Chapter 1

Introduction

1.1 Introduction

Parallel and distributed computing systems are now widely available. A *parallel system* consists of multiple processors that communicate with each other using shared memory. As the number of transistors on a chip increases, multiprocessor chips will become fairly common. With enough parallelism available in applications, such systems will easily beat sequential systems in performance. Figure 1.1 shows a parallel system with multiple processors. These processors communicate with each other using the shared memory. Each processor may also have local memory that is not shared with other processors.

We define *distributed systems* as those computer systems that contain multiple processors connected by a communication network. In these systems processors communicate with each other using messages that are sent over the network. Such systems are increasingly available because of decrease in prices of computer processors and the high-bandwidth links to connect them. Figure 1.2 shows a distributed system. The communication network in the figure could be a local area network such as an Ethernet, or a wide area network such as the Internet.

Programming parallel and distributed systems requires a different set of tools and techniques than that required by the traditional sequential software. The focus of this book is on these techniques.

1.2 Distributed Systems versus Parallel Systems

In this book, we make a distinction between distributed systems and parallel systems. This distinction is only at a logical level. Given a physical system in which processors have shared memory, it is easy to simulate messages. Conversely, given a physical system in which processors are connected by a network, it is possible to simulate shared memory. Thus a parallel hardware system may run distributed software and vice versa.

This distinction raises two important questions. Should we build parallel hardware or distributed hardware? Should we write applications assuming shared memory or message passing? At the hardware level, we would expect the prevalent model to be multiprocessor workstations connected by a network. Thus the system is both parallel and distributed. Why would the system not be completely parallel? There are many reasons.

- *Scalability*: Distributed systems are inherently more scalable than parallel systems. In parallel systems shared memory becomes a bottleneck when the number of processors is increased.

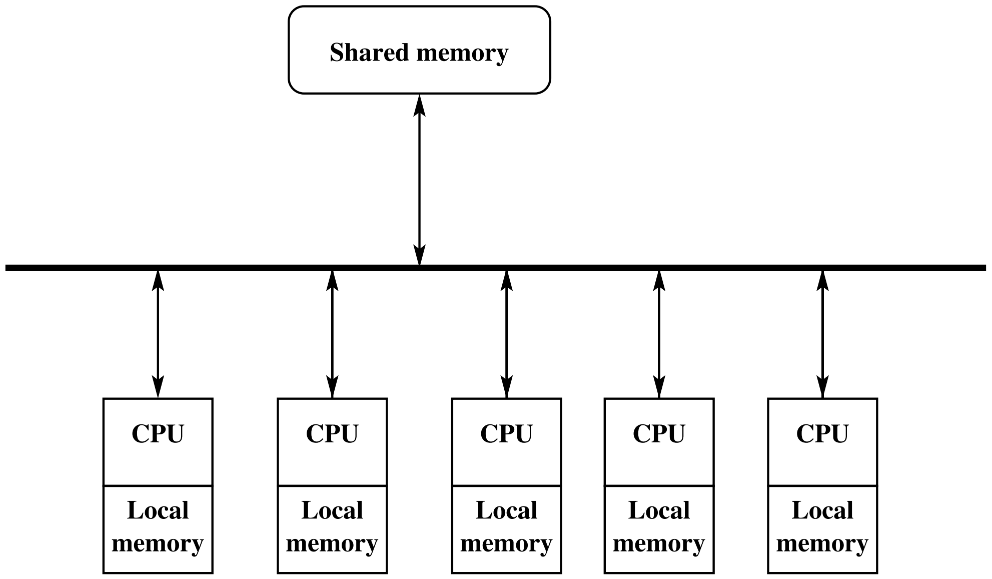

Figure 1.1: A parallel system

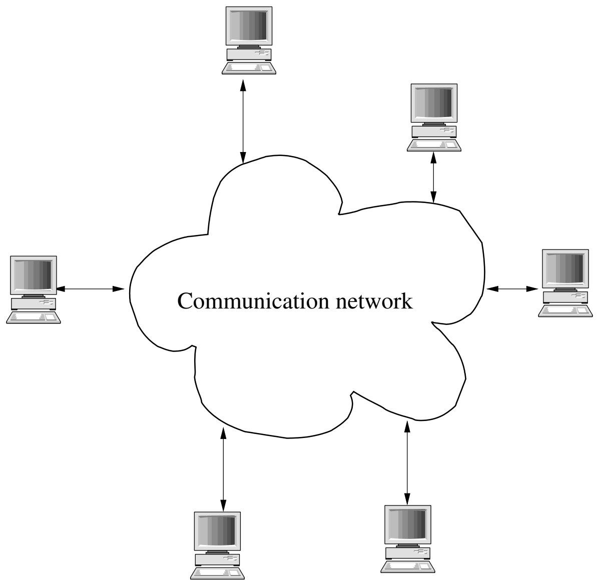

Figure 1.2: A distributed system

- *Modularity and heterogeneity*: A distributed system is more flexible because a single processor can be added or deleted easily. Furthermore, this processor can be of a type completely different from that of the existing processors.

- *Data sharing*: Distributed systems provide data sharing as in distributed databases. Thus multiple organizations can share their data with each other.

- *Resource sharing*: Distributed systems provide resource sharing. For example, an expensive special-purpose processor can be shared by multiple organizations.

- *Geographic structure*: The geographic structure of an application may be inherently distributed. The low communication bandwidth may force local processing. This is especially true for wireless networks.

- *Reliability*: Distributed systems are more reliable than parallel systems because the failure of a single computer does not affect the availability of others.

- *Low cost*: Availability of high-bandwidth networks and inexpensive workstations also favors distributed computing for economic reasons.

Why would the system not be a purely distributed one? The reasons for keeping a parallel system at each node of a network are mainly technological in nature. With the current technology it is generally faster to update a shared memory location than to send a message to another processor. This is especially true when the new value of the variable must be communicated to multiple processors. Consequently, it is more efficient to get fine-grain parallelism from a parallel system than from a distributed system.

So far our discussion has been at the hardware level. As mentioned earlier, the interface provided to the programmer can actually be independent of the underlying hardware. So which model would then be used by the programmer? At the programming level, we expect that programs will be written using multithreaded distributed objects. In this model, an application consists of multiple heavyweight processes that communicate using messages (or remote method invocations). Each heavyweight process consists of multiple lightweight processes called *threads*. Threads communicate through the shared memory. This software model mirrors the hardware that is (expected to be) widely available. By assuming that there is at most one thread per process (or by ignoring the parallelism within one process), we get the usual model of a distributed system. By restricting our attention to a single heavyweight process, we get the usual model of a parallel system. We expect the system to have aspects of distributed objects. The main reason is the logical simplicity of the distributed object model. A distributed program is more object-oriented because data in a remote object can be accessed only through an explicit message (or a remote procedure call). The object orientation promotes reusability as well as design simplicity. Furthermore, these object would be multithreaded because threads are useful for implementing efficient objects. For many applications such as servers, it is useful to have a large shared data structure. It is a programming burden and inefficient to split the data structure across multiple heavyweight processes.

1.3 Overview of the Book

This book is intended for a one-semester advanced undergraduate or introductory graduate course on concurrent and distributed systems. It can also be used as a supplementary book in a course on operating systems or distributed operating systems. For an undergraduate course, the instructor may skip the chapters on consistency conditions, wait-free synchronization, synchronizers, recovery, and self-stabilization without any loss of continuity.

Chapter 1 provides the motivation for parallel and distributed systems. It compares advantages of distributed systems with those of parallel systems. It gives the defining characteristics of parallel and distributed systems and the fundamental difficulties in designing algorithms for such systems. It also introduces basic constructs of starting threads in Java.

Chapters 2–5 deal with multithreaded programming. Chapter 2 discusses the mutual exclusion problem in shared memory systems. This provides motivation to students for various synchronization primitives discussed in Chapter 3. Chapter 3 exposes students to multithreaded programming. For a graduate course, Chapters 2 and 3 can be assigned for self-study. Chapter 4 describes various consistency conditions on concurrent executions that a system can provide to the programmers. Chapter 5 discusses a method of synchronization which does not use locks. Chapters 4 and 5 may be skipped in an undergraduate course.

Chapter 6 discusses distributed programming based on sockets as well as remote method invocations. It also provides a layer for distributed programming used by the programs in later chapters. This chapter is a prerequisite to understanding programs described in later chapters.

Chapter 7 provides the fundamental issues in distributed programming. It discusses models of a distributed system and a distributed computation. It describes the *interleaving model* that totally orders all the events in the system, and the *happened before model* that totally orders all the events on a single process. It also discusses mechanisms called *clocks* used to timestamp events in a distributed computation such that order information between events can be determined with these clocks. This chapter is fundamental to distributed systems and should be read before all later chapters.

Chapter 8 discusses one of the most studied problems in distributed systems—mutual exclusion. This chapter provides the interface `Lock` and discusses various algorithms to implement this interface. `Lock` is used for coordinating resources in distributed systems.

Chapter 9 discusses the abstraction called `Camera` that can be used to compute a consistent snapshot of a distributed system. We describe Chandy and Lamport's algorithm in which the receiver is responsible for recording the state of a channel as well as a variant of that algorithm in which the sender records the state of the channel. These algorithms can also be used for detecting stable global properties—properties that remain true once they become true.

Chapters 10 and 11 discuss the abstraction called `Sensor` that can be used to evaluate global properties in a distributed system. Chapter 10 describes algorithms for detecting conjunctive predicates in which the global predicate is simply a conjunction of local predicates. Chapter 11 describe algorithms for termination and deadlock detection. Although termination and deadlock can be detected using techniques described in Chapters 9 and 10, we devote a separate chapter for termination and deadlock detection because these algorithms are more efficient than those used to detect general global properties. They also illustrate techniques in designing distributed algorithms.

Chapter 12 describes methods to provide messaging layer with stronger properties than provided by the Transmission Control Protocol (TCP). We discuss the causal ordering of messages, the synchronous and the total ordering of messages.

Chapter 13 discusses two abstractions in a distributed system—`Election` and `GlobalFunction`. We discuss election in ring-based systems as well as in general graphs. Once a leader is elected, we show that a global function can be computed easily via a convergecast and a broadcast.

Chapter 14 discusses synchronizers, a method to abstract out asynchrony in the system. A synchronizer allows a synchronous algorithm to be simulated on top of an asynchronous system. We apply synchronizers to compute the breadth-first search (BFS) tree in an asynchronous network.

Chapters 1–14 assume that there are no faults in the system. The rest of the book deals with techniques for handling various kinds of faults.

Chapter 15 analyzes the possibility (or impossibility) of solving problems in the presence of various types of faults. It includes the fundamental impossibility result of Fischer, Lynch, and Paterson that shows that consensus is impossible to solve in the presence of even one unannounced failure in an asynchronous system.

It also shows that the consensus problem can be solved in a synchronous environment under crash and Byzantine faults. It also discusses the ability to solve problems in the absence of reliable communication. The two-generals problem shows that agreement on a bit (gaining common knowledge) is impossible in a distributed system.

Chapter 16 describes the notion of a transaction and various algorithms used in implementing transactions.

Chapter 17 discusses methods of recovering from failures. It includes both checkpointing and message-logging techniques.

Finally, Chapter 18 discusses self-stabilizing systems. We discuss solutions of the mutual exclusion problem when the state of any of the processors may change arbitrarily because of a fault. We show that it is possible to design algorithms that guarantee that the system converges to a legal state in a finite number of moves irrespective of the system execution. We also discuss self-stabilizing algorithms for maintaining a spanning tree in a network.

There are numerous starred and unstarred problems at the end of each chapter. A student is expected to solve unstarred problems with little effort. The starred problems may require the student to spend more effort and are appropriate only for graduate courses.

1.4 Characteristics of Parallel and Distributed Systems

Recall that we distinguish between parallel and distributed systems on the basis of shared memory. A distributed system is characterized by absence of shared memory. Therefore, in a distributed system it is impossible for any one processor to know the global state of the system. As a result, it is difficult to observe any global property of the system. We will later see how efficient algorithms can be developed for evaluating a suitably restricted set of global properties.

A parallel or a distributed system may be *tightly coupled* or *loosely coupled* depending on whether multiple processors work in a lock step manner. The absence of a shared clock results in a loosely coupled system. In a geographically distributed system, it is impossible to synchronize the clocks of different processors precisely because of uncertainty in communication delays between them. As a result, it is rare to use physical clocks for synchronization in distributed systems. In this book we will see how the concept of causality is used instead of time to tackle this problem. In a parallel system, although a shared clock can be simulated, designing a system based on a tightly coupled architecture is rarely a good idea, due to loss of performance because of synchronization. In this book, we will assume that systems are loosely coupled.

Distributed systems can further be classified into synchronous and asynchronous systems. A distributed system is *asynchronous* if there is no upper bound on the message communication time. Assuming asynchrony leads to most general solutions to various problems. We will see many examples in this book. However, things get difficult in asynchronous systems when processors or links can fail. In an asynchronous distributed system it is impossible to distinguish between a slow processor and a failed processor. This leads to difficulties in developing algorithms for consensus, election, and other important problems in distributed computing. We will describe these difficulties and also show algorithms that work under faults in synchronous systems.

1.5 Design Goals

The experience in large parallel and distributed software systems has shown that their design should take the following concepts into consideration [TvS02]:

- *Fault tolerance*: The software system should mask the failure of one or more components in the system, including processors, memory, and network links. This generally requires redundancy, which may be expensive depending on the degree of fault tolerance. Therefore, cost–benefit analysis is required to determine an appropriate level of fault tolerance.

- *Transparency*: The system should be as user-friendly as possible. This requires that the user not have to deal with unnecessary details. For example, in a heterogeneous distributed system the differences in the internal representation of data (such as the little endian format versus the big endian format for integers) should be hidden from the user, a concept called *access transparency*. Similarly, the use of a resource by a user should not require the user to know where it is located (*location transparency*), whether it is replicated (*replication transparency*), whether it is shared (*concurrency transparency*), or whether it is in volatile memory or hard disk (*persistence transparency*).

- *Flexibility*: The system should be able to interact with a large number of other systems and services. This requires that the system adhere to a fixed set of rules for syntax and semantics, preferably a standard, for interaction. This is often facilitated by specification of services provided by the system through an *interface definition language*. Another form of flexibility can be given to the user by a separation between *policy* and *mechanism*. For example, in the context of Web caching, the mechanism refers to the implementation for storing the Web pages locally. The policy refers to the high-level decisions such as size of the cache, which pages are to be cached, and how long those pages should remain in the cache. Such questions may be answered better by the user and therefore it is better for users to build their own caching policy on top of the caching mechanism provided. By designing the system as one monolithic component, we lose the flexibility of using different policies with different users.

- *Scalability*: If the system is not designed to be scalable, then it may have unsatisfactory performance when the number of users or the resources increase. For example, a distributed system with a single server may become overloaded when the number of clients requesting the service from the server increases. Generally, the system is either completely decentralized using distributed algorithms or partially decentralized using a hierarchy of servers.

1.6 Specification of Processes and Tasks

In this book we cover the programming concepts for shared memory-based languages and distributed languages. It should be noted that the issues of concurrency arise even on a single CPU computer where a system may be organized as a collection of cooperating processes. In fact, the issues of synchronization and deadlock have roots in the development of early operating systems. For this reason, we will refer to constructs described in this section as *concurrent* programming.

Before we embark on concurrent programming constructs, it is necessary to understand the distinction between a *program* and a *process*. A computer program is simply a set of instructions in a high-level or a machine-level language. It is only when we execute a program that we get one or more *processes*. When the program is sequential, it results in a single process, and when concurrent—multiple processes. A process can be viewed as consisting of three segments in the memory: code, data and execution stack. The *code* is the machine instructions in the memory which the process executes. The *data* consists of memory used by static global variables and runtime allocated memory (heap) used by the program. The *stack* consists of local variables and the activation records of function calls. Every process has its own stack. When processes share the address space, namely, code and data, then they are called *lightweight processes* or *threads*. Figure 1.3 shows four threads. All threads share the address space but have their own local stack. When process has its own code and data, it is called a *heavyweight process*, or simply a process. Heavyweight processes may share data through files or by sending explicit messages to each other.

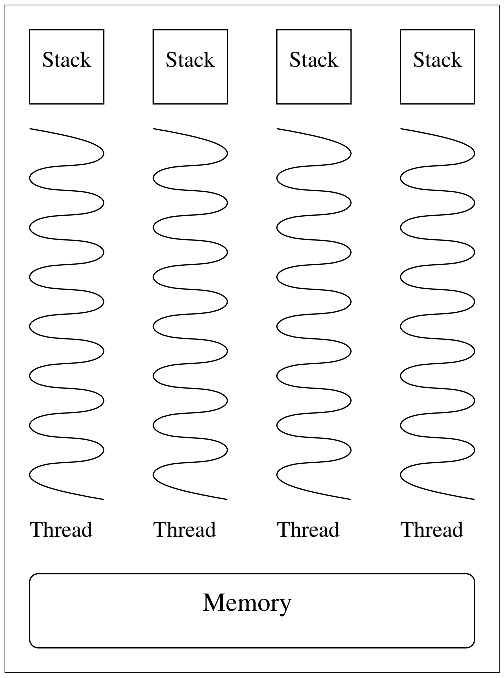

Figure 1.3: A process with four threads

Any programming language that supports concurrent programming must have a way to specify the process structure, and how various processes communicate and synchronize with each other. There are many ways a program may specify the process structure or creation of new processes. We look at the most popular ones. In UNIX, processes are organized as a tree of processes with each process identified using a unique process id (pid). UNIX provides system calls *fork* and *wait* for creation and synchronization of processes. When a process executes a *fork* call, a child process is created with a copy of the address space of the parent process. The only difference between the parent process and the child process is the value of the return code for the *fork*. The parent process gets the pid of the child process as the return code, and the child process gets the value 0 as shown in the following example.

```
pid = fork();
if (pid == 0) {
   // child process
   cout << "child process";
}
else {
   // parent process
   cout << "parent process";
}
```

The *wait* call is used for the parent process to wait for termination of the child process. A process terminates when it executes the last instruction in the code or makes an explicit call to the system call *exit*. When a child process terminates, the parent process, if waiting, is awakened and the pid of the child process is returned for the wait call. In this way, the parent process can determine which of its child processes terminated.

Frequently, the child process makes a call to the *execve* system call, which loads a binary file into memory and starts execution of that file.

Another programming construct for launching parallel tasks is *cobegin-coend* (also called *parbegin-parend*). Its syntax is given below:

$$\textit{cobegin } S_1 \mid\mid S_2 \textit{ coend}$$

This construct says that S_1 and S_2 must be executed in parallel. Further, if one of them finishes earlier than the other, it should wait for the other one to finish. Combining the cobegin-coend with the sequencing, or the series operator, semicolon (;), we can create any series-parallel task structure. For example,

$$S_0;\ \textit{cobegin } S_1 \mid\mid S_2 \textit{ coend};\ S_3$$

starts off with one process that executes S_0. When S_0 is finished, we have two processes (or threads) that execute S_1 and S_2 in parallel. When both the statements are done, only then S_3 is started.

Yet another method for specification of concurrency is to explicitly create thread objects. For example, in Java there is a predefined class called `Thread`. One can extend the class `Thread`, override the method `run` and then call `start()` to launch the thread. For example, a thread for printing "Hello World" can be launched as shown in Figure 1.4.

```
public class HelloWorldThread extends Thread {
    public void run() {
        System.out.println("Hello World");
    }
    public static void main(String[] args) {
        HelloWorldThread t = new HelloWorldThread();
        t.start();
    }
}
```

Figure 1.4: HelloWorldThread.java

1.6.1 Runnable Interface

In the `HelloWorld` example, the class `HelloWorldThread` needed to inherit methods only from the class `Thread`. What if we wanted to extend a class, say, `Foo`, but also make the objects of the new class run as separate thread? Since Java does not have multiple inheritance, we could not simply extend both `Foo` and the `Thread` class. To solve this problem, Java provides an interface called `Runnable` with the following single method:

```
public void run()
```

To design a runnable class `FooBar` that extends `Foo`, we proceed as shown in Figure 1.5. The class `FooBar` implements the `Runnable` interface. The `main` function creates a runnable object `f1` of type `FooBar`. Now we can create a thread `t1` by passing the runnable object `f1` as an argument to the constructor for `Thread`. This thread can then be started by invoking the `start` method. The program creates two threads in this manner. Each of the threads prints out the string `getName()` inherited from the class `Foo`.

```
class Foo {
    String name;
    public Foo(String s) {
        name = s;
    }
    public void setName(String s) {
        name = s;
    }
    public String getName() {
        return name;
    }
}
class FooBar extends Foo implements Runnable {
    public FooBar(String s) {
        super(s);
    }
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.println(getName() + ": Hello World");
    }
    public static void main(String[] args) {
        FooBar f1 = new FooBar("Romeo");
        Thread t1 = new Thread(f1);
        t1.start();
        FooBar f2 = new FooBar("Juliet");
        Thread t2 = new Thread(f2);
        t2.start();
    }
}
```

Figure 1.5: FooBar.java

1.6.2 Callable Interface

A `runnable` object cannot return a result or throw an exception. Sometimes it is convenient to create a task that can return a result or throw an exception. A task that implements the interface `Callable` must provide the method

```
public V calls() throws Exception
```

where `V` is the return type of the task.

1.6.3 Join Construct in Java

We have seen that we can use `start()` to start a thread. The following example shows how a thread can wait for other thread to finish execution via the `join` mechanism. We write a program in Java to compute the nth Fibonacci number F_n using the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

for $n \geq 2$. The base cases are

$$F_0 = 1$$

and

$$F_1 = 1$$

To compute F_n, the `run` method forks two threads that compute F_{n-1} and F_{n-2} recursively. The main thread waits for these two threads to finish their computation using `join`. The complete program is shown in Figure 1.6.

1.6.4 Futures

An alternative method to spawn an asynchronous computation and then later wait for its result is based on the notion of `Future`. A `Future` denotes the result of a task which can be obtained using the method `get`. The `get` method waits for the task to finish before returning the result. We illustrate the use of `Future` in conjunction with `Callable` by rewriting Fibonacci program (shown in Figure 1.7) using these concepts.

We also illustrate in this example the use of `ThreadPools` and `ExecutorService`. Creating a thread for every task and destroying it after the completion of the task results in excessive overhead. It is better to create a pool of threads such that these threads are reused for computing multiple tasks. An `ExecutorService` provides a high-level abstraction for user to submit tasks for future executions without worrying about explicit creation and scheduling of threads. The public class `Executors` is used to create `ExecutorService` with desired thread configurations. The method `newCachedThreadPool()` creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. The method `newSingleThreadExecutor()` creates an Executor that uses a single worker thread. Another useful method is `newFixedThreadPool(int n)` to create a thread pool that reuses a fixed number of threads.

We use `shutdown()` method in ExecutorService to stop all threads in an orderly fashion.

1.6.5 ForkJoinPool

In some applications, `ExecutorService` may not be able to reuse thread for computing tasks when existing threads are waiting for results of subtasks. For example, when a thread requires a result from another task

```java
public class Fibonacci extends Thread {
    int n;
    int result;
    public Fibonacci(int n) {
        this.n = n;
    }
    public void run() {
        if ((n == 0)||(n == 1 )) result = 1;
        else {
            Fibonacci f1 = new Fibonacci(n-1);
            Fibonacci f2 = new Fibonacci(n-2);
            f1.start();
            f2.start();
            try {
                f1.join();
                f2.join();
            } catch (InterruptedException e){};
            result = f1.getResult() + f2.getResult();
        }
    }
    public int getResult(){
        return result;
    }
    public static void main(String[] args) {
        Fibonacci f1 = new Fibonacci(Integer.parseInt(args[0]));
        f1.start();
        try {
            f1.join();
        } catch (InterruptedException e){};
        System.out.println("Answer is " + f1.getResult());
    }
}
```

Figure 1.6: Fibonacci.java

```java
import java.util.concurrent.*;
class Fibonacci2 implements Callable<Integer> {
  public static ExecutorService threadPool = Executors.newCachedThreadPool();
  int n;
  public Fibonacci2(int n) {
    this.n = n;
  }
  public Integer call() {
    try {
       if ((n == 0)||(n == 1 )) return 1;
       else {
        Future<Integer> f1 = threadPool.submit(new Fibonacci2(n-1));
        Future<Integer> f2 = threadPool.submit(new Fibonacci2(n-2));
        return f1.get() + f2.get();
      }
    } catch (Exception e) { System.err.println (e); return 1;}
  }

  public static void main(String[] args) {
    try {
        ExecutorService es = Executors.newSingleThreadExecutor();
        Fibonacci2 f = new Fibonacci2(Integer.parseInt(args[0]));
        Future<Integer> f1 = es.submit(f);
        System.out.println("Answer is " + f1.get());
        es.shutdown ();
        f.threadPool.shutdown();
    } catch (Exception e) { System.err.println (e); }
  }
}
```

Figure 1.7: Fibonacci2.java

as in `Fibonacci2` and uses `f.get()` for some future `f`, it goes into a waiting state. Can this thread be used to compute other tasks in this waiting state? This is the idea behind `ForkJoinPool` and `forkjoin` tasks. It uses the idea of *work-stealing*. If a thread is in the waiting state (idle), it starts executing another task.

To assign a task to the `ForkJoinPool`, we can either wait for the result or arrange for *asynchronous* execution and obtain a future. To invoke a task and obtain the result we use the method `invoke`. For example, to invoke the task `f` on a threadpool `pool`, we can use `pool.invoke(f)`. However, if we want to arrange for an asynchronous execution and obtain a future then we use the method `submit`, i.e., `pool.submit(f)`. If we are assigning subtask `f` from a `ForkJoin` task then we use `f.invoke()` for a synchronous call and `f.fork()` for an asynchronous call.

Figure 1.8 shows how `ForkJoinPool` can be used for computing Fibonacci numbers. We use `RecursiveTask<T>` to define a task that can be submitted to the thread pool and returns a value of type `T`. Just as a `Callable` task overrides the method `call`, a `RecursiveTask` overrides the method `compute`. In addition, it can `fork` a subtask and then wait for its result using `join`.

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

class Fibonacci3 extends RecursiveTask<Integer> {
   final int n;
   Fibonacci3(int n) { this.n = n; }
   protected Integer compute() {
      if ((n == 0)||(n == 1 )) return 1;
      Fibonacci3 f1 = new Fibonacci3(n - 1);
      f1.fork();
      Fibonacci3 f2 = new Fibonacci3(n - 2);
      return f2.compute() + f1.join();
   }
 public static void main(String[] args) {
         int processors = Runtime.getRuntime().availableProcessors();
         System.out.println("Number of processors: " + processors);
         Fibonacci3 f = new Fibonacci3(Integer.parseInt(args[0]));
         ForkJoinPool pool = new ForkJoinPool(processors);
         int result = pool.invoke(f);
         System.out.println("Result: " + result);
  }
 }
```

Figure 1.8: Fibonacci3.java

1.6.6 Thread Scheduling

In the `FooBar` example, we had two threads. The same Java program will work for a single-CPU machine as well as for a multiprocessor machine. In a single-CPU machine, if both threads are runnable, which one would be picked by the system to run? The answer to this question depends on the *priority* and the *scheduling policy* of the system.

The programmer may change the priority of threads using `setPriority` and determine the current priority by `getPriority`. `MIN_PRIORITY` and `MAX_PRIORITY` are integer constants defined in the `Thread` class. The method `setPriority` can use a value only between these two constants. By default, a thread has the priority `NORM_PRIORITY`.

A Java thread that is running may block by calling *sleep*, *wait*, or any system function that is blocking (these calls will be described later). When this happens, a highest-priority runnable thread is picked for execution. When the highest-priority thread is running, it may still be suspended when its time slice is over. Another thread at the same priority level may then be allowed to run.

1.7 Limits on Parallelism: Amdahl's Law

Suppose that we have a large task that needs to be executed and n cores are available to execute that task. Suppose that the task takes T_s units of time to execute on a single core. It may seem that we can decrease the overall execution time of the task by a factor that is propotional to n. However, for many tasks this is impossible because of dependency between subtasks. Suppose that p is the fraction of the task that can be executed in parallel and $1-p$ is the remaining fraction that must be executed sequentially. We can compute the limits on speedup for this task by a formula called Amdahl's Law. Let us determine T_p, the time for completion when we use parallelism.

$$T_p \geq (1-p) * T_s + \frac{p * T_s}{n}$$

Hence, the speedup is at most

$$T_s/T_p \leq \frac{1}{(1-p)+p/n}$$

To get a feel for this expression, suppose that an application has at least 20% part that is inherently sequential. The speedup is bounded above by

$$\frac{1}{(0.2+0.8/n)}$$

Therefore, we can never get a speedup of greater than 5 even if we had million cores available to us.

A lesson from Amdahl's law is that we must design our programs so that there is as small a sequential component as possible.

1.8 Problems

1.1. Give advantages and disadvantages of a parallel programming model over a distributed system (message-based) model.

1.2. Write a Java class that allows parallel search in an array of integer. It provides the following `static` method:

```
public static int parallelSearch(int x, int[] A, int numThreads)
```

This method creates as many threads as specified by `numThreads`, divides the array `A` into that many parts, and gives each thread a part of the array to search for `x` sequentially. If any thread finds `x`, then it returns an index `i` such that `A[i] = x`. Otherwise, the method returns -1.

1.3. Consider the class shown below.

```
class Schedule {
    static int x = 0;
    static int y = 0;
    public static int op1(){x = 1; return y;}
    public static int op2(){y = 2; return 3*x;}
}
```

If one thread calls `op1` and the other thread calls `op2`, then what values may be returned by `op1` and `op2`?

1.4. Write a multithreaded program in Java that sorts an array using recursive merge sort. The main thread forks two threads to sort the two halves of arrays, which are then merged.

1.5. Write a program in Java that uses two threads to search for a given element in a doubly linked list. One thread traverses the list in the forward direction and the other, in the backward direction.

1.9 Bibliographic Remarks

There are many books available on distributed systems. The reader is referred to books by Attiya and Welch [AW98], Barbosa [Bar96], Chandy and Misra [CM89], Garg [Gar96, Gar02], Lynch [Lyn96], Raynal [Ray88], and Tel [Tel94] for the range of topics in distributed algorithms. Couloris, Dollimore and Kindberg [CDK94], and Chow and Johnson [CJ97] cover some other practical aspects of distributed systems such as distributed file systems, which are not covered in this book. Goscinski [Gos91] and Singhal and Shivaratri [SS94] cover concepts in distributed operating systems. The book edited by Yang and Marsland [YM94] includes many papers that deal with global time and state in distributed systems. The book edited by Mullender [SM94] covers many other topics such as protection, fault tolerance, and real-time communications.

There are many books available for concurrent computing in Java as well. The reader is referred to the books by Farley [Far98], Hartley [Har98] and Lea [Lea99] as examples. These books do not discuss distributed algorithms.