CHAPTER

INTRODUCING THREADS

2

2.1 APPLICATIONS AND PROCESSES

Applications executed in a computing platform run as an operating system process, *which is the basic unit of resource allocation to an application*. Indeed, when the application starts execution, the operating system allocates a number of resources to the process:

- A protected fraction of the available memory. Protected means that no other process can access it.
- File handles required by the application.
- Access to communication ports and I/O devices.
- A share of the CPU cycles available in the executing platform (multitasking).

The process executes first some startup code needed to allocate the resources and set up the application environment. Then, it proceeds to execute the main() function provided by the programmer. Figure 2.1 shows the way in which the memory block allocated to the process is organized by the operating system. There are three distinct memory domains:

- A memory region where the program global variables are allocated. Global variables are those declared outside the function main(), and they are naturally allocated by the startup code before main() starts.
- Another memory region, called the *heap*, is reserved for the dynamic memory allocations the application may eventually perform. This is the memory allocated by the library functions malloc() in C or new in C++.
- Finally, there is a memory buffer called the *stack*, which is used to allocate the local variables defined inside main(), as well as the local variables involved in all the possible nested function calls originating from main(), as we will soon explain.

Pointers—initialized to point to the address of some memory block by malloc() or by the operator new—can be declared either as global variables or as local variables to main(). For a sequential process this is not important, but this difference becomes relevant for multithreaded processes.

Shared Memory Application Programming. http://dx.doi.org/10.1016/B978-0-12-803761-4.00002-2

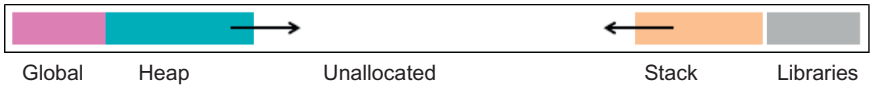

FIGURE 2.1

Memory organization in a sequential process.

2.1.1 ROLE OF THE STACK

The stack is a *container* in C++ language, used to store data, with a particular way of inserting or extracting data out of it. A clear understanding of its role is useful to understand a number of thread operation issues.

The stack operates like a pile of dishes: it is a LIFO (Last In, First Out) data structure where data can only inserted *(pushed)* or removed *(popped)* at the top. The stack buffer has an associated data item, the *stack pointer, SP*, which contains the address of the top of the stack. This address is of course increased (or decreased) when data is pushed (or popped). The addresses of the different data items stored in the stack are simply determined by their distances (offsets) to the stack pointer. When main() starts, its local variables are allocated in the stack.

There are several important reasons for using this kind of data structure for allocating local variables, having to do mainly with multithreading, that will be clarified later on. One important issue is the management of function calls. The discussion that follows captures the essential ideas, avoiding unnecessary complications. Details may differ across different computing platforms and operating systems.

When the compiler writes the code for a function, it assumes that its local variables are placed in a well-defined order—the order in which they have been declared inside the function code—in a stack. In the function code, their addresses are manipulated as offsets with respect to a *yet unknown* stack pointer. Indeed, at compile time the place in memory where the stack buffer will be allocated is not yet known. The compiler simply assumes that the correct SP address is stored in a specific, dedicated processor register. When main() starts, the stack memory address is known, and so is the stack pointer SP after the local variables have been allocated on the stack.

When a function is called, the following steps are taken by the process (see Figure 2.2):

- The return address where the code must resume execution after the function returns is pushed to the stack.
- The argument values that need to be passed to the function (represented by the first buffer following the return address in Figure 2.2) are also pushed to the stack.
- At this point, execution is transferred to the called function.

When the called function starts execution, it pushes all its local variables to the stack and performs its work (see Figure 2.2). The situation is now very much the same as before the call. The local variables of the running function are sitting just below the new stack pointer, and the argument values passed to the function are also sitting in a well-defined place on the stack, so that the function code can access them. The compiler has also determined their addresses in terms of their offsets with respect to the new stack pointer.

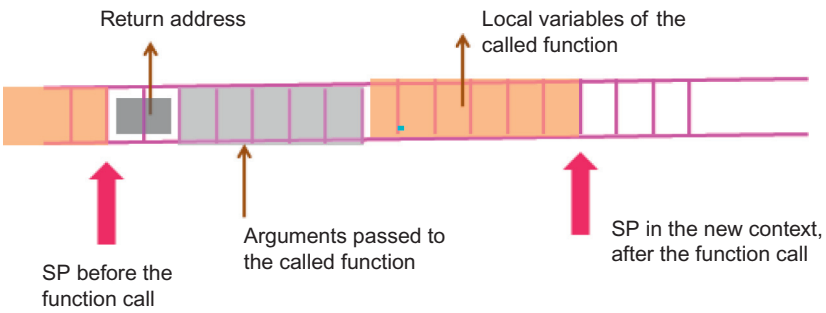

FIGURE 2.2

Behavior of the stack in a function call.

When the function returns, it destroys its local variables and the function arguments by popping them from the stack. After that, the runtime code pops the return address that was previously pushed by the caller and transfers control to this address. At this point, the original state of the caller stack is fully restored. It is clear that this mechanism is recursive, supporting nested function calls.

Application programmers do not really need to understand how function calls operate in the background, but this discussion is useful because it helps to better understand a few best practices required for multithreaded programming.

2.1.2 MULTITASKING

Multitasking is the capability of the operating system of running several applications at the same time. It is therefore deeply related to the OS capability for managing shared resources, and in particular for arbitrating conflicts resulting from resource over-commitment. In a multitasking environment, the totality of the computer's resources (memory, files, CPU time) are allocated to different applications, and they are managed in such a way that each one of them gets a share according to specific priority policies.

When CPU resources are over-committed, the operating system schedules the different active processes by performing a process switch at regular time intervals, distributing the available CPU cycles according to a well-defined priority policy. Multitasking is therefore the natural execution model of single or multiprocessor systems. It is the way several applications share the resources of a computing platform. Processes are, in fact, *highly autonomous and protected* independent execution streams. If a resource—CPU time, I/O device—is available to one of them, it may not available to the others. The operating system provides inter-process communication mechanisms, like pipes or signals. They are meant to enable communication *across* applications.

Switching among different processes naturally induces an execution overhead. Whenever a process switch occurs, all the process resources must be saved. If, for example, there are several open files, the file handles that identify each file, as well as the file pointers that identify the current positions inside

each file, must be saved. The operating system is also forced to save all other information related to the state of the process (instruction pointer, stack pointer, processor registers, etc.) needed to reconstruct its state at a later time slice.

2.2 MULTITHREADED PROCESSES

Processes are, as we said, the basic units of resource allocation, but *they are not the basic units of dispatching*. Indeed, they are not doomed to live with only the initial execution stream encoded in the main() function: additional internal asynchronous execution streams, or *threads*, can be launched. This is done by the native multithreading libraries that come with the OS: the Windows thread API, or the Posix Threads library—Pthreads for short—in Unix-Linux systems. A thread is therefore *a single sequential flow of control within a program*. When mapped to different cores allocated to an application, these independent, asynchronous flows of control implement parallel processing.

Parallel processing relies on exploitable concurrency. There are a number of concepts that will be systematically referred to, and it is convenient to be precise about their meaning:

- Concurrency exists in a computational problem when the problem can be decomposed into sub-problems—called tasks—that can safely execute at the same time.
- Tasks are fundamental units of sequential computation. They are units of work that express the underlying concurrency in the application, providing the opportunity for parallel execution. Tasks that can be active at the same time are considered to be concurrent.
- Parallel execution occurs when several tasks are actually doing some work at the same point in time. This happens when they are simultaneously executed by several threads.
- Concurrency can be beneficial to an application, even if there is no parallel execution. This point is further discussed at the end of the chapter.

The application performance can naturally be enhanced if concurrent tasks can be executed in parallel, overlapped in time as much as possible. Threads provide the way to implement this fact. This is summarized by the following statement:

> Threads execute tasks, transforming the potential parallelism expressed by the existence of concurrent tasks into real, mandatory parallelism implemented by their simultaneous execution.

One of the first things a programmer learns about programming is organizing and isolating tasks by subroutines or function calls. Indeed, we will see in the next chapter that, *in order to launch a new thread, a function must be provided that encapsulates the sequence of instructions to be executed by the thread*, in the same way the main() function displays the sequence of instructions to be executed by the initial thread in the process, called the *main thread*.

Threads are nevertheless more than functions. Function calls correspond to a *synchronous* transfer of control in the program. A standard function call is a very convenient way of *inserting* a set of instructions into an existing execution flow. Threads, on the other hand, correspond to *asynchronous* transfers of control to a new execution flow. The initial execution flow that launches the new thread

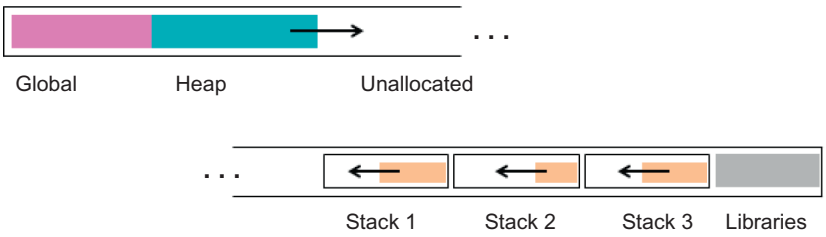

FIGURE 2.3

Memory organization in a multithreaded Unix process.

continues its own progress, and it is not forced to wait until the new thread finishes and returns (it may do so if needed, as will be explained in the next chapters).

2.2.1 LAUNCHING THREADS

Initially, a process consists of only one thread, the *main thread*. Next, the main thread can launch other threads by calling the appropriate library functions provided by the native libraries. From here on, there is absolutely no difference between the main thread and the other threads. Any thread can call again the appropriate library function and launch new threads. All the threads, no matter how they are created, are treated as equal by the operating system and scheduled in principle in a fair way. All these new execution streams are totally equivalent to the initial main() function. Threads can be thought of as asynchronous functions that become totally autonomous and take off *with their own stack*, where they manage their own local variables as well as their own nested function calls.

Threads share all the process resources *except the stack*. Each thread creation involves the creation of a new stack where the new thread function will allocate its local variables or the local variables of the functions it calls.

2.2.2 MEMORY ORGANIZATION OF A MULTITHREADED PROCESS

The memory organization in a multithreaded process is shown in Figure 2.3. The main difference with a sequential process is that there are now several stack buffers—one per thread—with a definite size. In fact, the stack size is one of the fundamental attributes of a thread. Default values are provided by the operating system, but they sometimes can be modified by the programmer, as discussed in the next chapter.

It is important to notice that:

- Global variables are *shared variables* that can be accessed by any thread.
- Local variables on the thread's stack are *private variables* that are only accessed in principle by the owner thread.

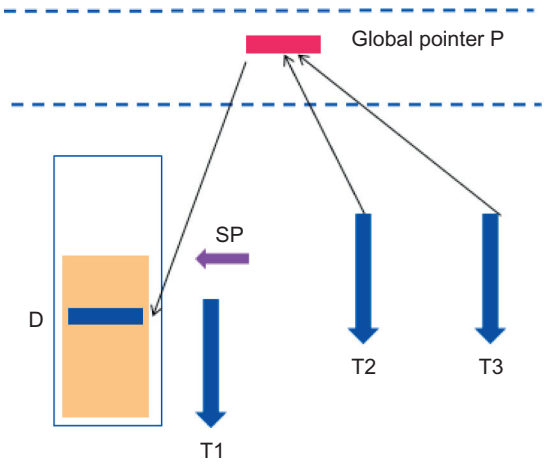

FIGURE 2.4

Publishing private data.

Notice also that dynamically allocated memory can be shared by all threads if it is referenced by a global pointer. Otherwise, dynamically allocated memory is only accessible by the thread that owns the pointer. This is the standard pattern for distinguishing between *shared* and *private* variables in multithreaded codes.

However, it would be wrong to conclude that it is impossible for a thread to read or modify data that is private to another thread. In fact, the owner thread can *publish* its local data by initializing a global pointer with its address. In Figure 2.4, a local data item is sitting in the thread T1 stack. This thread initializes a global pointer with the address of this private data item, and then threads T2 and T3 can use it to access T1's data item and eventually modify it. This mechanism is not unusual, and it will often be used later on to transfer data values across threads. However, as we will see when the time comes, it must be used with care.

2.2.3 THREADS AS LIGHTWEIGHT PROCESSES

We conclude therefore that each thread created inside the process has its own stack and stack pointer, as well as its own instruction list. However, all the other resources (global memory, files, I/O ports, . . .) allocated to the process are shared by all the threads.

A process may have more threads than available cores, and in this case CPU resources must be shared among threads: each one disposes of time slices of the available CPU time. However, switching among threads is much more efficient than switching among different processes. Indeed, since most of the global resources are shared by all threads, they do not need to be saved when the thread switch occurs.

To sum up:

- Threads are slightly more complex than ordinary method functions. Each thread has its own stack.
- Threads are simpler than processes.

Indeed, a thread can be seen as a sort of a "lightweight" process with simpler resource management, since the only data structure linked to the thread is the stack. It can also be seen as a function that has grown up and acquired its own execution context, i.e., its own stack.

2.2.4 HOW CAN SEVERAL THREADS CALL THE SAME FUNCTION?

In general, *threads call library functions*. In fact, several threads can call the *same* function. This means they all incorporate, somewhere in their code sequences, the function code (see Figure 2.5). Therefore, several threads can, at some time, find themselves *executing the same library function*. It is useful to take a close look at the way this happens.

Look at Figure 2.5, displaying two threads, T1 and T2, calling the same function, e.g., a function of the standard C or C++ libraries. These two calls are asynchronous, and they may or may not overlap in time. When called from T1 (or T2), the function will be operating on argument values and local variables that are in the T1 (or T2) stack. *These are two independent data sets*. However, there is only *one* piece of code for the function, sitting somewhere in memory. Now the question is, how can a unique suite of instructions operate simultaneously on two different data sets, and yet provide the correct results?

It is at this point that the stack properties play a role. The function body constructed by the compiler references all local addresses inside the function with the offset to an unknown stack pointer SP. Let SP1 and SP2 be the stack pointers of threads T1 and T2. The core running the thread T1 (or T2) has a stack pointer register SP holding the value SP1 (or SP2). When using a stack to manage

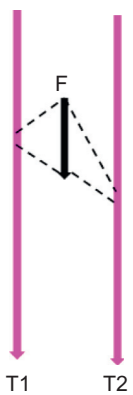

FIGURE 2.5

Two threads calling the same function.

local variables, switching the SP values switches a complete data set. The stack strategy to address local variables allows different threads to execute simultaneously the same function acting on different data sets.

A word of caution is needed here. Functions that are executed in a multithreaded environment must be *thread safe*, namely, they must be able to render the correct service to each caller even if they are called concurrently by different threads that are manipulating different data sets. *This is true only if the action of the function depends only on the arguments passed to it*. But this is not always the case. Some functions carry an internal state, persistent across successive calls, that also determines the outcome of the next function call. These functions are not thread safe, and the way to cope with this issue is the subject of Chapter 4.

2.2.5 COMMENT ABOUT HYPERTHREADING

As discussed in Chapter 1, hyperthreading is the capability of a core of *simultaneously* executing two or more threads. Figure 2.6 shows a core running two hardware threads. In this case, the two instruction sets are interleaved, and the core takes advantage of eventual delays in the execution of one of the threads to execute instructions of the other. In the Intel Xeon Phi, for example, decoding an instruction takes two cycles. The only way to try to obtain the execution of one instruction per cycle is running two hardware threads.

Once again, we have a core operating on two data sets. In this case, when hyperthreading is enabled, the core dedicates two registers to hold the stack pointer of the two stacks. In this way, each instruction hits its correct data target.

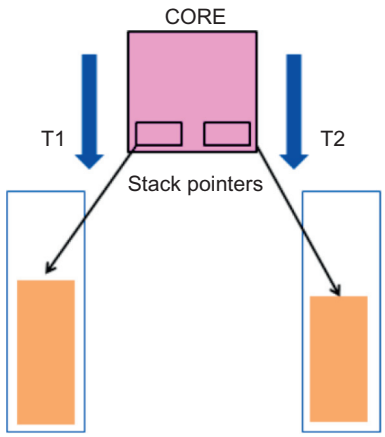

FIGURE 2.6

Single core running two hardware threads.

2.3 PROGRAMMING AND EXECUTION MODELS

It is useful to keep in mind the difference between *programming* and *execution models*:

- A programming model is the logical view that the programmer has of the application.
- An execution model is the precise way in which the application is processed by the target platform.

Programming models—like OpenMP, TBB, or the Pthreads library—tend to be as generic and universal as possible, to guarantee the portability of the application across different platforms. Execution models, in contrast, depend on the specific hardware and software environments of the target platform. For portable code, the same programming model must be mapped to different execution contexts. Here is the place where the compiler optimization options play a fundamental role in performance.

Figure 1.2 in Chapter 1 shows the *shared memory programming model* that applies to multithreaded applications. This is the simple and straightforward vision a programmer has of a multithreaded application: a number of virtual processing units—*virtual CPUs*—allocated to the process, having access to the totality of the memory owned by it. Different virtual CPUs are of course running different threads. When this simple programming model is mapped to an executing platform, ideally each virtual CPU should be mapped to a unique core. But this need not be the case: it is possible to have in an application more threads than cores allocated to the process.

When a multithreaded application is executed in a computing platform, two extreme cases can be distinguished:

- *Parallel execution:* the different threads are executed by different cores in a multicore system.
- *Concurrent execution:* the different threads are executed by only one core, or by a number of cores smaller than the number of threads.

The benefit of parallel execution in enhancing performance is evident. A process runs with optimal performance if it disposes of as many cores as active threads. But this is not necessary, because the operating system can schedule M threads on N cores ($M > N$) in a fair way, by allocating to each thread successive *time slices* on the available CPUs. It is therefore possible to engage more threads than CPUs in an application. This is a very common situation, for example, in the case of interactive applications, as will be shown later on. In particular, N may be equal to 1, and a correctly crafted code with M threads must run equally well (with much less performance, of course) and deliver the correct results. Testing a multithreaded code on a single CPU is a useful first step in the development of multithreaded applications.

One may, however, have doubts about the interest of over-committing the number of threads, in particular for an application that will run on a single processor system: a single execution stream looks, in principle, more natural and efficient. Nevertheless, because of the way the life cycle of a thread is conceived, concurrent programming with overcommitted cores can in many circumstances contribute to improve the performance or the efficiency of an application.

2.3.1 THREAD LIFE CYCLES

Figure 2.7 is a diagram showing the thread life cycle in an application. It represents all possible states of a thread from the moment it is created:

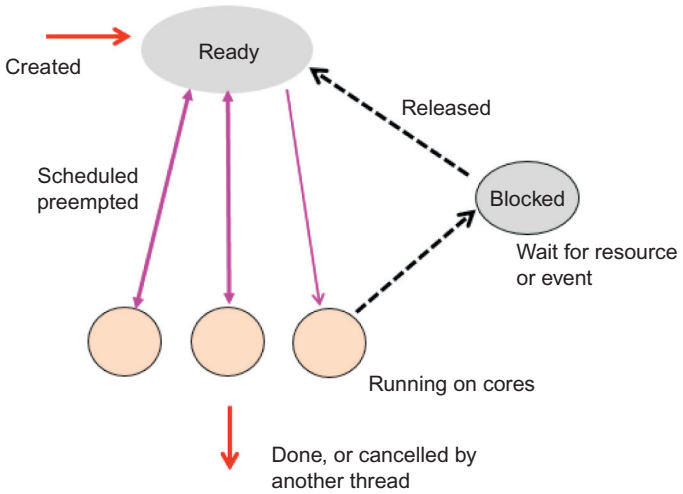

FIGURE 2.7

Life cycle of a thread.

- *Ready:* the thread is a ready pool that contains the set of threads that are waiting to be scheduled on a core.
- *Running on cores:* the thread is in progress. Two things can happen to a running thread:
  - If threads are over-committed, it will at some point be preempted by the operating system and placed back in the ready pool, in order to give other threads the opportunity to run. The operating system *does not preempts* threads if there are as many CPUs available as active threads.
  - It may be moved to a *blocked state*, releasing the core, waiting for an event. At some point, another thread will produce the expected event and release the waiting thread, which goes back to the ready pool to be rescheduled. This is an *event synchronization mechanism*, controlled by the programmer, and discussed in Chapter 6.
  - *Termination* occurs when the thread is done, or when it is canceled by other threads.

One important observation is that *the transition to a blocked state can be controlled by the programmer*. In all the native or basic libraries there are logical constructs that allow a programmer to put a thread out of the way for some time, until, for example, some data item takes a predefined value. This is called *waiting on a condition*, and plays a fundamental role in the synchronization of threads.

2.4 BENEFITS OF CONCURRENT PROGRAMMING

Figure 2.7 is a diagram showing the possible states in the life cycle of a thread. The existence of a blocked state sheds some ligth on the benefits of concurrent programming. We insist on the fact that *a*

thread in a blocked state is not using CPU resources, simply because it is not scheduled. CPU resources are fully available to the remaining running threads.

Overlapping computation and IO.

Let us assume that a process does a lot of computation and I/O. The process may be running concurrently with other processes in a multitasking environment, and the I/O devices it needs are not immediately available. In such a situation, *the whole process is blocked*. If, instead, the process is multithreaded and the I/O operation is handled by one specific thread, then only the I/O thread is blocked. The other threads can continue to run, for as long as the treatment they perform is independent of the outcome of the I/O operation. Overlapping computation and I/O in this way boosts the application performance—even in a single core system—when shared resources are over-committed (Figure 2.8).

Multithreaded servers.

Another example of concurrent programming can be found in the context of client—server software architectures. Let us assume that a heavyweight server has to deal with a large crowd of different clients, accessing the server to request specific services. A server that is not multithreaded will be providing undivided attention to the first client that starts a transaction. Any other client that comes later on will be obliged to wait in face of a dumb server, until it becomes available again for a new client.

One way out of this unacceptable situation is to have a multithreaded server that dispatches one thread per client to handle the requests, and returns immediately to listen to eventual new requests. If the server runs on a platform with a large number of processors, every client will get immediate optimum service. If the number of concurrent clients is much bigger than the CPU resources available, transactions will take roughly the same amount of time as the number of active clients, but at least every client will have the illusion of being immediately served. Performance is not boosted, but the response time of the server is acceptable. This example shows that concurrent programming allows for better interactive interfaces.

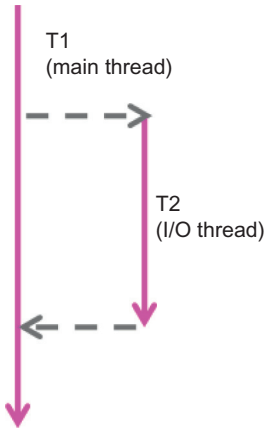

FIGURE 2.8

Overlapping computation and I/O.

Graphical interfaces.

Exactly the same argumentation applies to graphical interfaces (GUIs). Any application disposing of a GUI for interactive access will have from the start at least another thread, called the *event thread*, in addition to the main thread. The event thread is created by the system, and it is not directly seen by the programmer. But the programmer knows that this is the thread that will execute whatever functions he writes for the following operations:

- Handling external events, like mouse clicks or keyboard hits. These events are put by the system in a queue—called the *event queue*—and serviced by the event thread on a first come, first served basis.
- Partially or totally repainting and upgrading the GUI window. The repaint requests are queued with the external events, and serviced in the same way.

We conclude, as before, that concurrent programming is a necessary software technology for efficient interactive interfaces. This is the reason why the Windows operating system and the Java programming language—strongly oriented from the start toward graphical interfaces—have intrinsic multithreading services, instead of an external library.