Chapter 16

Thread Synchronization

In the previous chapter, we explained how to create and manage a POSIX thread. We also demonstrated two of the most common concurrency issues: race conditions and data races.

In this chapter, we are going to complete our discussion about multithreaded programming using the POSIX threading library and give you the required skills to control a number of threads.

If you remember from *Chapter 14*, *Synchronization*, we showed that concurrency-related problems are not actually issues; rather, they are consequences of the fundamental properties of a concurrent system. Therefore, you are likely to encounter them in any concurrent system.

We showed in the previous chapter that we could indeed produce such issues with the POSIX threading library as well. *Examples 15.2* and *15.3* from the previous chapter demonstrated the race condition and data race issues. Therefore, they will be our starting point to use the synchronization mechanisms provided by the pthread library in order to synchronize a number of threads.

In this chapter, we will cover the following topics:

- Using POSIX mutexes to protect critical sections accessing a shared resource.
- Using POSIX condition variables to wait for a specific condition.
- Using various types of locks together with mutexes and condition variables.
- Using POSIX barriers and the way they can help synchronize a number of threads.
- The concept of the semaphore and its counterpart object in the pthread library: the POSIX semaphore. You are going to find out that mutexes are just binary semaphores.
- The memory structure of a thread and how this structure can affect memory visibility in a multi-core system.

We start this chapter with a general talk about concurrency control. The following sections give you the necessary tools and constructs to write well-behaved multithreaded programs.

POSIX concurrency control

In this section, we are going to have a look at possible control mechanisms that are offered by the pthread library. Semaphores, mutexes, and condition variables alongside different types of locks are used in various combinations to bring determinism to multithreaded programs. First, we start with POSIX mutexes.

POSIX mutexes

The mutexes introduced in the pthread library can be used to synchronize both processes and threads. In this section, we are going to use them in a multithreaded C program in order to synchronize a number of threads.

As a reminder, a mutex is a semaphore that only allows one thread at a time to enter the critical section. Generally, a semaphore has the potential to let more than one thread enter its critical section.

Note:
Mutexes are also called *binary semaphores* because they are semaphores that accept only two states.

We start this section by resolving the data race issue observed as part of *example 15.3* in the previous chapter, using a POSIX mutex. The mutex only allows one thread at a time to enter the critical section and performs read and write operations on the shared variable. This way, it guarantees the data integrity of the shared variable. The following code box contains the solution to the data race issue:

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

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

// The mutex object used to synchronize the access to
// the shared state.
pthread_mutex_t mtx;

void* thread_body_1(void* arg) {
  // Obtain a pointer to the shared variable
  int* shared_var_ptr = (int*)arg;
```

```
  // Critical section
  pthread_mutex_lock(&mtx);
  (*shared_var_ptr)++;
  printf("%d\n", *shared_var_ptr);
  pthread_mutex_unlock(&mtx);

  return NULL;
}

void* thread_body_2(void* arg) {
  int* shared_var_ptr = (int*)arg;

  // Critical section
  pthread_mutex_lock(&mtx);
  *shared_var_ptr += 2;
  printf("%d\n", *shared_var_ptr);
  pthread_mutex_unlock(&mtx);

  return NULL;
}

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

  // The shared variable
  int shared_var = 0;

  // The thread handlers
  pthread_t thread1;
  pthread_t thread2;

  // Initialize the mutex and its underlying resources
  pthread_mutex_init(&mtx, NULL);

  // 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);
  }

  pthread_mutex_destroy(&mtx);

  return 0;
}
```

Code Box 16-1 [ExtremeC_examples_chapter15_3_mutex.c]: Using a POSIX mutex to resolve the data race issue found as part of example 15.3 in the previous chapter

If you compile the preceding code and run it as many times as you like, you will see only `1 3` or `2 3` in the output. That's because we are using a POSIX mutex object to synchronize the critical sections in the preceding code.

At the beginning of the file, we have declared a global POSIX mutex object as `mtx`. Then inside the `main` function, we have initialized the mutex with default attributes using the function `pthread_mutex_init`. The second argument, which is `NULL`, could be custom attributes specified by the programmer. We will go through an example of how to set these attributes in the upcoming sections.

The mutex is used in both threads to protect the critical sections embraced between the the `pthread_mutex_lock(&mtx)` and `pthread_mutex_unlock(&mtx)` statements.

Finally, before leaving the `main` function, we destroy the mutex object.

The first pair of `pthread_mutex_lock(&mtx)` and `pthread_mutex_unlock(&mtx)` statements, in the companion function `thread_body_1`, is making up the critical section for the first thread. Also, the second pair in the companion function `thread_body_2` is making up the critical section for the second thread. Both critical sections are protected by the mutex, and at each time, only one of the threads can be in its critical section and the other thread should wait outside of its critical section until the busy thread leaves.

As soon as a thread enters the critical section, it locks the mutex, and the other thread should wait behind the `pthread_mutex_lock(&mtx)` statement to have the mutex unlocked again.

By default, a thread waiting for a mutex to become unlocked goes into sleeping mode and doesn't do a *busy-wait*. But what if we wanted to do *busy-waiting* instead of going to sleep? Then we could use a *spinlock*. It would be enough to use the following functions instead of all the preceding mutex-related functions. Thankfully, pthread uses a consistent convention in naming the functions.

The spinlock-related types and functions are as follows.

- `pthread_spin_t`: The type used for creating a spinlock object. It is similar to the `pthread_mutex_t` type.
- `pthread_spin_init`: Initializes a spinlock object. It is similar to `pthread_mutex_init`.
- `pthread_spin_destroy`: Similar to `pthread_mutex_destory`.
- `pthread_spin_lock`: Similar to `pthread_mutex_lock`.
- `pthread_spin_unlock`: Similar to `pthread_mutex_unlock`.

As you see, it's pretty easy to just replace the preceding mutex types and functions with spinlock types and functions to have a different behavior, busy-waiting in this case, while waiting for a mutex object to become released.

In this section, we introduced POSIX mutexes and how they can be used to resolve a data race issue. In the next section, we will demonstrate how to use a condition variable in order to wait for a certain event to occur. We will be addressing the race condition that occurred in *example 15.2*, but we will make some modifications to the original example.

POSIX condition variables

If you remember from *example 15.2* in the previous chapter, we faced a race condition. Now, we want to bring up a new example that is very similar to *example 15.2*, but one where it would be simpler to use a condition variable. *Example 16.1* has two threads instead of three (which was the case for *example 15.2*), and they are required to print the characters `A` and `B` to the output, but we want them to be always in a specific order; first `A` and then `B`.

Our invariant constraint for this example is to *see first A and then B in the output* (plus data integrity for all shared states, no bad memory access, no dangling pointer, no crashes, and other obvious constraints). The following code demonstrates how we use a condition variable to come up with a working solution written in C for this example:

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

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

#define TRUE  1
#define FALSE 0
```

```
typedef unsigned int bool_t;

// A structure for keeping all the variables related
// to a shared state
typedef struct {
  // The flag which indicates whether 'A' has been printed or not
  bool_t          done;
  // The mutex object protecting the critical sections
  pthread_mutex_t mtx;
  // The condition variable used to synchronize two threads
  pthread_cond_t  cv;
} shared_state_t;

// Initializes the members of a shared_state_t object
void shared_state_init(shared_state_t *shared_state) {
  shared_state->done = FALSE;
  pthread_mutex_init(&shared_state->mtx, NULL);
  pthread_cond_init(&shared_state->cv, NULL);
}

// Destroy the members of a shared_state_t object
void shared_state_destroy(shared_state_t *shared_state) {
  pthread_mutex_destroy(&shared_state->mtx);
  pthread_cond_destroy(&shared_state->cv);
}

void* thread_body_1(void* arg) {
  shared_state_t* ss = (shared_state_t*)arg;
  pthread_mutex_lock(&ss->mtx);
  printf("A\n");
  ss->done = TRUE;
  // Signal the threads waiting on the condition variable
  pthread_cond_signal(&ss->cv);
  pthread_mutex_unlock(&ss->mtx);
  return NULL;
}

void* thread_body_2(void* arg) {
  shared_state_t* ss = (shared_state_t*)arg;
  pthread_mutex_lock(&ss->mtx);
  // Wait until the flag becomes TRUE
  while (!ss->done) {
    // Wait on the condition variable
    pthread_cond_wait(&ss->cv, &ss->mtx);
  }
  printf("B\n");
  pthread_mutex_unlock(&ss->mtx);
  return NULL;
```

```
}

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

  // The shared state
  shared_state_t shared_state;

  // Initialize the shared state
  shared_state_init(&shared_state);

  // The thread handlers
  pthread_t thread1;
  pthread_t thread2;

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

  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);
  }

  // Destroy the shared state and release the mutex
  // and condition variable objects
  shared_state_destroy(&shared_state);

  return 0;
}
```

Code Box 16-2 [ExtremeC_examples_chapter16_1_cv.c]: Using a POSIX condition variable to dictate a specific order between two threads

In the preceding code, it's good to use a structure in order to encapsulate the shared mutex, the shared condition variable, and the shared flag into a single entity. Note that we are only able to pass a single pointer to each thread. Therefore, we had to stack up the needed shared variables into a single structure variable.

As the second type definition (after `bool_t`) in the example, we have defined a new type, `shared_state_t`, as follows:

```
typedef struct {
  bool_t          done;
  pthread_mutex_t mtx;
  pthread_cond_t  cv;
} shared_state_t;
```

Code Box 16-3: Putting all shared variables required for example 16.1 into one structure

After the type definitions, we defined two functions in order to initialize and destroy the `shared_state_t` instances. They can be thought of as the *constructor* and *destructor* functions for the type `shared_state_t` respectively. To read more about constructor and destructor functions, please refer to *Chapter 6*, *OOP and Encapsulation*.

This is how we use a condition variable. A thread can *wait* (or *sleep*) on a condition variable, and then in the future, it becomes notified to wake up. More than that, a thread can *notify* (or *wake up*) all other threads waiting (or sleeping) on a condition variable. All these operations *must* be protected by a mutex, and that's why you should always use a condition variable together with a mutex.

We did the very same in the preceding code. In our shared state object, we have a condition variable, together with a companion mutex that is supposed to protect the condition variable. To emphasize again, a condition variable is supposed to be used only in critical sections protected by its companion mutex.

So, what happens in the preceding code? In the thread that is supposed to print `A`, it tries to lock the `mtx` mutex using a pointer to the shared state object. When the lock is acquired, the thread prints `A`, it sets the flag `done`, and it finally notifies the other thread, which could be waiting on the condition variable `cv`, by calling the `pthread_cond_signal` function.

On the other hand, if in the meantime the second thread becomes active and the first thread has not printed `A` yet, the second thread tries to acquire the lock over `mtx`. If it succeeds, it checks the flag `done`, and if it's false, it simply means that the first thread has not entered its critical section yet (otherwise the flag should have been true). Therefore, the second thread waits on the condition variable and immediately releases the CPU by calling the `pthread_cond_wait` function.

It is very important to note that upon waiting on a condition variable, the associated mutex becomes released and the other thread can continue. Also, upon becoming active and exiting the waiting state, the associated mutex should be acquired again. For some good practice in condition variables, you could go through other possible interleavings.

Note:
The function `pthread_cond_signal` can only be used to notify just one single thread. If you're going to notify all the threads waiting for a condition variable, you have to use the `pthread_cond_broadcast` function. We are going to give an example of this shortly.

But why did we use a `while` loop for checking the flag `done` when it could be a simple `if` statement? That's because the second thread can be notified by other sources rather than just the first thread. In those cases, if the thread could obtain the lock over its mutex upon exiting the wait and become active again, it could check the loop's condition, and if it is not met yet, it should wait again. It is an accepted technique to wait for a condition variable inside a loop, until its condition matches what we are waiting for.

The preceding solution satisfies the memory visibility constraint too. As we've explained in the previous chapters, all locking and unlocking operations are liable to trigger a memory coherence among various CPU cores; therefore, the values seen in different cached versions of the flag `done` are always recent and equal.

The race condition issue observed in examples 15.2 and 16.1 (in case of having no control mechanism in place), could also be resolved using POSIX barriers. In the next section, we are going to talk about them and rewrite *example 16.1* using a different approach.

POSIX barriers

POSIX barriers use a different approach for synchronizing a number of threads. Just like a group of people who are planning to do some tasks in parallel and at some points need to rendezvous, reorganize, and continue, the same thing can happen for threads (or even processes). Some threads do their tasks faster, and others are slower. But there can be a checkpoint (or rendezvous point) at which all threads must stop and wait for the others to join them. These checkpoints can be simulated by using *POSIX barriers*.

The following code uses barriers to propose a solution to the issues seen in *example 16.1*. As a reminder, in *example 16.1*, we had two threads. One of them was to `print A`, and the other thread was to `print B`, and we wanted to always see `A` first and `B` second in the output, regardless of various interleavings:

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

```
#include <pthread.h>

// The barrier object
pthread_barrier_t barrier;

void* thread_body_1(void* arg) {
  printf("A\n");
  // Wait for the other thread to join
  pthread_barrier_wait(&barrier);
  return NULL;
}

void* thread_body_2(void* arg) {
  // Wait for the other thread to join
  pthread_barrier_wait(&barrier);
  printf("B\n");
  return NULL;
}

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

  // Initialize the barrier object
  pthread_barrier_init(&barrier, NULL, 2);

  // The thread handlers
  pthread_t thread1;
  pthread_t thread2;

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

  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);
  }

  // Destroy the barrier object
```

```
  pthread_barrier_destroy(&barrier);

  return 0;
}
```

Code Box 16-4 [ExtremeC_examples_chapter16_1_barrier.c]: A solution for example 16.1 using POSIX barriers

As you can see, the preceding code is much smaller than the code we wrote using condition variables. Using POSIX barriers, it would be very easy to synchronize some threads at some certain points during their execution.

First, we have declared a global barrier object of type `pthread_barrier_t`. Then, inside the `main` function, we have initialized the barrier object using the function `pthread_barrier_init`.

The first argument is a pointer to the barrier object. The second argument is the custom attributes of the barrier object. Since we are passing `NULL`, it means that the barrier object will be initialized using the default values for its attributes. The third argument is important; it is the number of threads that should become waiting on the same barrier object by calling the function `pthread_barrier_wait` and only after that are all of them released and allowed to continue.

For the preceding example, we set it to 2. Therefore, only when there are two threads waiting on the barrier object, both of them are unblocked and they can continue. The rest of the code is pretty similar to previous examples, and has been explained in the previous section.

A barrier object can be implemented using a mutex and a condition variable similar to what we did in the previous section. In fact, a POSIX-compliant operating system doesn't provide such a thing as a barrier in its system call interface, and most implementations are made using a mutex and a condition variable.

That's basically why some operating systems like macOS does not provide implementations for POSIX barriers. The preceding code cannot be compiled in a macOS machine since the POSIX barrier functions are not defined. The preceding code is tested both in Linux and FreeBSD and works on both of them. Therefore, be careful about using barriers, because using them makes your code less portable.

The fact that macOS doesn't provide POSIX barrier functions simply means that it is partially POSIX-compliant and the programs using barriers (which is standard of course) cannot be compiled on macOS machines. This is against the C philosophy, which is *to write once, and compile anywhere*.

As the final note in this section, POSIX barriers guarantee memory visibility. Similarly to lock and unlock operations, waiting on barriers ensures that all the cached versions of the same variable are synchronized throughout various threads while they are going to leave the barrier point.

In the next section, we will be giving an example of semaphores. They are not used often in concurrent development, but they have their own special usages.

A specific type of semaphore, binary semaphores (interchangeably referred to as mutexes), is used often and you have seen a number of examples relating to that in the previous sections.

POSIX semaphores

In most cases, mutexes (or *binary semaphores*) are enough to synchronize a number of threads accessing a shared resource. That's because, in order to make read and write operations sequentially, only one thread should be able to enter the critical section at a time. It's known as *mutual exclusion*, hence, "mutex."

In some scenarios however, you might want to have more than one thread to enter the critical section and operate on the shared resource. This is the scenario in which you should use *general semaphores*.

Before we go into an example regarding general semaphores, let's bring up an example regarding a binary semaphore (or a mutex). We won't be using the `pthread_mutex_*` functions in this example; instead, we will be using `sem_*` functions which are supposed to expose semaphore-related functionalities.

Binary semaphores

The following code is the solution made using semaphores for *example 15.3*. As a reminder, it involved two threads; each of them incrementing a shared integer by a different value. We wanted to protect the data integrity of the shared variable. Note that we won't be using POSIX mutexes in the following code:

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

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

// The semaphores are not exposed through pthread.h
#include <semaphore.h>
```

```
// The main pointer addressing a semaphore object used
// to synchronize the access to the shared state.
sem_t *semaphore;

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

void* thread_body_2(void* arg) {
  // Obtain a pointer to the shared variable
  int* shared_var_ptr = (int*)arg;
  // Waiting for the semaphore
  sem_wait(semaphore);
  // Increment the shared variable by 1 by writing directly
  // to its memory address
  (*shared_var_ptr) += 2;
  printf("%d\n", *shared_var_ptr);
  // Release the semaphore
  sem_post(semaphore);
  return NULL;
}

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

  // The shared variable
  int shared_var = 0;

  // The thread handlers
  pthread_t thread1;
  pthread_t thread2;

#ifdef __APPLE__
  // Unnamed semaphores are not supported in OS/X. Therefore
  // we need to initialize the semaphore like a named one using
  // sem_open function.
  semaphore = sem_open("sem0", O_CREAT | O_EXCL, 0644, 1);
#else
  sem_t local_semaphore;
  semaphore = &local_semaphore;
```

```
  // Initiliaze the semaphore as a mutex (binary semaphore)
  sem_init(semaphore, 0, 1);
#endif

  // 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);
  }

#ifdef __APPLE__
  sem_close(semaphore);
#else
  sem_destroy(semaphore);
#endif

  return 0;
}
```

Code Box 16-5 [ExtremeC_examples_chapter15_3_sem.c]: A solution for example 15.3 using POSIX semaphores

The first thing you might notice in the preceding code is the different semaphore functions that we've used in Apple systems. In Apple operating systems (macOS, OS X, and iOS), *unnamed semaphores* are not supported. Therefore, we couldn't just use `sem_init` and `sem_destroy` functions. Unnamed semaphores don't have names (surprisingly enough) and they can only be used inside a process, by a number of threads. Named semaphores, on the other hand, are system-wide and can be seen and used by various processes in the system.

In Apple systems, the functions required for creating unnamed semaphores are marked as deprecated, and the semaphore object won't get initialized by `sem_init`. So, we had to use `sem_open` and `sem_close` functions in order to define named semaphores instead.

Named semaphores are used to synchronize processes, and we will explain them in *Chapter 18*, *Process Synchronization*. In other POSIX-compliant operating systems, Linux specifically, we still can use unnamed semaphores and have them initialized and destroyed by using the `sem_init` and `sem_destroy` functions respectively.

In the preceding code, we have included an extra header file, `semaphore.h`. As we've explained before, semaphores have been added as an extension to the POSIX threading library, and therefore, they are not exposed as part of the `pthread.h` header file.

After the header inclusion statements, we have declared a global pointer to a semaphore object. This pointer is going to point to a proper address addressing the actual semaphore object. We have to use a pointer here because, in Apple systems, we have to use the `sem_open` function, which returns a pointer.

Then, inside the `main` function, and in Apple systems, we create a named semaphore `sem0`. In other POSIX-compliant operating systems, we initialize the semaphore using `sem_init`. Note that in this case the pointer `semaphore` points to the variable `local_sempahore` allocated on top of the main thread's Stack. The pointer `semaphore` won't become a dangling pointer because the main thread doesn't exit and waits for the threads to get complete by joining them.

Note that we could distinguish between Apple and not - Apple systems by using the macro `__APPLE__`. This is a macro that is defined by default in C preprocessors being used in Apple systems. Therefore, we can rule out the code that is not supposed to be compiled on Apple systems by using this macro.

Let's look inside the threads. In companion functions, the critical sections are protected by `sem_wait` and `sem_post` functions which correspond to `pthread_mutex_lock` and `pthread_mutex_unlock` functions in the POSIX mutex API respectively. Note that `sem_wait` may allow more than one thread to enter the critical section.

The maximum number of threads that are allowed to be in the critical section is determined when initializing the semaphore object. We have passed the value `1` for the maximum number of threads as the last argument to the `sem_open` and `sem_init` functions; therefore, the semaphore is supposed to behave like a mutex.

To get a better understanding of semaphores, let's dive a bit more into the details. Each semaphore object has an integer value. Whenever a thread waits for a semaphore by calling the `sem_wait` function, if the semaphore's value is greater than zero, then the value is decreased by 1 and the thread is allowed to enter the critical section. If the semaphore's value is 0, the thread must wait until the semaphore's value becomes positive again. Whenever a thread exits the critical section by calling the `sem_post` function, the semaphore's value is incremented by 1. Therefore, by specifying the initial value `1`, we will eventually get a binary semaphore.

We end the preceding code by calling `sem_destroy` (or `sem_close` in Apple systems) which effectively releases the semaphore object with all its underlying resources. Regarding the named semaphores, since they can be shared among a number of processes, more complex scenarios can occur when closing a semaphore. We will cover these scenarios in *Chapter 18*, *Process Synchronization*.

General semaphores

Now, it's time to give a classic example that uses general semaphores. The syntax is pretty similar to the preceding code, but the scenario in which multiple threads are allowed to enter the critical section could be interesting.

This classic example involves the creation of 50 water molecules. For 50 water molecules, you need to have 50 oxygen atoms and 100 hydrogen atoms. If we simulate each atom using a thread, we require two hydrogen threads, and one oxygen thread
to enter their critical sections, in order to generate one water molecule and have it counted.

In the following code, we firstly create 50 oxygen threads and 100 hydrogen threads. For protecting the oxygen thread's critical section, we use a mutex, but for the hydrogen threads' critical sections, we use a general semaphore that allows two threads to enter the critical section simultaneously.

For signaling purposes, we use POSIX barriers, but since barriers are not implemented in Apple systems, we need to implement them using mutexes and condition variables. The following code box contains the code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include <errno.h> // For errno and strerror function

// The POSIX standard header for using pthread library
#include <pthread.h>
// Semaphores are not exposed through pthread.h
#include <semaphore.h>

#ifdef __APPLE__
// In Apple systems, we have to simulate the barrier
functionality.
pthread_mutex_t barrier_mutex;
pthread_cond_t  barrier_cv;
unsigned int    barrier_thread_count;
```

```
unsigned int    barrier_round;
unsigned int    barrier_thread_limit;

void barrier_wait() {
  pthread_mutex_lock(&barrier_mutex);
  barrier_thread_count++;
  if (barrier_thread_count >= barrier_thread_limit) {
    barrier_thread_count = 0;
    barrier_round++;
    pthread_cond_broadcast(&barrier_cv);
  } else {
    unsigned int my_round = barrier_round;
    do {
      pthread_cond_wait(&barrier_cv, &barrier_mutex);
    } while (my_round == barrier_round);
  }
  pthread_mutex_unlock(&barrier_mutex);
}

#else
// A barrier to make hydrogen and oxygen threads synchronized
pthread_barrier_t water_barrier;
#endif

// A mutex in order to synchronize oxygen threads
pthread_mutex_t   oxygen_mutex;

// A general semaphore to make hydrogen threads synchronized
sem_t*            hydrogen_sem;

// A shared integer counting the number of made water molecules
unsigned int      num_of_water_molecules;

void* hydrogen_thread_body(void* arg) {
  // Two hydrogen threads can enter this critical section
  sem_wait(hydrogen_sem);
  // Wait for the other hydrogen thread to join
#ifdef __APPLE__
  barrier_wait();
#else
  pthread_barrier_wait(&water_barrier);
#endif
  sem_post(hydrogen_sem);
  return NULL;
}

void* oxygen_thread_body(void* arg) {
  pthread_mutex_lock(&oxygen_mutex);
```

```
  // Wait for the hydrogen threads to join
#ifdef __APPLE__
  barrier_wait();
#else
  pthread_barrier_wait(&water_barrier);
#endif
  num_of_water_molecules++;
  pthread_mutex_unlock(&oxygen_mutex);
  return NULL;
}

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

  num_of_water_molecules = 0;

  // Initialize oxygen mutex
  pthread_mutex_init(&oxygen_mutex, NULL);

  // Initialize hydrogen semaphore
#ifdef __APPLE__
  hydrogen_sem = sem_open("hydrogen_sem",
          O_CREAT | O_EXCL, 0644, 2);
#else
  sem_t local_sem;
  hydrogen_sem = &local_sem;
  sem_init(hydrogen_sem, 0, 2);
#endif

  // Initialize water barrier
#ifdef __APPLE__
  pthread_mutex_init(&barrier_mutex, NULL);
  pthread_cond_init(&barrier_cv, NULL);
  barrier_thread_count = 0;
  barrier_thread_limit = 0;
  barrier_round = 0;
#else
  pthread_barrier_init(&water_barrier, NULL, 3);
#endif

  // For creating 50 water molecules, we need 50 oxygen atoms and
  // 100 hydrogen atoms
  pthread_t thread[150];

  // Create oxygen threads
  for (int i = 0; i < 50; i++) {
    if (pthread_create(thread + i, NULL,
                oxygen_thread_body, NULL)) {
      printf("Couldn't create an oxygen thread.\n");
      exit(1);
```

```
    }
  }

  // Create hydrogen threads
  for (int i = 50; i < 150; i++) {
    if (pthread_create(thread + i, NULL,
                hydrogen_thread_body, NULL)) {
      printf("Couldn't create an hydrogen thread.\n");
      exit(2);
    }
  }

  printf("Waiting for hydrogen and oxygen atoms to react ...\n");
  // Wait for all threads to finish
  for (int i = 0; i < 150; i++) {
    if (pthread_join(thread[i], NULL)) {
      printf("The thread could not be joined.\n");
      exit(3);
    }
  }

  printf("Number of made water molecules: %d\n",
         num_of_water_molecules);

#ifdef __APPLE__
  sem_close(hydrogen_sem);
#else
  sem_destroy(hydrogen_sem);
#endif

  return 0;
}
```

Code Box 16-6 [ExtremeC_examples_chapter16_2.c]: Using a general semaphore to simulate the process of creating 50 water molecules out of 50 oxygen atoms and 100 hydrogen atoms

In the beginning of the code, there are a number of lines that are surrounded by `#ifdef __APPLE__` and `#endif`. These lines are only compiled in Apple systems. These lines are mainly the implementation and variables required for simulating POSIX barrier behavior. In other POSIX-compliant systems other than Apple, we use an ordinary POSIX barrier. We won't go through the details of the barrier implementation on Apple systems here, but it is worthwhile to read the code and understand it thoroughly.

As part of a number of global variables defined in the preceding code, we have declared the mutex `oxygen_mutex`, which is supposed to protect the oxygen threads' critical sections. At each time, only one oxygen thread (or oxygen atom) can enter the critical section.

Then in its critical section, an oxygen thread waits for two other hydrogen threads to join and then it continues to increment the water molecule counter. The increment happens within the oxygen's critical section.

To elaborate more on the things that happen inside the critical sections, we need to explain the role of the general semaphore. In the preceding code, we have also declared the general semaphore `hydrogen_sem`, which is supposed to protect hydrogen threads' critical sections. At each time, only a maximum of two hydrogen threads can enter their critical sections, and they wait on the barrier object shared between the oxygen and hydrogen threads.

When the number of waiting threads on the shared barrier object reaches two, it means that we have got one oxygen and two hydrogens, and then voilà: a water molecule is made, and all waiting threads can continue. Hydrogen threads exit immediately, but the oxygen thread exists only after incrementing the water molecules counter.

We close this section with this last note. In *example 16.2*, we used the `pthread_cond_broadcast` function when implementing the barriers for Apple systems. It signals all threads waiting on the barrier's condition variable that are supposed to continue after having other threads joining them.

In the next section, we are going to talk about the memory model behind POSIX threads and how they interact with their owner process's memory. We will also look at examples about using the Stack and Heap segments and how they can lead to some serious memory-related issues.

POSIX threads and memory

This section is going to talk about the interactions between the threads and the process's memory. As you know, there are multiple segments in a process's memory layout. The Text segment, Stack segment, Data segment, and Heap segment are all part of this memory layout, and we covered them in *Chapter 4*, *Process Memory Structure*. Threads interact differently with each of these memory segments. As part of this section, we only discuss Stack and Heap memory regions because they are the most used and problematic areas when writing multithreaded programs.

In addition, we discuss how thread synchronization and a true understanding of the memory model behind a thread can help us develop better concurrent programs. These concepts are even more evident regarding the Heap memory because the memory management is manual there and in a concurrent system, threads are responsible for allocating and releasing Heap blocks. A trivial race condition can cause serious memory issues, therefore proper synchronization should be in place to avoid such disasters.

In the next subsection, we are going to explain how the Stack segment is accessed by different threads and what precautions should be taken.

Stack memory

Each thread has its own Stack region that is supposed to be private to that thread only. A thread's Stack region is part of the owner process's Stack segment and all threads, by default, should have their Stack regions allocated from the Stack segment. It is also possible that a thread has a Stack region that is allocated from the Heap segment. We will show in future examples how to do this, but for now, we assume that a thread's Stack is part of the process's Stack segment.

Since all threads within the same process can read and modify the process's Stack segment, they can effectively read and modify each other's Stack regions, but they *should not*. Note that working with other threads' Stack regions is considered dangerous behavior because the variables defined on top of the various Stack regions are subject to deallocation at any time, especially when a thread exits or a function returns.

That's why we try to assume that a Stack region is only accessible by its owner thread and not by the other threads. So, *local variables* (those variables declared on top the Stack) are considered to be private to the thread and should not be accessed by other threads.

In single-threaded applications, we have always one thread which is the main thread. Therefore, we use its Stack region like we use the process's Stack segment. That's because, in a single-threaded program, there is no boundary between the main thread and the process itself. But the situation is different for a multithreaded program. Each thread has its own Stack region which is different from another thread's Stack region.

When creating a new thread, a memory block is allocated for the Stack region. If not specified by the programmer upon creation, the Stack region will have a default Stack Size, and it will be allocated from the Stack segment of the process. The default Stack size is platform dependent and varies from one architecture to another. You can use the command `ulimit -s` to retrieve the default Stack size in a POSIX-compliant system.

On my current platform, which is macOS on an Intel 64-bit machine, the default Stack size is 8 MB:

```
$ ulimit -s
8192
$
```

Shell Box 16-1: Reading the default Stack size

The POSIX threading API allows you to set the Stack region for a new thread. In the following example, *example 16.3*, we have two threads. For one of them, we use the default Stack settings, and for the other one, we will allocate a buffer from the Heap segment and set it as the Stack region of that thread. Note that, when setting the Stack region, the allocated buffer should have a minimum size; otherwise it cannot be used as a Stack region:

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

#include <pthread.h>

void* thread_body_1(void* arg) {
  int local_var = 0;
  printf("Thread1 > Stack Address: %p\n", (void*)&local_var);
  return 0;
}

void* thread_body_2(void* arg) {
  int local_var = 0;
  printf("Thread2 > Stack Address: %p\n", (void*)&local_var);
  return 0;
}

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

  size_t buffer_len = PTHREAD_STACK_MIN + 100;
  // The buffer allocated from heap to be used as
  // the thread's stack region
  char *buffer = (char*)malloc(buffer_len * sizeof(char));

  // The thread handlers
  pthread_t thread1;
  pthread_t thread2;

  // Create a new thread with default attributes
  int result1 = pthread_create(&thread1, NULL,
          thread_body_1, NULL);

  // Create a new thread with a custom stack region
  pthread_attr_t attr;
  pthread_attr_init(&attr);
  // Set the stack address and size
  if (pthread_attr_setstack(&attr, buffer, buffer_len)) {
    printf("Failed while setting the stack attributes.\n");
    exit(1);
  }
```

```
  int result2 = pthread_create(&thread2, &attr,
          thread_body_2, NULL);

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

  printf("Main Thread > Heap Address: %p\n", (void*)buffer);
  printf("Main Thread > Stack Address: %p\n", (void*)&buffer_len);

  // 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(3);
  }

  free(buffer);

  return 0;
}
```

Code Box 16-7 [ExtremeC_examples_chapter16_3.c]: Setting a Heap block as a thread's Stack region

To start the program, we create the first thread with the default Stack settings. Therefore, its Stack should be allocated from the Stack segment of the process. After that, we create the second thread by specifying the memory address of a buffer supposed to be the Stack region of the thread.

Note that the specified size is `100` bytes more than the already defined minimum Stack size indicated by the `PTHREAD_STACK_MIN` macro. This constant has different values on different platforms, and it is included as part of the header file `limits.h`.

If you build the preceding program and run it on a Linux device, you will see something like the following:

```
$ gcc ExtremeC_examples_chapter16_3.c -o ex16_3.out -lpthread
$ ./ex16_3.out
Main Thread > Heap Address: 0x55a86a251260
Main Thread > Stack Address: 0x7ffcb5794d50
Thread2 > Stack Address: 0x55a86a2541a4
```

```
Thread1 > Stack Address: 0x7fa3e9216ee4
$
```

Shell Box 16-2: Building and running example 16.3

As is clear from the output seen in *Shell Box 16-2*, the address of the local variable `local_var` that is allocated on top of the second thread's Stack belongs to a different address range (the range of the Heap space). This means that the Stack region of the second thread is within the Heap. This is not true for the first thread, however.

As the output shows, the address of the local variable in the first thread falls within the address range of the Stack segment of the process. As a result, we could successfully set a new Stack region allocated from the Heap segment, for a newly created thread.

The ability to set the Stack region of a thread can be crucial in some use cases. For example, in memory-constrained environments where the total amount of memory is low for having big Stacks, or in high-performance environments in which the cost of allocating the Stack for each thread cannot be tolerated, using some preallocated buffers can be useful and the preceding procedure can be employed to set a preallocated buffer as the Stack region of a newly created thread.

The following example demonstrates how sharing an address in one thread's Stack can lead to some memory issues. When an address from a thread is shared, the thread should remain alive otherwise all pointers keeping that address become dangling.

The following code is not thread-safe, therefore we expect to see crashes from time to time in successive runs. The threads also have the default Stack settings which means their Stack regions are allocated from the process's Stack segment:

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

#include <pthread.h>

int* shared_int;

void* t1_body(void* arg) {
  int local_var = 100;
  shared_int = &local_var;
```

```
  // Wait for the other thread to print the shared integer
  usleep(10);
  return NULL;
}

void* t2_body(void* arg) {
  printf("%d\n", *shared_int);
  return NULL;
}

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

  shared_int = NULL;

  pthread_t t1;
  pthread_t t2;

  pthread_create(&t1, NULL, t1_body, NULL);
  pthread_create(&t2, NULL, t2_body, NULL);

  pthread_join(t1, NULL);
  pthread_join(t2, NULL);

  return 0;
}
```

Code Box 16-8 [ExtremeC_examples_chapter16_4.c]: Trying to read a variable allocated from another thread's Stack region

At the beginning, we have declared a global shared pointer. Since it is a pointer, it can accept any address regardless of where the address points to in the process's memory layout. It could be from the Stack segment or the Heap segment or even the Data segment.

In the preceding code, inside the `t1_body` companion function, we store the address of a local variable in the shared pointer. This variable belongs to the first thread, and it is allocated on top of the first thread's Stack.

From now on, if the first thread exits, the shared pointer becomes dangling, and any dereferencing probably leads to a crash, a logical error, or a hidden memory issue in the best case. In some interleavings, this would happen, and you see crashes from time to time if you run the preceding program multiple times.

As an important note, proper synchronization techniques should be employed if one thread is willing to use a variable allocated from another thread's Stack region. Since the lifetime of a Stack variable is bound to its scope, the synchronization should aim at keeping the scope alive until the consumer thread is done with the variable.

Note that for simplicity we didn't check the results of the pthread functions. It is always advised to do so and check the return values. Not all pthread functions behave the same on different platforms; if something goes wrong, you will become aware by checking the return values.

In this section, generally speaking, we showed why the addresses belonging to Stack regions shouldn't be shared, and why shared states better not be allocated from Stack regions. The next section talks about Heap memory, which is the most common place for storing shared states. As you might have guessed, working with the Heap is also tricky, and you should be careful about memory leaks.

Heap memory

The Heap segment and the Data segment are accessible by all threads. Unlike the Data segment, which is generated at compile time, the Heap segment is dynamic, and it is shaped at runtime. Threads can both read and modify the contents of the Heap. In addition, the contents of the Heap can stay as long as the process lives, and stay independent of the lifetime of the individual threads. Also, big objects can be put inside the Heap. All these factors together have caused the Heap to be a great place for storing states that are going to be shared among some threads.

Memory management becomes a nightmare when it comes to Heap allocation, and that is because of the fact that allocated memory should be deallocated at some point by one of the running threads otherwise it could lead to memory leaks.

Regarding concurrent environments, interleavings can easily produce dangling pointers; hence crashes show up. The critical role of synchronization is to put things in a specific order where no dangling pointer can be produced, and this is the hard part.

Let's look at the following example, *example 16.5*. There are five threads in this example. The first thread allocates an array from the Heap. The second and third threads populate the array in this form. The second thread populates the even indices in the array with the capital alphabet letters starting from *Z* and moving backward to *A*, and the third thread populates the odd indices with small alphabet letters starting from *a* and moving forward to *z*. The fourth thread prints the array. And finally, the fifth thread deallocates the array and reclaims the Heap memory.

All the techniques described in the previous sections about POSIX concurrency control should be employed in order to keep these threads from misbehaving within the Heap. The following code has no control mechanism in place, and obviously, it is not thread-safe. Note that the code is not complete. The complete version with the concurrency control mechanisms in place will come in the next code box:

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

```
#include <unistd.h>

#include <pthread.h>

#define CHECK_RESULT(result) \
if (result) { \
  printf("A pthread error happened.\n"); \
  exit(1); \
}

int TRUE = 1;
int FALSE = 0;

// The pointer to the shared array
char* shared_array;
// The size of the shared array
unsigned int shared_array_len;

void* alloc_thread_body(void* arg) {
  shared_array_len = 20;
  shared_array = (char*)malloc(shared_array_len * sizeof(char*));
  return NULL;
}

void* filler_thread_body(void* arg) {
  int even = *((int*)arg);
  char c = 'a';
  size_t start_index = 1;
  if (even) {
    c = 'Z';
    start_index = 0;
  }
  for (size_t i = start_index; i < shared_array_len; i += 2) {
    shared_array[i] = even ? c-- : c++;
  }
  shared_array[shared_array_len - 1] = '\0';
  return NULL;
}

void* printer_thread_body(void* arg) {
  printf(">> %s\n", shared_array);
  return NULL;
}

void* dealloc_thread_body(void* arg) {
  free(shared_array);
  return NULL;
}
```

```
int main(int argc, char** argv) {
  … Create threads ...
}
```

Code Box 16-9 [ExtremeC_examples_chapter16_5_raw.c]: Example 16.5 without any synchronization mechanism in place

It is easy to see that the preceding code is not thread-safe and it causes serious crashes because of the interference of the deallocator thread in deallocating the array.

Whenever the deallocator thread obtains the CPU, it frees the Heap-allocated buffer immediately, and after that the pointer `shared_array` becomes dangling, and other threads start to crash. Proper synchronization techniques should be used to ensure that the deallocation thread runs last and the proper order of logic in different threads are run.

In the following code block, we decorate the preceding code with POSIX concurrency control objects to make it thread-safe:

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

#include <pthread.h>

#define CHECK_RESULT(result) \
if (result) { \
  printf("A pthread error happened.\n"); \
  exit(1); \
}

int TRUE = 1;
int FALSE = 0;

// The pointer to the shared array
char* shared_array;
// The size of the shared array
size_t shared_array_len;

pthread_barrier_t alloc_barrier;
pthread_barrier_t fill_barrier;
pthread_barrier_t done_barrier;

void* alloc_thread_body(void* arg) {
  shared_array_len = 20;
  shared_array = (char*)malloc(shared_array_len * sizeof(char*));
```

```
  pthread_barrier_wait(&alloc_barrier);
  return NULL;
}

void* filler_thread_body(void* arg) {
  pthread_barrier_wait(&alloc_barrier);
  int even = *((int*)arg);
  char c = 'a';
  size_t start_index = 1;
  if (even) {
    c = 'Z';
    start_index = 0;
  }
  for (size_t i = start_index; i < shared_array_len; i += 2) {
    shared_array[i] = even ? c-- : c++;
  }
  shared_array[shared_array_len - 1] = '\0';
  pthread_barrier_wait(&fill_barrier);
  return NULL;
}

void* printer_thread_body(void* arg) {
  pthread_barrier_wait(&fill_barrier);
  printf(">> %s\n", shared_array);
  pthread_barrier_wait(&done_barrier);
  return NULL;
}

void* dealloc_thread_body(void* arg) {
  pthread_barrier_wait(&done_barrier);
  free(shared_array);
  pthread_barrier_destroy(&alloc_barrier);
  pthread_barrier_destroy(&fill_barrier);
  pthread_barrier_destroy(&done_barrier);
  return NULL;
}

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

  shared_array = NULL;

  pthread_barrier_init(&alloc_barrier, NULL, 3);
  pthread_barrier_init(&fill_barrier, NULL, 3);
  pthread_barrier_init(&done_barrier, NULL, 2);

  pthread_t alloc_thread;
  pthread_t even_filler_thread;
  pthread_t odd_filler_thread;
```

```
  pthread_t printer_thread;
  pthread_t dealloc_thread;

  pthread_attr_t attr;
  pthread_attr_init(&attr);
  int res = pthread_attr_setdetachstate(&attr,
          PTHREAD_CREATE_DETACHED);
  CHECK_RESULT(res);

  res = pthread_create(&alloc_thread, &attr,
          alloc_thread_body, NULL);
  CHECK_RESULT(res);

  res = pthread_create(&even_filler_thread,
          &attr, filler_thread_body, &TRUE);
  CHECK_RESULT(res);

  res = pthread_create(&odd_filler_thread,
          &attr, filler_thread_body, &FALSE);
  CHECK_RESULT(res);

  res = pthread_create(&printer_thread, &attr,
          printer_thread_body, NULL);
  CHECK_RESULT(res);

  res = pthread_create(&dealloc_thread, &attr,
          dealloc_thread_body, NULL);
  CHECK_RESULT(res);

  pthread_exit(NULL);

  return 0;
}
```

Code Box 16-10 [ExtremeC_examples_chapter16_5.c]: Example 16.5 with synchronization mechanisms in place

To make the code found in *Code Box 16-9* thread-safe, we have only used the POSIX barriers in the new code. It is the easiest approach to form a sequential execution order between a number of threads.

If you compare *Code Boxes 16-9* and *16-10*, you see how POSIX barriers are used to impose an order between various threads. The only exception is between two filler threads. The filler threads can be running independently without blocking each other, and since they are changing odd and even indices separately, no concurrent issue can be raised. Note that the preceding code cannot be compiled on Apple systems. You need to simulate the barrier behavior using mutexes and condition variables in these systems (as we did for *example 16.2*).

The following is the output of the preceding code. No matter how many times you run the program, it never crashes. In other words, the preceding code is guarded against the various interleavings, and it is thread-safe:

```
$ gcc ExtremeC_examples_chapter16_5.c -o ex16_5 -lpthread
$ ./ex16_5
>> ZaYbXcWdVeUfTgShRiQ
$ ./ex16_5
>> ZaYbXcWdVeUfTgShRiQ
$
```

Shell Box 16-3: Building and running example 16.5

In this section, we gave an example of using the Heap space as a place holder for shared states. Unlike the Stack memory, where memory deallocation happens automatically, Heap space deallocation should be performed explicitly. Otherwise, memory leaks are an imminent side effect.

The easiest and sometimes the best available place to keep the shared states, in terms of least memory management effort for the programmer, is the Data segment in which both allocation and deallocation happen automatically. Variables residing in the Data segment are considered global, and have the longest possible lifetime, from the very beginning moments of the process's birth until its very last moments. But this long lifetime can be considered negative in certain use cases, especially when you're going to keep a big object in the Data segment.

In the next section, we will talk about memory visibility and how POSIX functions guarantee that.

Memory visibility

We explained *memory visibility* and *cache coherency* in the previous chapters, regarding the systems with more than one CPU core. In this section, we want to look at the pthread library and see how it guarantees memory visibility.

As you know, a cache coherency protocol among CPU cores ensures that all cached versions of a single memory address in all CPU cores remain synchronized and updated regarding the latest changes made in one of the CPU cores. But this protocol should be triggered somehow.

There are APIs in the system call interface to trigger the cache coherency protocol and make the memory visible to all CPU cores. In pthread also, there are a number of functions that guarantee the memory visibility before their execution.

You may have encountered some of these functions before. A list of them is presented below:

- `pthread_barrier_wait`
- `pthread_cond_broadcast`
- `pthread_cond_signal`
- `pthread_cond_timedwait`
- `pthread_cond_wait`
- `pthread_create`
- `pthread_join`
- `pthread_mutex_lock`
- `pthread_mutex_timedlock`
- `pthread_mutex_trylock`
- `pthread_mutex_unlock`
- `pthread_spin_lock`
- `pthread_spin_trylock`
- `pthread_spin_unlock`
- `pthread_rwlock_rdlock`
- `pthread_rwlock_timedrdlock`
- `pthread_rwlock_timedwrlock`
- `pthread_rwlock_tryrdlock`
- `pthread_rwlock_trywrlock`
- `pthread_rwlock_unlock`
- `pthread_rwlock_wrlock`
- `sem_post`
- `sem_timedwait`
- `sem_trywait`
- `sem_wait`
- `semctl`
- `semop`

Other than local caches in CPU cores, the compilers can also introduce caching mechanisms for the frequently used variables. For this to happen, the compiler needs to analyze the code and optimize it in a way that means frequently used variables are written to and read from the compiler caches. These are software caches that are put in the final binary by the compiler in order to optimize and boost the execution of the program.

While these caches can be beneficial, they potentially add another headache while writing multithreaded code and raise some memory visibility issues. Therefore, sometimes these caches must be disabled for specific variables.

The variables that are not supposed to be optimized by the compiler via caching can be declared as *volatile*. Note that a volatile variable still can be cached at the CPU level, but the compiler won't optimize it by keeping it in compiler caches. A variable can be declared as volatile using the keyword `volatile`. Following is a declaration of an integer that is volatile:

```
volatile int number;
```

Code Box 16-11: Declaring a volatile integer variable

The important thing about volatile variables is that they don't solve the memory visibility problems in multi-threaded systems. In order to solve this issue, you need to use the preceding POSIX functions in their proper places in order to ensure memory visibility.

Summary

In this chapter, we covered the concurrent control mechanisms provided by the POSIX threading API. We have discussed:

- POSIX mutexes and how they should be used
- POSIX condition variables and barriers and how they should be used
- POSIX semaphores, and how binary semaphores and general semaphores differ
- How threads interact with the Stack region
- How to define a new Heap-allocated Stack region for a thread
- How threads interact with the Heap space
- Memory visibility and POSIX functions that guarantee memory visibility
- Volatile variables and compiler caches

In the next chapter, we will continue our discussion and we will talk about another approach for having concurrency in a software system: multi-processing. We will discuss how a process can be executed and how it is different from a thread.