i < increments_per_thread; ++i) {
        (*counter)++;
    }
    return NULL;
}

int main() {
    pthread_t threads[num_threads];
    int counter = 0;

    for (int i = 0; i < num_threads; ++i) {
        pthread_create(&threads[i], NULL, increment_counter, &counter);
    }
    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], NULL);
    }
    printf("Final counter value (expected %d): %d\n",
           num_threads * increments_per_thread, counter);
    return 0;
}
```

Output

```
Final counter value (expected 40000): 21330
```

Every time this code runs the final counter value may be different, and usually lower than expected. This is because multiple threads are accessing and modifying the shared counter variable simultaneously, leading to race conditions. To prevent this, we need to ensure that only one thread can access the critical section of code that increments the counter at any given time.

Info

Race condition
A situation in which the behavior of a program depends on the relative timing of events, such as the order in which threads are scheduled. This can lead to unpredictable and erroneous outcomes when multiple threads access shared data concurrently.

We can solve this kind of problem in a number of ways. Let us cover the most common ones.

2.1 Atomic operations

When the shared data being accessed is a simple variable (like an integer counter), we can use atomic operations to ensure that the read-modify-write sequence is performed as a single, indivisible operation.

This prevents other threads from intervening during the operation, thus avoiding race conditions.

Info

Atomic operation
An operation that is completed in a single step relative to other threads.

In C11, we can use the `stdatomic.h` library to define atomic variables and perform atomic operations on them (more here). Here is how we can modify the previous example to use atomic operations:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdatomic.h>
constexpr int num_threads = 4;
constexpr int increments_per_thread = 10000;
void* increment_counter(void* arg) {
    _Atomic int* counter = arg;
    for (int i = 0; i < increments_per_thread; ++i) {
        (*counter)++;
    }
    return NULL;
}

  int main() {
      pthread_t threads[num_threads];
      _Atomic int counter = 0;

      for (int i = 0; i < num_threads; ++i) {
          pthread_create(&threads[i], NULL, increment_counter, &counter);
      }
      for (int i = 0; i < num_threads; ++i) {
          pthread_join(threads[i], NULL);
      }
      printf("Final counter value (expected %d): %d\n",
           num_threads * increments_per_thread, counter);
      return 0;
  }
```

Output

```
Final counter value (expected 40000): 40000
```

The only change in this example is that we marked the `counter` variable as atomic using the `_Atomic` qualifier (as well as the pointer in the thread function). The increment operation (`++`) is now atomic, ensuring that each increment is completed without interference from other threads. As a result, the final counter value will always be as expected.

Advice

Typedefs in `stdatomic.h`
This header comes with several convenient aliases to define atomic versions of standard types, like `atomic_int`, `atomic_long`, `atomic_bool`, etc. You can use these typedefs instead of the `_Atomic` qualifier if you prefer.

Advanced

Advanced atomic operations
In addition to basic atomic types and operations, the `stdatomic.h` library provides more advanced features like atomic flags, memory orderings, and atomic compare-and-swap operations. These can be useful for implementing more complex synchronization mechanisms and lock-free data structures. However, they also require a deeper understanding of concurrency and memory models, so they should be used with caution.
In the example above, the expression `(*counter)++` is equivalent to `atomic_fetch_add_explicit(counter, 1, memory_order_relaxed)`, which atomically adds 1 to the value pointed to by `counter` and returns the previous value. You can explore other atomic functions in the `stdatomic.h` documentation for more advanced use cases.

2.2 Mutexes

Often the shared data, or the operation being performed on it, is more complex than a simple atomic operation can handle. In such cases, we can use mutexes (mutual exclusion locks) to protect critical sections of code.

Info

Mutex
A synchronization primitive that provides mutual exclusion, allowing only one thread to access a critical section of code at a time.
In simpler terms, a mutex is like a lock that has a single key. Only one thread can hold the key (lock the mutex) at a time, preventing other threads from entering the critical section until the key is released (the mutex is unlocked).

Say that our operation cannot be expressed as simple operation on a single (atomic) variable. For instance, say that at some point our threads need to append some data to a file, calling the following function:

```
// In append_to_file.h
void append_to_file(const char* filename, const char* data) {
    FILE* file = fopen(filename, "a");
    if (file != NULL) {
        fprintf(file, "---- new entry ----\n");
        fprintf(file, "%s\n", data);
        fprintf(file, "---- end entry ----\n");
        fclose(file);
    }
}
```

If multiple threads call this function simultaneously, the contents of the file may become corrupted, as the write operations from different threads may interleave in unpredictable ways. To prevent this, we can use a mutex to ensure that only one thread can execute the critical section of code that writes to the file at any given time:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include "append_to_file.h"

constexpr int num_threads = 2;
pthread_mutex_t file_mutex;

void* thread_function(void* arg) {
  const char* data = "Some arbitrary data";
  pthread_mutex_lock(&file_mutex);
  append_to_file("shared_file.txt", data);
```

```
  pthread_mutex_unlock(&file_mutex);
  return NULL;
}
int main() {
  pthread_t threads[num_threads];
  pthread_mutex_init(&file_mutex, NULL);
  for (int i = 0; i < num_threads; ++i) {
    pthread_create(&threads[i], NULL, thread_function, NULL);
  }
  for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
  }
  pthread_mutex_destroy(&file_mutex);
  // Print the contents of the file
  FILE* file = fopen("shared_file.txt", "r");
  if (file != NULL) {
      char ch;
      while ((ch = fgetc(file)) != EOF) {
          putchar(ch);
      }
      fclose(file);
  }
  return 0;
}
```

Output

```
---- new entry ----
Some arbitrary data
---- end entry ----
---- new entry ----
Some arbitrary data
---- end entry ----
```

In this example, we define a mutex variable `file_mutex` and initialize it in the main thread using `pthread_mutex_init`. In the thread function, we lock the mutex before calling `append_to_file` and unlock it afterward. As with any resource, we destroy the mutex at the end of the program using `pthread_mutex_destroy`.

The code in between `pthread_mutex_lock` and `pthread_mutex_unlock` is the critical section that accesses the shared resource (the file). Only one thread can hold the lock on the mutex at a time, ensuring that the file operations are performed atomically and preventing data corruption.

Warning

Mutex deadlock
The main culprit when using mutexes is deadlock, which occurs when two or more threads are waiting indefinitely for each other to release locks. This can happen if a thread locks a mutex and then tries to lock another mutex that is already held by another thread, which in turn is waiting for the first mutex to be released.

A semaphore is another synchronization primitive that can be used to control access to a shared resource. Unlike a mutex, which allows only one thread to access the critical section at a time, a semaphore maintains a count that allows multiple threads to access the resource concurrently, up to a specified limit.

Advanced

Semaphore
A synchronization primitive that maintains a count to control access to a shared resource, allowing multiple threads to access the resource concurrently up to a specified limit. If the count reaches zero, threads attempting to access the resource will block until the count is incremented (i.e., when another thread releases the resource).
POSIX semaphores can be used in C/C++ programs by including the `semaphore.h` header and using functions like `sem_init`, `sem_wait`, `sem_post`, and `sem_destroy`. However, support for POSIX semaphores may vary across platforms, so be sure to check your system's documentation. Instead, we can use mutexes and condition variables to implement similar functionality, or explore other synchronization mechanisms provided by the threading library you are using.

Threads may have complex interdependencies, requiring them to wait for each other to reach certain points in their execution before proceeding. We need some mechanism to be able to affect the execution order of threads, ensuring that certain operations are completed before others begin. This is known as synchronization.

2.3 Condition variables

A condition variable is a synchronization primitive that allows threads to wait for certain conditions to be met before proceeding. Condition variables are typically used in conjunction with mutexes to protect shared data and ensure that threads can safely wait for and signal changes in the state of the shared data.

Info

Condition variable
A synchronization primitive that allows threads to wait for certain conditions to be met before proceeding.

In POSIX threads, we can use the `pthread_cond_t` type and associated functions to implement condition variables. The main functions are:

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Here is an example:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
constexpr int num_threads = 4;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
int shared_data = 0;
void* thread_function(void* arg) {
  // This function ensures that threads execute in order based on their thread ID
  size_t thread_id = (size_t)arg; // Convert void* to size_t
  pthread_mutex_lock(&mutex);
  while (shared_data < thread_id) {
    pthread_cond_wait(&cond_var, &mutex);
  }
  printf("Thread %zu: shared_data = %d\n", thread_id, shared_data);
  shared_data++;
  pthread_cond_broadcast(&cond_var);
```

```
  pthread_mutex_unlock(&mutex);
  return NULL;
}
int main() {
  pthread_t threads[num_threads];
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&cond_var, NULL);
  for (int i = 0; i < num_threads; ++i) {
    // Store the thread ID as a size_t casted to void*
    pthread_create(&threads[i], NULL, thread_function, (void*)(size_t)i);
  }
  for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
  }
  pthread_mutex_destroy(&mutex);
  pthread_cond_destroy(&cond_var);
  return 0;
}
```

Output

```
Thread 0: shared_data = 0
Thread 1: shared_data = 1
Thread 2: shared_data = 2
Thread 3: shared_data = 3
```

Note how the output is sequential, despite the threads being started simultaneously. All threads will execute `thread_function` and try to lock the mutex. Only one of them will succeed and follow through to the `while` loop. The first thread (thread 0) will see that `shared_data` (0) is equal to its `thread_id` (0), so it will be able to continue and print the value. But say that thread 1 is the one that acquires the `mutex` first. It will see that `shared_data` (0) is less than its `thread_id` (1), so it will call `pthread_cond_wait`. This call does two things:

- It releases the `mutex`, allowing other threads to acquire it.
- It puts the thread to sleep, waiting for a signal on the condition variable `cond_var`. When the thread wakes up, it will re-acquire the `mutex` before returning from `pthread_cond_wait`.

That means that all threads except for the thread (0) will go to sleep, waiting for the condition variable to be signaled. When thread 0 increments `shared_data` and calls `pthread_cond_broadcast`, all waiting threads are woken up in an undefined order. Each thread will re-acquire the `mutex` one by one, check the condition in the `while` loop, and proceed if the condition is met.

Say that thread 2 wakes up next after thread 0. It will see that `shared_data` (1) is still less than its `thread_id` (2), so it will go back to sleep (by calling `pthread_cond_wait` again). Eventually thread 1 will acquire the mutex and wake up, see that the condition is met, print the value, increment `shared_data`, and broadcast the condition variable again. This process continues until all threads have executed in order.

The `pthreads_cond_broadcast` function wakes up all threads waiting on the condition variable, while `pthread_cond_signal` wakes up only one waiting thread. In this example, we use `pthread_cond_broadcast` to ensure that all threads are woken up whenever `shared_data` is incremented, allowing them to re-check the condition and proceed if it is met. Sometimes, using `pthread_cond_signal` may be more efficient if only one thread needs to be woken up at a time.

Info

Barrier
A synchronization primitive that allows multiple threads to wait until all of them have reached a certain point in their execution before any of them can proceed.
A barrier is a synchronization primitive that allows multiple threads to wait until all of them have reached a certain point in their execution before any of them can proceed.

Advanced

POSIX Barriers

In POSIX threads, we can use the `pthread_barrier_t` type and associated functions to implement barriers. Unfortunately, similar to semaphores, support for POSIX barriers may vary across platforms, so be sure to check your system's documentation. In order to maximize portability, we can implement our own barrier using mutexes and condition variables.

3 Exercises

3.1 Semaphore

Goal

The semaphore header is unreliable, as it is not supported on all platforms. Implement a counting semaphore using mutexes and condition variables. You should provide the following interface:

```c
// In semx.h
#pragma once
#include <pthread.h>
typedef struct {
    // Your semaphore implementation here
} semx_t;
/**
 * @brief Initialize the semaphore with the given initial count.
 * @param sem Pointer to the semaphore to initialize.
 * @param initial_count The initial count for the semaphore.
 * @return 0 on success, non-zero on failure.
 */
int semx_init(semaphore_t* sem, int initial_count);
/**
 * @brief Decrement (wait) the semaphore.
 *        If the count is zero, block until it becomes positive.
 * @param sem Pointer to the semaphore to wait on.
 * @return 0 on success, non-zero on failure.
 */
int semx_wait(semaphore_t* sem);
/**
 * @brief Increment (post) the semaphore, potentially unblocking a waiting thread.
 * @param sem Pointer to the semaphore to post.
 * @return 0 on success, non-zero on failure.
 */
int semx_post(semaphore_t* sem);
/**
 * @brief Destroy the semaphore, freeing any associated resources.
 * @param sem Pointer to the semaphore to destroy.
 * @return 0 on success, non-zero on failure.
 */
int semx_destroy(semaphore_t* sem);
```

Go to the next milestone to see a test program you can use to validate your implementation.

hint

- You will need to use a mutex to protect access to the semaphore's internal state.
- A condition variable will be useful to block and wake up threads waiting on the semaphore.
- Keep track of the current count of the semaphore, and use it to determine when threads should block or proceed.

Milestone

Here is a test program that creates multiple threads which use your semaphore implementation to control access to a shared resource. Each thread will attempt to acquire the semaphore, perform some work (simulated with a sleep), and then release the semaphore.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdlib.h>
#include "semx.h"

constexpr int num_threads = 12;
constexpr int semaphore_limit = 3;
semx_t semaphore;

void* thread_function(void* arg) {
  atomic_int* counter = arg;
  semx_wait(&semaphore);
  (*counter)++;
  assert(*counter <= semaphore_limit);
  usleep(10000); // Simulate work
  (*counter)--;
  semx_post(&semaphore);
  return NULL;
}

int main() {
  pthread_t threads[num_threads];
  semx_init(&semaphore, semaphore_limit);
  atomic_int counter = 0;
  for (int i = 0; i < num_threads; ++i) {
    pthread_create(&threads[i], NULL, thread_function, &counter);
  }
  for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
  }
  semx_destroy(&semaphore);
  return 0;
}
```

Without the semaphore, all threads could increment the counter simultaneously, triggering the assertion failure. The semaphore limits the number of threads that can access the critical section at the same time, preventing the assertion from failing.

3.2 Barrier

Goal

Implement a barrier synchronization primitive using mutexes and condition variables. Your barrier should provide the following interface:

```c
// In barrier.h
#pragma once
#include <pthread.h>
typedef struct {
    // Your barrier implementation here
} barrier_t;
/**
* @brief Initialize the barrier for the given number of threads.
* @param barrier Pointer to the barrier to initialize.
* @param count The number of threads that must call wait before any can proceed.
* @return 0 on success, non-zero on failure.
*/
int barrier_init(barrier_t* barrier, int count);
/**
* @brief Wait at the barrier until the required number of threads have called wait.
* @param barrier Pointer to the barrier to wait on.
* @return 0 on success, non-zero on failure.
*/
int barrier_wait(barrier_t* barrier);
/**
* @brief Destroy the barrier, freeing any associated resources.
* @param barrier Pointer to the barrier to destroy.
* @return 0 on success, non-zero on failure.
*/
int barrier_destroy(barrier_t* barrier);
```

Go to the next milestone to see a test program you can use to validate your implementation.

hint

- Use a mutex to protect access to the barrier's internal state.
- A condition variable will be useful to block and wake up threads waiting at the barrier.
- Keep track of the number of threads that have reached the barrier, and use this count to determine when to release all waiting threads.

Milestone

Here is a test program that creates multiple threads which use your barrier implementation to synchronize their progress. Each thread will perform some work (simulated with a sleep), wait at the barrier, and then continue execution.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include "barrier.h"
constexpr int num_threads = 5;
barrier_t barrier;
void* thread_function(void* arg) {
  int thread_id = (int)(size_t)arg;
  printf("Thread %d: Performing work before barrier\n", thread_id);
  usleep(rand() % 100000); // Simulate work
  printf("Thread %d: Waiting at barrier\n", thread_id);
  barrier_wait(&barrier);
  printf("Thread %d: Passed the barrier\n", thread_id);
  return NULL;
}
int main() {
  pthread_t threads[num_threads];
  barrier_init(&barrier, num_threads);
  for (int i = 0; i < num_threads; ++i) {
    pthread_create(&threads[i], NULL, thread_function, (void*)(size_t)i);
  }
  for (int i = 0; i < num_threads; ++i) {
    pthread_join(threads[i], NULL);
  }
  barrier_destroy(&barrier);
  return 0;
}
```

In this test program, each thread performs some work before reaching the barrier. Once all threads have called `barrier_wait`, they will all be released to continue execution. The output should show that all threads wait at the barrier before any of them proceed past it.