Threads

In the previous chapter, we discussed threads as part of the multithreading approach that you can use when wanting to write concurrent programs in a POSIX-compliant operating system.

In this section, you will find a recap on everything you should know about threads. We will also bring in some new information that is relevant to topics we will discuss later. Remember that all of this information will act as a foundation for continuing to develop multithreaded programs.

Every thread is initiated by a process. It will then belong to that process forever. It is not possible to have a shared thread or transfer the ownership of a thread to another process. Every process has at least one thread that is its *main thread*. In a C program, the `main` function is executed as part of the main thread.

All the threads share the same **Process ID** (**PID**). If you use utilities like `top` or `htop`, you can easily see the threads are sharing the same process ID, and are grouped under it. More than that, all the attributes of the owner process are inherited by all of its threads for example, group ID, user ID, current working directory, and signal handlers. As an example, the current working directory of a thread is the same as its owner process.

Every thread has a unique and dedicated **Thread ID** (**TID**). This ID can be used to pass signals to that thread or track it while debugging. You will see that in POSIX threads, the thread ID is accessible via the `pthread_t` variable. In addition, every thread has also a dedicated signal mask that can be used to filter out the signals it may receive.

All of the threads within the same process have access to all of the *file descriptors* opened by other threads in that process. Therefore, all the threads can both read or modify the resources behind those file descriptors. This is also true regarding *socket descriptors* and opened *sockets*. In upcoming chapters, you'll learn more about file descriptors and sockets.

Threads can use all the techniques used by processes introduced in chapter 14 to share or transfer a state. Take note of the fact that having a shared state in a shared place (like a database) is different from transmitting it on a network for example, and this results in two different categories of IPC techniques. We will come back to this point in future chapters.

Here, you can find a list of methods that can be used by threads to share or transfer a state in a POSIX-compliant system:

- Owner process's memory (Data, Stack, and Heap segments). This method is *only* specific to threads and not processes.
- Filesystem.

- Memory-mapped files.
- Network (using internet sockets).
- Signal passing between threads.
- Shared memory.
- POSIX pipes.
- Unix domain sockets.
- POSIX message queues.
- Environment variables.

To proceed with the thread properties, all of the threads within the same process can use the same process's memory space to store and maintain a shared state. This is the most common way of sharing a state among a number of threads. The Heap segment of the process is usually used for this purpose.

The lifetime of a thread is dependent on the lifetime of its owner process. When a process gets *killed* or *terminated*, all the threads belonging to that process will also get terminated.

When the main thread ends, the process quits immediately. However, if there are other *detached* threads running, the process waits for all of them to be finished before getting terminated. Detached threads will be explained while explaining the thread creation in POSIX.

The process that creates a thread can be the kernel process. At the same time, it can also be a user process initiated in the user space. If the process is the kernel, the thread is called a *kernel-level thread* or simply a *kernel thread*, otherwise, the thread is called a *user-level thread*. Kernel threads typically execute important logic, and because of this they have higher priorities than that of user threads. As an example, a device driver may be using a kernel thread to wait for a hardware signal.

Similar to user threads that have access to the same memory region, kernel threads are also able to access the kernel's memory space and, subsequently, all the procedures and units within the kernel.

Throughout this book we will be mainly talking about user threads, not kernel threads. That's because the required API for working with user threads is provided by the POSIX standard. But there is no standard interface for creating and managing a kernel thread and they are only specific to each kernel.

Creating and managing kernel threads is beyond the scope of this book. Thus, from this point on, when we are using the term *thread*, we are referring to user threads and not kernel threads.

A user cannot create a thread directly. The user needs to spawn a process first, as only then can that process's main thread initiate another thread. Note that only threads can create threads.

Regarding the memory layout of threads, every thread has its own Stack memory region that can be considered as a private memory region dedicated to that thread. In practice, however, it can be accessed by other threads (within the same process) when having a pointer addressing it.

You should remember that all these Stack regions are part of the same process's memory space and can be accessed by any thread within the same process.

Regarding synchronization techniques, the same control mechanisms that are used to synchronize processes can be used to synchronize a number of threads. Semaphores, mutexes, and condition variables are part of the tools that can be used to synchronize threads, as well as processes.

When its threads are synchronized and no further data race or race condition can be observed, a program is usually referred to as a *thread-safe* program. Similarly, a library or a set of functions that can be easily used in a multithreaded program without introducing any new concurrency issue is called a *thread-safe library*. Our goal as programmers is to produce a thread safe piece of code.

Note:
In the following link, you can find more information about POSIX threads and the properties they share. The following link is about the NTPL implementation of the POSIX threading interface. This is dedicated to a Linux environment but most of it is applicable to other Unix-like operating systems.

`http://man7.org/linux/man-pages/man7/pthreads.7.html`

In this section we looked at some foundational concepts and properties concerning threads in order to better understand the upcoming sections. You will see many of these properties in action later as we talk about various multithreaded examples.

The next section will introduce you to the first code examples on how to create a POSIX thread. The section is going to be simple because it only addresses the basics of threading in POSIX. These basics will lead us into more advanced topics afterwards.

POSIX threads

This section is dedicated to the POSIX threading API, better known as the *pthread library*. This API is very important because it's the main API used for creating and managing the threads in a POSIX-compliant operating system.

In non-POSIX-compliant operating systems such as Microsoft Windows, there should be another API designed for this purpose and it can be found in the documentation of that operating system. For example, in the case of Microsoft Windows, the threading API is provided as part of the Windows API, known as the Win32 API. This is the link to Microsoft's documentation regarding Windows' threading API: `https://docs.microsoft.com/en-us/windows/desktop/procthread/process-and-thread-functions`.

However, as part of C11, we expect to have a unified API to work with threads. In other words, regardless of whether you're writing a program for a POSIX system or a non-POSIX system, you should be able to use the same API provided by C11. While this is highly desirable, not much support exists for such universal APIs among the various C standard implementations, like glibc, at this point in time.

To proceed with the topic, the pthread library is simply a set of *headers* and *functions* that can be used to write multithreaded programs in POSIX-compliant operating systems. Each operating system has its own implementation for pthread library. These implementations could be totally different from what another POSIX-compliant operating system have, but at the end of the day, they all expose the same interface (API).

One famous example is the **Native POSIX Threading Library**, or **NPTL** for short, which is the main implementation of pthread library for the Linux operating system.

As described by the pthread API, all threading functionality is available by including the header `pthread.h`. There are also some extensions to the pthread library that are only available if you include `semaphore.h`. As an example, one of the extensions involves operations that are semaphore-specific, for example, creating a semaphore, initializing it, destroying it, and so on.

The POSIX threading library exposes the following functionalities. They should be familiar to you since we have given detailed explanations to them in the previous chapters:

- Thread management, which includes thread creation, joining threads, and detaching threads
- Mutexes
- Semaphores
- Condition variables
- Various types of locks like spinlocks and recursive locks

To explain the preceding functionalities, we must start with the `pthread_` prefix. All pthread functions start with this prefix. This is true in all cases except for semaphores, which have not been part of the original POSIX threading library and have been added later as an extension. In this case, the functions will start with the `sem_` prefix.

In the following sections of this chapter, we will see how to use some of the preceding functionalities when writing a multithreaded program. To start with, we'll learn how to create a POSIX thread in order to run a code concurrent with the main thread. Here, we will learn about the `pthread_create` and `pthread_join` functions, which belong to the main API used for *creating* and *joining* threads, respectively.

Spawning POSIX threads

Having gone through all the fundamental concepts like interleaving, locks, mutexes, and condition variables, in the previous chapters, and introducing the concept of POSIX threads in this chapter, it is the time to write some code.

The first step is to create a POSIX thread. In this section, we are going to demonstrate how we can use the POSIX threading API to create new threads within a process. Following *example 15.1* describes how to create a thread that performs a simple task like printing a string to the output:

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

// The POSIX standard header for using pthread library
#include <pthread.h>

// This function contains the logic which should be run
// as the body of a separate thread
void* thread_body(void* arg) {
  printf("Hello from first thread!\n");
  return NULL;
}

int main(int argc, char** argv) {

  // The thread handler
  pthread_t thread;

  // Create a new thread
  int result = pthread_create(&thread, NULL, thread_body, NULL);
  // If the thread creation did not succeed
```

```
  if (result) {
    printf("Thread could not be created. Error number: %d\n",
           result);
    exit(1);
  }

  // Wait for the created thread to finish
  result = pthread_join(thread, NULL);
  // If joining the thread did not succeed
  if (result) {
    printf("The thread could not be joined. Error number: %d\n",
           result);
    exit(2);
  }
  return 0;
}
```

Code Box 15-1 [ExtremeC_examples_chapter15_1.c]: Spawning a new POSIX thread

The example code, seen in *Code Box 15-1*, creates a new POSIX thread. This is the first example in this book that has two threads. All previous examples were single-threaded, and the code was running within the main thread all the time.

Let's explain the code we've just looked at. At the top, we have included a new header file: `pthread.h`. This is the standard header file that exposes all the pthread functionalities. We need this header file so that we can bring in the declarations of both `pthread_create` and `pthread_join` functions.

Just before the `main` function, we have declared a new function: `thread_body`. This function follows a specific signature. It accepts a `void*` pointer and returns another `void*` pointer. As a reminder, `void*` is a generic pointer type that can represent any other pointer type, like `int*` or `double*`.

Therefore, this signature is the most general signature that a C function can have. This is imposed by the POSIX standard that all functions willing to be the *companion function* for a thread (being used as thread logic) should follow this generic signature. That's why we have defined the `thread_body` function like this.

Note:
The `main` function is a part of the main thread's logic. When the main thread is created, it executes the `main` function as part of its logic. This means that there might be other code that is executed before and after the `main` function.

Back to the code, as the first instruction in the `main` function, we have declared a variable of type `pthread_t`. This is a thread handle variable, and upon its declaration, it doesn't refer to any specific thread. In other words, this variable doesn't hold any valid thread ID yet. Only after creating a thread successfully does this variable contain a valid handle to the newly created thread.

After creating the thread, the thread handle actually refers to the thread ID of the recently created thread. While thread ID is the thread identifier in the operating system, the thread handle is the representative of the thread in the program. Most of the time, the value stored in the thread handle is the same as thread ID. Every thread is able to access its thread ID through obtaining a `pthread_t` variable that refers to itself. A thread can use the `pthread_self` function to obtain a self-referring handle. We are going to demonstrate the usage of these functions in future examples.

Thread creation happens when the `pthread_create` function is called. As you can see, we have passed the address of the `thread` handle variable to the `pthread_create` function in order to have it filled with a proper handle (or thread ID), referring to the newly created thread.

The second argument determines the thread's attributes. Every thread has some attributes like *stack size*, *stack address*, and *detach* state that can be configured before spawning the thread.

We show more examples of how to configure these attributes and how they affect the way the threads behave. If a `NULL` is passed as the second argument, it means that the new thread should use the default values for its attributes. Therefore, in the preceding code, we have created a thread that has attributes with default values.

The third argument passed to `pthread_create` is a function pointer. This is pointing to the thread's *companion function*, which contains the thread's logic. In the preceding code, the thread's logic is defined in the `thread_body` function. Therefore, its address should be passed in order to get bound to the handle variable `thread`.

The fourth and last argument is the input argument for the thread's logic, which in our case is `NULL`. This means that we don't want to pass anything to the function. Therefore, the parameter `arg` in the `thread_body` function would be `NULL` upon the thread's execution. In the examples provided in the next section, we'll look at how we can pass a value to this function instead of a `NULL`.

All pthread functions, including `pthread_create`, are supposed to return zero upon successful execution. Therefore, if any number other than zero is returned, then it means that the function has failed, and an *error number* has been returned.

Note that creating a thread using `pthread_create` doesn't mean that the thread's logic is being executed immediately. It is a matter of scheduling and cannot be predicted when the new thread gains one of the CPU cores and starts its execution.

After creating the thread, we join the newly created thread, but what exactly does that mean? As we explained before, each process starts with exactly one thread, which is the *main thread*. Except for the main thread, whose parent is the owning process, all other threads have a *parent thread*. In a default scenario, if the main thread is finished, the process will also be finished. When the process gets terminated, all other running or sleeping threads will also get terminated immediately.

So, if a new thread is created and it hasn't started yet (because it hasn't gained the use of the CPU) and in the meantime, the parent process is terminated (for whatever reason), the thread will die before even executing its first instruction. Therefore, the main thread needs to wait for the second thread to become executed and finished by joining it.

A thread becomes finished only when its companion function returns. In the preceding example, the spawned thread becomes finished when the `thread_body` companion function returns, and this happens when the function returns `NULL`. When the newly spawned thread is finished, the main thread, which was blocked behind calling `pthread_join`, is released and can continue, which eventually leads to successful termination of the program.

If the main thread didn't join the newly created thread, then it is unlikely that the newly spawned thread can be executed at all. As we've explained before, this happens due to the fact that the main thread exits even before the spawned thread has entered into its execution phase.

We should also remember that creating a thread is not enough to have it executed. It may take a while for the created thread to gain access to a CPU core, and through this eventually start running. If, in the meantime, the process gets terminated, then the newly created thread has no chance of running successfully.

Now that we've talked through the design of the code, *Shell Box 15-1* shows the output of running *example 15.1*:

```
$ gcc ExtremeC_examples_chapter15_1.c -o ex15_1.out -lpthread
$ ./ex15_1.out
Hello from first thread!
$
```

Shell Box 15-1: Building and running example 15.1

As you see in the preceding shell box, we need to add the `-lpthread` option to the compilation command. This is done because we need to link our program with the existing implementation of the pthread library. In some platforms, like macOS, your program might get linked without the `-lpthread` option as well; however, it is strongly recommended to use this option while you are linking programs that use pthread library. The importance of this advice is to make your *build scripts* working on any platform and prevent any cross-compatibility issues while building your C projects.

A thread that can be joined is known as *joinable*. The threads are joinable by default. Opposite to joinable threads, we have *detached* threads. Detached threads cannot be joined.

In *example 15.1*, the main thread could detach the newly spawned thread instead of joining it. This way, we would have let the process know that it must wait for the detached thread to become finished before it can get terminated. Note that in this case, the main thread can exit without the parent process being terminated.

In the final code of this section, we want to rewrite the preceding example using detached threads. Instead of joining the newly created thread, the main thread makes it detached and then exits. This way, the process remains running until the second thread finishes, despite the fact that the main thread has already exited:

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

// The POSIX standard header for using pthread library
#include <pthread.h>

// This function contains the logic which should be run
// as the body of a separate thread
void* thread_body(void* arg) {
  printf("Hello from first thread!\n");
  return NULL;
}

int main(int argc, char** argv) {

  // The thread handler
  pthread_t thread;

  // Create a new thread
  int result = pthread_create(&thread, NULL, thread_body, NULL);
  // If the thread creation did not succeed
  if (result) {
    printf("Thread could not be created. Error number: %d\n",
```

```
            result);
    exit(1);
  }

  // Detach the thread
  result = pthread_detach(thread);
  // If detaching the thread did not succeed
  if (result) {
    printf("Thread could not be detached. Error number: %d\n",
            result);
    exit(2);
  }

  // Exit the main thread
  pthread_exit(NULL);

  return 0;
}
```

Code Box 15-2 [ExtremeC_examples_chapter15_1_2.c]: Example 15.1 spawning a detached thread

The output of the preceding code is exactly the same as the previous code written using joinable threads. The only difference is the that way we managed the newly created thread.

Right after the creation of the new thread, the main thread has detached it. Then following that, the main thread exits. The instruction `pthread_exit(NULL)` was necessary in order to let the process know that it should wait for other detached threads to be finished. If the threads were not detached, the process would get terminated upon the exit of the main thread.

Note:
The *detach state* is one of the thread attributes that can be set before creating a new thread in order to have it detached. This is another method to create a new detached thread instead of calling `pthread_detach` on a joinable thread. The difference is that this way, the newly created thread is detached from the start.

In the next section, we're going to introduce our first example demonstrating a race condition. We will be using all the functions introduced in this section in order to write future examples. Therefore, you'll have a second chance to revisit them again in different scenarios.

Example of race condition

For our second example, we're going to look at a more problematic scenario. *Example 15.2*, shown in *Code Box 15-3*, shows just how interleavings happen and how we cannot reliably predict the final output of the example in practice, mainly because of the non-deterministic nature of concurrent systems. The example involves a program that creates three threads at almost the same time, and each of them prints a different string.

The final output of the following code contains the strings printed by three different threads but in an unpredictable order. If the invariant constraint (introduced in the previous chapter) for the following example was to see the strings in a specific order in the output, the following code would have failed at satisfying that constraint, mainly because of the unpredictable interleavings. Let's look at the following code box:

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

// The POSIX standard header for using pthread library
#include <pthread.h>

void* thread_body(void* arg) {
  char* str = (char*)arg;
  printf("%s\n", str);
  return NULL;
}

int main(int argc, char** argv) {

  // The thread handlers
  pthread_t thread1;
  pthread_t thread2;
  pthread_t thread3;

  // Create new threads
  int result1 = pthread_create(&thread1, NULL,
          thread_body, "Apple");
  int result2 = pthread_create(&thread2, NULL,
          thread_body, "Orange");
  int result3 = pthread_create(&thread3, NULL,
          thread_body, "Lemon");

  if (result1 || result2 || result3) {
    printf("The threads could not be created.\n");
    exit(1);
  }

  // Wait for the threads to finish
```

```
  result1 = pthread_join(thread1, NULL);
  result2 = pthread_join(thread2, NULL);
  result3 = pthread_join(thread3, NULL);

  if (result1 || result2 || result3) {
    printf("The threads could not be joined.\n");
    exit(2);
  }
  return 0;
}
```

Code Box 15-3 [ExtremeC_examples_chapter15_2.c]: Example 15.2 printing three different strings to the output

The code we've just looked at is very similar to the code written for *example 15.1*, but it creates three threads instead of the one. In this example, we use the same companion function for all three threads.

As you can see in the preceding code, we have passed a fourth argument to the `pthread_create` function, whereas in our previous example, *15.1*, it was `NULL`. These arguments will be accessible by the thread through the generic pointer parameter `arg` in the `thread_body` companion function.

Inside the `thread_body` function, the thread casts the generic pointer `arg` to a `char*` pointer and prints the string starting at that address using the `printf` function. This is how we are able to pass arguments to the threads. Likewise, it doesn't matter how big they are since we are only passing a pointer.

If you have multiple values that need to be sent to a thread upon their creation, you could use a structure to contain those values and pass a pointer to a structure variable filled by the desired values. We will demonstrate to how to do this in the next chapter, *Thread Synchronization*.

Note:
The fact that we can pass a pointer to a thread implies that the new threads should have access to the same memory region that the main thread has access to. However, access is not limited to a specific segment or region in the owning process's memory and all threads have full access to the Stack, Heap, Text, and Data segments in a process.

If you take *example 15.2* and run it several times, you'll see that the order of the printed strings can vary, as each run is expected to print the same strings but in a different order.

Shell Box 15-2 shows the compilation and the output of *example 15.2* after three consecutive runs:

```
$ gcc ExtremeC_examples_chapter15_2.c -o ex15_2.out -lpthread
$ ./ex15_2.out
Apple
Orange
Lemon
$ ./ex15_2.out
Orange
Apple
Lemon
$ ./ex15_2.out
Apple
Orange
Lemon
$
```

Shell Box 15-2: Running example 15.2 three times to observe the existing race condition and various interleavings

It is easy to produce the interleavings in which the first and second threads print their strings before the third thread, but it would be difficult to produce an interleaving in which the third thread prints its string, `Lemon`, as the first or second string in the output. However, this will certainly happen, albeit with a low probability. You might need to run the example many more times in order to produce that interleaving. This may require some patience.

The preceding code is also said to not be thread safe. This is an important definition; a multithreaded program is thread safe if, and only if, it has no race condition according to the defined invariant constraints. Therefore, since the preceding code has a race condition, it is not thread safe. Our job would be to make the preceding code thread safe through the use of proper control mechanisms that will be introduced in the next chapter.

As you see in the output of the preceding example, we don't have any interleaving between the characters of `Apple` or `Orange`. For example, we don't see the following output:

```
$ ./ex15_2.out
AppOrle
Ange
```

```
Lemon
$
```

Shell Box 15-3: An imaginary output that does not happen for the above example

This shows a fact that the `printf` function is *thread safe* and it simply means that it doesn't matter how interleavings happen, when one of the threads is in the middle of printing a string, `printf` instances in other threads don't print anything.

In addition, in the preceding code, the `thread_body` companion function was run three times in the context of three different threads. In the previous chapters and before giving multithreaded examples, all functions were being executed in the context of the main thread. From now on, every function call occurs in the context of a specific thread (not necessarily the main thread).

It's not possible for two threads to initiate a single function call. The reason is obvious because each function call needs to create a *stack frame* that should be put on top of the Stack of just one thread, and two different threads have two different Stack regions. Therefore, a function call can only be initiated by just one thread. In other words, two threads can call the same function separately and it results into two separate function calls, but they cannot share the same function call.

We should note that the pointer passed to a thread should not be a *dangling pointer*. It causes some serious memory issues that are hard to track. As a reminder, a dangling pointer points to an address in the memory where there is no allocated variable. More specifically, this is the case that at some moment in time; there might have been a variable or an array there originally, but as of the time when the pointer is about to be used, it's already been freed.

In the preceding code, we passed three literals to each thread. Since the memory required for these string literals are allocated from the Data segment and not from Heap or Stack segments, their addresses never become freed and the `arg` pointers won't become dangling.

It would be easy to write the preceding code in a way in which the pointers become dangling. The following is the same code but with dangling pointers, and you will see shortly that it leads to bad memory behaviors:

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

// The POSIX standard header for using pthread library
#include <pthread.h>
```

```
void* thread_body(void* arg) {
  char* str = (char*)arg;
  printf("%s\n", str);
  return NULL;
}

int main(int argc, char** argv) {

  // The thread handlers
  pthread_t thread1;
  pthread_t thread2;
  pthread_t thread3;

  char str1[8], str2[8], str3[8];
  strcpy(str1, "Apple");
  strcpy(str2, "Orange");
  strcpy(str3, "Lemon");

  // Create new threads
  int result1 = pthread_create(&thread1, NULL, thread_body, str1);
  int result2 = pthread_create(&thread2, NULL, thread_body, str2);
  int result3 = pthread_create(&thread3, NULL, thread_body, str3);

  if (result1 || result2 || result3) {
    printf("The threads could not be created.\n");
    exit(1);
  }

  // Detach the threads
  result1 = pthread_detach(thread1);
  result2 = pthread_detach(thread2);
  result3 = pthread_detach(thread3);

  if (result1 || result2 || result3) {
    printf("The threads could not be detached.\n");
    exit(2);
  }

  // Now, the strings become deallocated.
  pthread_exit(NULL);

  return 0;
}
```

Code Box 15-4 [ExtremeC_examples_chapter15_2_1.c]: Example 15.2 with literals allocated from the main thread's Stack region

The preceding code is almost the same as the code given in *example 15.2*, but with two differences.

Firstly, the pointers passed to the threads are not pointing to the string literals residing in Data segment, instead they point to character arrays allocated from the main thread's Stack region. As part of the `main` function, these arrays have been declared and in the following lines, they have been populated by some string literals.

We need to remember that the string literals still reside in the Data segment, but the declared arrays now have the same values as the string literals after being populated using the `strcpy` function.

The second difference is regarding how the main thread behaves. In the previous code it joined the threads, but in this code, it detaches the threads and exits immediately. This will deallocate the arrays declared on top of the main thread's Stack, and in some interleavings other threads may try to read those freed regions. Therefore, in some interleavings, the pointers passed to the threads can become dangling.

Note:
Some constraints, like having no crashes, having no dangling pointers, and generally having no memory-related issues, can always be thought of as being part of the invariant constraints for a program. Therefore, a concurrent system that yields a dangling pointer issue in some interleavings is definitely suffering from a serious race condition.

To be able to detect the dangling pointers, you need to use a *memory profiler*. As a simpler approach, you could run the program several times and wait for a crash to happen. However, you are not always fortunate enough to be able to see that and we are not lucky to see crashes in this example either.

To detect bad memory behavior in this example, we are going to use `valgrind`. You remember that we introduced this memory profiler in *Chapter 4*, *Process Memory Structure*, and *Chapter 5*, *Stack and Heap*, for finding the *memory leaks*. Back in this example, we want to use it to find the places where bad memory access has happened.

It's worth remembering that using a dangling pointer, and accessing its content, will not necessarily lead to a crash. This is especially true in the preceding code, in which the strings are placed on top of the main thread's Stack.

While the other threads are running, the Stack segment remains the same as it was when the main thread exited, therefore you can access the strings even though the `str1`, `str2`, and `str3` arrays are deallocated while leaving the `main` function. In other words, in C or C++, the runtime environment does not check if a pointer is dangling or not, it just follows the sequence of statements.

If a pointer is dangling and its underlying memory is changed, then bad things like crash or logical errors can happen but as long as the underlying memory is *untouched* then using the dangling pointers may not lead to a crash, and this is very dangerous and hard to track.

In short, just because you can access a memory region through a dangling pointer, that doesn't mean that you are allowed to access that region. This is the reason why we need to use a memory profiler like `valgrind` that will report on these invalid memory accesses.

In the following shell box, we compile the program and we run it with `valgrind` twice. In the first run, nothing bad happens but in the second run, `valgrind` reports a bad memory access.

Shell Box 15-4 shows the first run:

```
$ gcc -g ExtremeC_examples_chapter15_2_1.c -o ex15_2_1.out -lpthread
$ valgrind ./ex15_2_1.out
==1842== Memcheck, a memory error detector
==1842== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward
et al.
==1842== Using Valgrind-3.13.0 and LibVEX; rerun with -h for
copyright info
==1842== Command: ./ex15_2_1.out
==1842==
Orange
Apple
Lemon
==1842==
==1842== HEAP SUMMARY:
==1842==     in use at exit: 0 bytes in 0 blocks
==1842==   total heap usage: 9 allocs, 9 frees, 3,534 bytes
allocated
==1842==
==1842== All heap blocks were freed -- no leaks are possible
==1842==
==1842== For counts of detected and suppressed errors, rerun with:
-v
==1842== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0
from 0)
$
```

Shell Box 15-4: Running example 15.2 with valgrind for the first time

In the second run, `valgrind` reports some memory access issues (note that the full output will be viewable when you run it, but for purpose of length, we've refined it.):

```
$ valgrind ./ex15_2_1.out
==1854== Memcheck, a memory error detector
==1854== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward
et al.
==1854== Using Valgrind-3.13.0 and LibVEX; rerun with -h for
copyright info
==1854== Command: ./ex15_2_1.out
==1854==
Apple
Lemon
==1854== Thread 4:
==1854== Conditional jump or move depends on uninitialised
value(s)
==1854==    at 0x50E6A65: _IO_file_xsputn@@GLIBC_2.2.5
(fileops.c:1241)
==1854==    by 0x50DBA8E: puts (ioputs.c:40)
==1854==    by 0x1087C9: thread_body (ExtremeC_examples_
chapter15_2_1.c:17)
==1854==    by 0x4E436DA: start_thread (pthread_create.c:463)
==1854==    by 0x517C88E: clone (clone.S:95)
==1854==
...
==1854==
==1854== Syscall param write(buf) points to uninitialised byte(s)
==1854==    at 0x516B187: write (write.c:27)
==1854==    by 0x50E61BC: _IO_file_write@@GLIBC_2.2.5
(fileops.c:1203)
==1854==    by 0x50E7F50: new_do_write (fileops.c:457)
==1854==    by 0x50E7F50: _IO_do_write@@GLIBC_2.2.5
(fileops.c:433)
==1854==    by 0x50E8402: _IO_file_overflow@@GLIBC_2.2.5
(fileops.c:798)
==1854==    by 0x50DBB61: puts (ioputs.c:41)
==1854==    by 0x1087C9: thread_body (ExtremeC_examples_
chapter15_2_1.c:17)
==1854==    by 0x4E436DA: start_thread (pthread_create.c:463)
==1854==    by 0x517C88E: clone (clone.S:95)
```

```
...
==1854==
Orange
==1854==
==1854== HEAP SUMMARY:
==1854==     in use at exit: 272 bytes in 1 blocks
==1854==   total heap usage: 9 allocs, 8 frees, 3,534 bytes
allocated
==1854==
==1854== LEAK SUMMARY:
==1854==    definitely lost: 0 bytes in 0 blocks
==1854==    indirectly lost: 0 bytes in 0 blocks
==1854==      possibly lost: 272 bytes in 1 blocks
==1854==    still reachable: 0 bytes in 0 blocks
==1854==         suppressed: 0 bytes in 0 blocks
==1854== Rerun with --leak-check=full to see details of leaked
memory
==1854==
==1854== For counts of detected and suppressed errors, rerun with:
-v
==1854== Use --track-origins=yes to see where uninitialised values
come from
==1854== ERROR SUMMARY: 13 errors from 3 contexts (suppressed: 0
from 0)
$
```

Shell Box 15-5: Running example 15.2 with valgrind for the second time

As you can see, the first run went well, with no memory access issues, even though the aforementioned race condition is still clear to us. In the second run, however, something goes wrong when one of the threads tries to access the string `Orange` pointed to by `str2`.

What this means is that the passed pointer to the second thread has become dangling. In the preceding output, you can clearly see that the stack trace points to line inside the `thread_body` function where there is the `printf` statement. Note that the stack trace actually refers to the `puts` function because our C compiler has replaced the `printf` statement with the equivalent `puts` statement. The preceding output also shows that the `write` system call is using a pointer named `buf` that points to a memory region that *is not initialized or allocated*.

Looking at the preceding example, `valgrind` doesn't conclude whether a pointer is dangling or not. It simply reports the invalid memory access.

Before the error messages regarding the bad memory access, you can see that the string `Orange` is printed even though the access for reading it is invalid. This just goes to show how easily things can get complicated when we have code running in a concurrent fashion.

In this section, we've taken a significant step forward in seeing how easy it is to write code that is not thread safe. Moving on, we're now going to demonstrate another interesting example that produces a data race. Here, we will see a more complex use of the pthread library and its various functions.

Example of data race

E*xample 15.3* demonstrates a data race. In previous examples, we didn't have a shared state, but in this example, we are going to have a variable shared between two threads.

The invariant constraint of this example is to protect the *data integrity* of the shared state, plus all other obvious constraints, like having no crashes, having no bad memory accesses, and so on. In other words, it doesn't matter how the output appears, but a thread must not write new values while the value of the shared variable has been changed by the other thread and the writer thread doesn't know the latest value. This is what we mean by "data integrity":

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

// The POSIX standard header for using pthread library
#include <pthread.h>

void* thread_body_1(void* arg) {
  // Obtain a pointer to the shared variable
  int* shared_var_ptr = (int*)arg;
  // Increment the shared variable by 1 by writing
  // directly to its memory address
  (*shared_var_ptr)++;
  printf("%d\n", *shared_var_ptr);
  return NULL;
}

void* thread_body_2(void* arg) {
  // Obtain a pointer to the shared variable
  int* shared_var_ptr = (int*)arg;
  // Increment the shared variable by 2 by writing
  // directly to its memory address
  *shared_var_ptr += 2;
  printf("%d\n", *shared_var_ptr);
```

```
  return NULL;
}

int main(int argc, char** argv) {

  // The shared variable
  int shared_var = 0;

  // The thread handlers
  pthread_t thread1;
  pthread_t thread2;

  // Create new threads
  int result1 = pthread_create(&thread1, NULL,
          thread_body_1, &shared_var);
  int result2 = pthread_create(&thread2, NULL,
          thread_body_2, &shared_var);

  if (result1 || result2) {
    printf("The threads could not be created.\n");
    exit(1);
  }

  // Wait for the threads to finish
  result1 = pthread_join(thread1, NULL);
  result2 = pthread_join(thread2, NULL);

  if (result1 || result2) {
    printf("The threads could not be joined.\n");
    exit(2);
  }
  return 0;
}
```

Code Box 15-5 [ExtremeC_examples_chapter15_3.c]: Example 15.3 with two threads operating on a single shared variable

The shared state has been declared as the first line in the `main` function. In this example, we are dealing with a single integer variable allocated form the Stack region of the main thread, but in real applications it can be far more complex. The initial value of the integer variable is zero, and each thread contributes directly to an increase in its value by writing to its memory location.

In this example, there is no local variable that is keeping a copy of the shared variable's value in each thread. However, you should be careful about the increment operations in threads because they are not *atomic* operations, and therefore are subject to experiencing different interleavings. We have explained this thoroughly in the previous chapter.

Each thread is able to change the value of the shared variable by using the pointer that it receives inside its companion function through the argument `arg`. As you can see in both calls to `pthread_create`, we are passing the address of the variable `shared_var` as the fourth argument.

It's worth noting that the pointer never becomes dangling in threads because the main thread doesn't exit, and it waits for the threads to finish by joining them.

Shell Box 15-6 shows us the outputs of multiple runs of the preceding code in order to produce different interleavings. Remember that we want data integrity to be preserved for the shared variable `shared_var`.

So, based on the logic defined in `thread_body_1` and `thread_body_2`, we can only have `1 3` and `2 3` as the acceptable outputs:

```
$ gcc ExtremeC_examples_chapter15_3.c -o ex15_3.out -lpthread
$ ./ex15_3.out
1
3
$
...
...
...
$ ./ex15_3.out
3
1
$
...
...
...
$ ./ex15_3.out
1
2
$
```

Shell Box 15-6: Multiple runs of example 15.3, in which we eventually see that the data integrity of the shared variable is not preserved

As you can see, the last run reveals that the data integrity condition has not been met for the shared variable.

In the last run, the first thread, the thread that has `thread_body_1` as its companion function, has read the value of the shared variable and it is `0`.

The second thread, the thread that has `thread_body_2` as its companion function, has also read the shared value and it is `0`. After this point, both threads try to increment the value of the shared variable and print it immediately. This is a breach of data integrity because when one thread is manipulating a shared state, the other thread shouldn't be able to write to it.

As we explained before, we have a clear data race over `shared_var` in this example.

Note:
When executing *example 15.3* yourself, be patient and wait to see the `1 2` output. It might happen after running the executable 100 times! I could have observed the data race on both macOS and Linux.

In order to resolve the preceding data race, we need to use a control mechanism, such as a semaphore or a mutex, to synchronize the access to the shared variable. In the next chapter, we will introduce a mutex to the preceding code that will do that for us.

Summary

This chapter was our first step towards writing multithreaded programs in C using the POSIX threading library. As part of this chapter:

- We went through the basics of the POSIX threading library, which is the main tool for writing multithreaded applications in POSIX-compliant systems.
- We explored the various properties of threads and their memory structure.
- We gave some insight about the available mechanisms for threads to communicate and share a state.
- We explained that how the memory regions available to all threads within the same process are the best way to share data and communicate.
- We talked about the kernel threads and the user-level threads and how they differ.
- We explained the joinable and detached threads and how they differ from the execution point of view.
- We demonstrated how to use the `pthread_create` and `pthread_join` functions and what arguments they receive.