Concurrent programming I

Raul P. Pelaez

October 22, 2025

Contents

1 Introduction

Up until now our programs have consisted on a sequence of instructions that are executed one after the other (sequentially). Something like this:

```
int main(){
  do_this();
  do_that();
  return 0;
}
```

Modern computers have in the order of 10 cores, meaning that we can in theory perform many tasks at the same time (concurrently). Today, we will explore how to write programs that can take advantage of this.

Info

Concurrency
The ability to perform multiple tasks at the same time.

Info

Concurrency vs. Parallelism
Concurrency is the ability to *handle* multiple tasks at once, while parallelism is the ability to *execute* multiple tasks simultaneously. Concurrency is a broader concept that encompasses parallelism, but not all concurrent tasks are executed in parallel.
For instance, despite the personal computers of the era having a single CPU-code, the first versions of Windows had "multitasking" as a selling point. One could have multiple applications open at the same time (something unheard of until that point), say a text editor and a web browser. Even though the CPU could only execute one instruction at a time, the operative system would process a few instructions from the text editor, then switch to the web browser for a few instructions, and so on. This way, the user could interact with both applications seemingly at the same time. Those applications were running concurrently, but not in parallel.

Advice

Thread scheduling
Keep in mind that, when we express concurrent tasks in code, we are not necessarily implying that they will be executed in parallel. The operative system is in charge of scheduling the execution of the different tasks, and it may decide to run them sequentially if it deems it necessary. This means that, for instance, we can create an arbitrary number of tasks (threads) in our program, even if this number surpasses the number of CPU cores available.
In essence, our logic will express concurrency, i.e. tasks that can be executed independently.

Insight

It might be hard to grasp the difference between concurrency and parallelism at first. A good mental model is to think of concurrency as a set of tasks that can be executed independently, while parallelism is the actual execution of those tasks at the same time.
Some tasks are "concurrent" in nature without necessarily being parallel, for instance, downloading multiple files from the internet, or handling the GUI of a program while performing background computations. Other tasks are "parallel" in nature, for instance, rendering a 3D scene using the GPU, or performing vectorized operations on large datasets.

Before we move on, let me give you an example of two tasks that can be executed concurrently, encapsulated as functions in C:

```c
void task_a(int* a_list, int size){
  for(int i=0; i < size; i++){
    a_list[i] = a_list[i] * 2;
  }
}

void task_b(){
  printf("The current time is: %ld\n", time(NULL));
}
```

The execution of `task_a` does not depend on the execution of `task_b`, and viceversa. Therefore, we can say that these two tasks are concurrent, and we can execute them in parallel if we have the resources to do so.

On the other hand, some tasks depended on each other, for instance:

```c
void task_a(){
  FILE* f = fopen("data.txt", "r");
  int value;
  fscanf(f, "%d", &value);
  fclose(f);
}

void task_b(){
  FILE* f = fopen("data.txt", "a");
  fprintf(f, "%d\n", 42);
  fclose(f);
}
```

Here, both `task_a` and `task_b` depend on the same file, and clearly have an implicit order of execution in which `task_b` should be executed before `task_a`. Therefore, these two tasks are not concurrent, and cannot be executed in parallel without risking data corruption or other issues.

Advanced

Concurrent $\neq$ independent
It's important to note that just because two tasks are concurrent, it does not mean that they are completely independent. There may be shared resources or data that need to be managed carefully to avoid conflicts or inconsistencies. For instance, if both `task_a` and `task_b` were to write to the same array at some point, we would need to ensure that they do not overwrite each other's data. This can be achieved using synchronization mechanisms such as mutexes, semaphores, or condition variables, which we will explore later in this course.

2 Threads: The basic unit of concurrency

In C, a thread is a sequence of instructions that can be executed independently of other threads.

Info

Thread
A thread is the smallest unit of execution within a process. A process can contain multiple threads, each of which can execute independently. Threads within the same process share the same memory space, which allows them to communicate and share data easily.

Info

Program -> Process -> Thread
A program is a set of instructions that can be executed by a computer. When a program is executed, it becomes a process, which is an instance of the program that is being executed. This process is assigned a thread (or multiple), which will run/execute the instructions for that process.

There are two main ways to create threads in C: using POSIX threads (pthreads) or using the C11 standard `threads` library. In this section, we will focus on POSIX threads, as they are widely used and provide a good foundation for understanding threading in C. The C11 `threads` library is a (bit) higher-level abstraction that builds on top of pthreads. Quite unfortunately, despite being part of the C11 standard, it is not as widely supported as pthreads, so in practice, you will find yourself using pthreads more often. For instance, the threads library is not available in MacOS.

Both APIs are really similar in terms of functionality, and even share similar function names, so learning one will make it easy to learn the other. I will teach you pthreads here, and you can easily translate the knowledge to the C11 `threads` library if needed.

2.1 POSIX threads: Basic API

The essence of concurrency in C consists of associating a *thread* with a function that will be executed independently of the main program flow. We can start the execution of a thread and also block the main program flow until the thread has finished its execution (called joining).

Info

Joining a thread
Joining a thread is the process of waiting for a thread to finish its execution before continuing with the main program flow.

There are two APIs to create threads in C: POSIX threads (pthreads) and the C11 standard `threads` library. We will use the latter, as it is part of the C standard library and is therefore more portable. POSIX threads are lower level, and are not available on all platforms (e.g. Windows). For that price, you get more control over the thread management, but for our purposes, the C11 `threads` library is sufficient.

The main elements to create and manage threads in C11 are:

```c
#include <pthreads.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
int pthread_join(pthread_t thread, void **value_ptr);
void pthread_exit(void *value_ptr);
```

You can check the manual for these functions using `man pthread_create` and `man pthread_join` for details. In essence, the `pthread_create` function allow us to instruct the OS to execute a given function (`start_routine`) in a separate thread (a.i. asynchronously), the last argument, `arg`, will be passed to `start_routine` as argument when it is executed. The function `pthread_create` returns immediately, and the thread starts executing in the background.

The `pthread_join` function allows us to block the main program flow until the specified thread has finished its execution. The second argument, `value_ptr`, is a pointer to a pointer that will be used to store the return value of the thread (if any).

The `thread` argument is an opaque type (`pthread_t`) that uniquely identifies a thread after it has been created. The `attr` argument allows us to specify attributes for the thread, such as its stack size or scheduling policy. For now, we will pass `NULL` to this argument, which will use the default attributes.

Finally, the `pthread_exit` function allows a thread to terminate its execution early and return a value to the caller (the thread that joined it). Sometimes you need this, think about it, if you call `exit()` inside a thread, the whole program will terminate, not just the thread. We can use the argument to `pthread_exit` to return a value to the caller of `pthread_join`.

Lets continue the discussion with an example. The following code creates two threads that execute two functions concurrently:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
void* task_a(void* arg){
  int seconds = *((int*) arg);
  sleep(seconds);
  int *result = malloc(sizeof(int));
  *result = seconds * 2; // An arbitrary value
  return (void*) result;
}

void* task_b(void* arg){
  int seconds = *((int*) arg);
  sleep(seconds);
  float* result = malloc(sizeof(float));
  *result = seconds * 2.0f; // An arbitrary value
  return (void*) result;
}

int main(){
  pthread_t thread1, thread2;
  int arg1 = 1;
  int arg2 = 1;

  time_t start_time = time(NULL);
  // Create threads
  pthread_create(&thread1, NULL, task_a, (void*)&arg1);
  pthread_create(&thread2, NULL, task_b, (void*)&arg2);

  // Wait for threads to finish and get their results
  void* result1;
```

```c
    void* result2;
    pthread_join(thread1, &result1);
    pthread_join(thread2, &result2);

    printf("Result from task_a: %d\n", *((int*) result1));
    printf("Result from task_b: %.2f\n", *((float*) result2));

    time_t end_time = time(NULL);
    printf("Total time taken: %ld seconds\n", end_time - start_time);
    // Free allocated memory
    free(result1);
    free(result2);
    return 0;
}
```

Output

```
Result from task_a: 2
Result from task_b: 2.00
Total time taken: 1 seconds
```

The program above will launch two threads, which will simply wait for a given number of seconds (as given by their argument), and then return some arbitrary value (that we have to `malloc`, since we can only return a single pointer from a thread). The main program will wait for both threads to finish using `pthread_join`, and then print their results. The snippet is sending `1` as argument to both threads, so they will each wait for 1 second before returning their results.

The interesting thing to note here is that the total execution time is only 1 second, even though each thread waits for 1 second. This is because both threads are executed concurrently, and therefore the total time taken is only the maximum of the two wait times (1 second in this case).

Advice

Void pointers
The void pointer is used here as a way to allow arbitrary data to be passed to and from the thread functions. By the rules of the language, we can cast any type to/from a void pointer. Threading is not the only place where this trick is employed.
As you can see in these examples, passing something to a function usually involves writing a helper `struct` that contains all the arguments you want to pass, and then passing a pointer to that struct as a void pointer. Inside the function, you cast the void pointer back to the original struct type, and then access the fields of the struct as needed. The obvious downside to this, from a software design perspective, is that you momentarily lose the information about the type of the data. This makes the function signature less informative, and by "going over" the type system, it also introduces the potential bug of changing the type of the data being passed without updating the function that receives it. Such is the life of a C programmer.
On the other hand, returning data from a thread is usually done by allocating memory for the return value using `malloc`, and then returning a pointer to that memory as a void pointer. The caller is then responsible for casting the void pointer back to the original type, and freeing the allocated memory when it is no longer needed.

Info

Pthread is a library you must link to
When compiling a C program that uses POSIX threads, you need to link against the pthread library. This is done by adding the `-lpthread` flag to the compiler command line. For instance:
`$ cc -o my_program my_program.c -lpthread`
You can also add an `add_library` directive to your `CMakeLists.txt` file if you are using CMake.

The computer I am running this examples on has 12 cores, so in principle, I can launch 12 threads that will wait for 1 second each, and the total time taken will still be 1 second. Let me write a quick program

to demonstrate this:

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

void* task(void* arg){
  // Simulate a task that takes some time
  // Takes about 2 seconds on my machine
  for (volatile long i = 0; i < 2000000000L; i++);
  return NULL;
}

int main(){
  const int num_threads = 12;
  pthread_t threads[num_threads];
  time_t start_time = time(NULL);
  // Launch all threads
  for(int i = 0; i < num_threads; i++){
    pthread_create(&threads[i], NULL, task, NULL);
  }
  // Wait for all threads to finish
  for(int i = 0; i < num_threads; i++){
    pthread_join(threads[i], NULL);
  }
  time_t end_time = time(NULL);
  printf("Total time taken for %d threads: %ld seconds\n", num_threads, end_time - start_time);
  return 0;
}
```

Output

```
Total time taken for 12 threads: 2 seconds
```

I also wanted to make explicit here the fact that you can launch the same function to a different thread multiple times, each with its own argument. In this case, we are launching the same `task` function 12 times, each time with no argument (NULL), and waiting for all of them to finish.

If I ran this program asking for, say, 24 threads, the total time taken would be around 4 seconds, since my computer has only 12 cores, and therefore only 12 threads can be executed in parallel at any given time. The operative system will schedule the execution of the threads, so that when one thread finishes, another one can start executing.

Insight

I am not using sleep here to simulate a task that takes time, because sleep does not consume CPU time. Instead, I am using a simple loop that performs a large number of iterations to simulate a CPU-bound task.

Advice

You can have more threads than cores
A thread is an abstraction that allows us to express concurrency in our programs, but it is not necessarily tied to the number of CPU cores available. This means that our programs will run fine even if we create more threads than the number of cores available. The operative system will take care of scheduling the execution of the threads, so that they can share the available CPU time.

Advice

Threads are not for high performance computing
A thread is a relatively heavy-weight abstraction, and creating and managing threads has a non-negligible overhead. So for instance, say that you have two vectors consisting of 10M elements, and you need to sum them element-wise. You could be tempted to create a thread for each element (or group of elements) and sum them in parallel. However, the overhead of creating and managing 10M threads would be much higher than the time taken to sum the vectors sequentially. POSIX threads are designed for general-purpose concurrency, things like handling multiple connections in a web server, or performing background tasks in a GUI application. For high-performance computing tasks, such as numerical simulations or data processing, it is usually better to use other abstractions, such as OpenMP or MPI, which are designed specifically for parallel computing. You will cover these technologies in the future :).

2.2 Thread utilities

Info

Which thread am I?
Each thread has a unique identifier that can be obtained using the `pthread_self()` function. This function returns a `pthread_t` value that uniquely identifies the calling thread.

```c
#include <stdio.h>
#include <pthread.h>
void* print_thread_id(void* arg){
  pthread_t thread_id = pthread_self();
  printf("Thread ID: %lu\n", (unsigned long) thread_id);
  return NULL;
}
int main(){
  pthread_t thread;
  pthread_create(&thread, NULL, print_thread_id, NULL);
  pthread_join(thread, NULL);
  return 0;
}
```

Output

```
Thread ID: 6138605568
```

Info

Detach a thread

Sometimes, we may want to create a thread that runs independently of the main program flow, and we do not need to wait for it to finish. In this case, we can detach the thread using the `pthread_detach()` function. This function allows the thread to run independently, and its resources will be automatically released when it finishes executing.

```c
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void* detached_task(void* arg){
  sleep(2);
  printf("Detached thread finished executing.\n");
  return NULL;
}
int main(){
  pthread_t thread;
  pthread_create(&thread, NULL, detached_task, NULL);
  pthread_detach(thread);
  printf("Main thread continues executing.\n");
  sleep(3); // Wait for a while to see the detached thread finish
  // No need to join the detached thread
  return 0;
}
```

Output

```
Main thread continues executing.
Detached thread finished executing.
```

3 Exercises

3.1 Parallel reduction

Goal

The following function allocates and returns a pointer to a list of random floats:

```c
#include <stdlib.h>
float* generate_random_list(int size){
  float* list = malloc(size * sizeof(float));
  for(int i = 0; i < size; i++){
    list[i] = (float) rand() / RAND_MAX; // Random float between 0 and 1
  }
  return list;
}

int main(){
  int size = 1000000;
  float* random_list = generate_random_list(size);
  // Do something with the list
  free(random_list);
  return 0;
}
```

Write a function that will compute the average of a list of floats concurrently using multiple threads. The function should take as input the list of floats, its size, and the number of threads to use. It should divide the list into equal-sized chunks, and each thread should compute the average of its chunk. Finally, the main function should combine the results from all threads to compute the overall average.
Write a single-threaded naive version and assert that your function is correct.

hint

- Accumulate the sum using `double` instead of `float` to avoid precision issues.

Insight

IRL, I would not use pthreads for such a numerical computation. Although you will probably see some gains in performance, there are other abstractions better suited for this kind of task, such as OpenMP or MPI. However, this exercise is meant to help you understand the basics of threading in C, so we will use pthreads here.

Advanced milestone

Modify the `generate_random_list` function so that it generates the random numbers concurrently using multiple threads. Each thread should generate a portion of the list.

hint

- The function `rand` is not thread-safe. Read about the `rand_r` function.
- Each thread will need to be given a unique seed for the random number generator.