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.

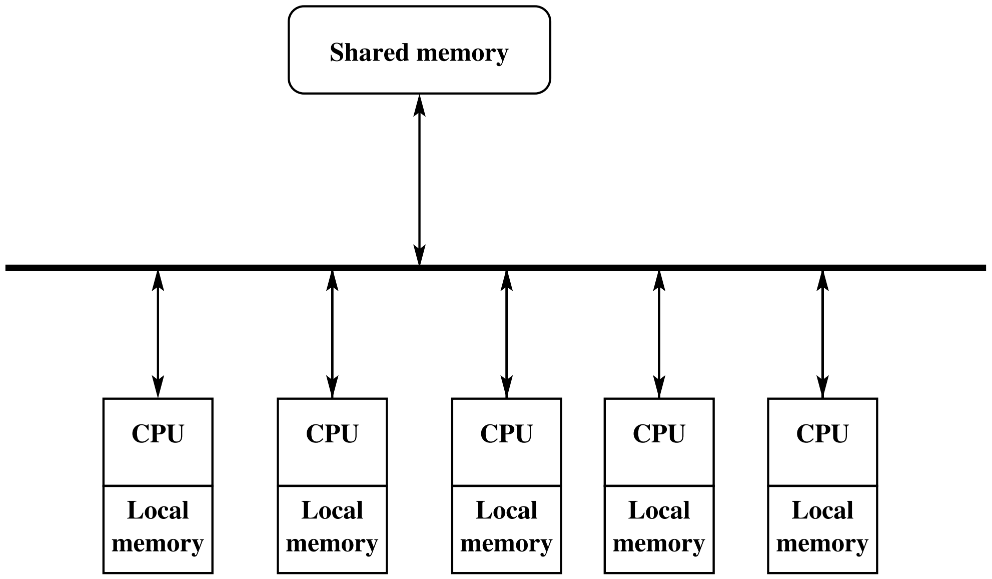

Figure 1.1: A parallel system

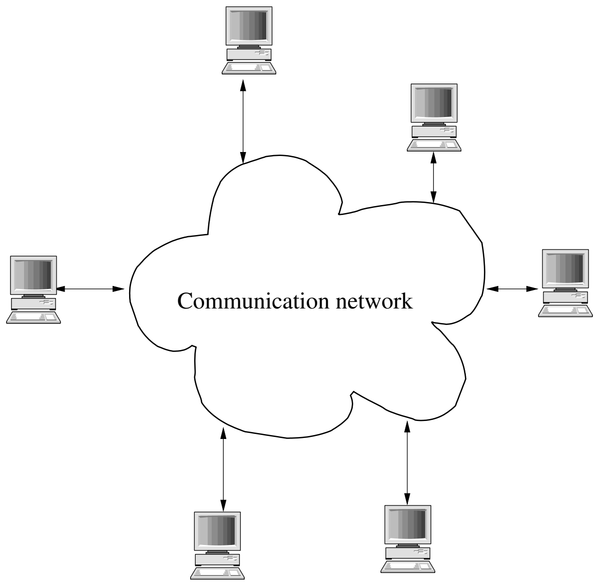

Figure 1.2: A distributed system

- *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.

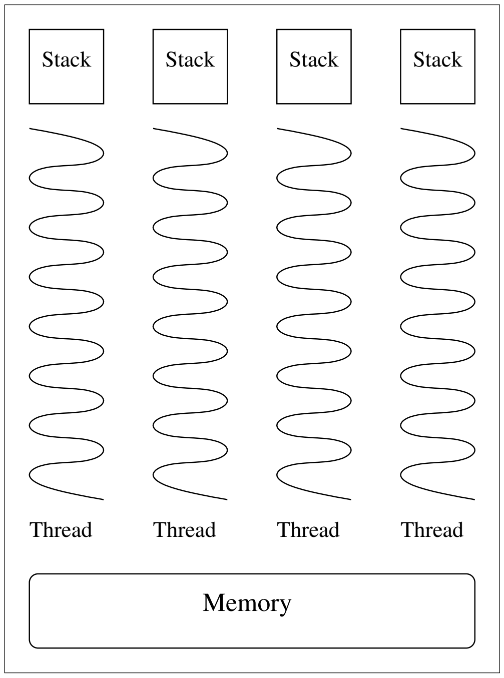

Figure 1.3: A process with four threads

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.

```java
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`.

```java
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 $n$th 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

```
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

```
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`.

```java
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.

# Chapter 2

# Mutual Exclusion Problem

## 2.1 Introduction

When processes share data, it is important to synchronize their access to the data so that updates are not lost as a result of concurrent accesses and the data are not corrupted. This can be seen from the following example. Assume that the initial value of a shared variable $x$ is 0 and that there are two processes, $P_0$ and $P_1$ such that each one of them increments $x$ by the following statement in some high-level programming language:

$$x = x + 1$$

It is natural for the programmer to assume that the final value of $x$ is 2 after both the processes have executed. However, this may not happen if the programmer does not ensure that $x = x + 1$ is executed atomically. The statement $x = x + 1$ may compile into the machine-level code of the form

```
LD R, x    ; load register R from x
INC R      ; increment register R
ST R, x    ; store register R to x
```

Now the execution of $P_0$ and $P_1$ may get interleaved as follows:

```
P0: LD R, x                  ; load register R from x
P0: INC R                    ; increment register R
                P1: LD R, x  ; load register R from x
                P1: INC R    ; increment register R
P0: ST R,x                   ; store register R to x
                P1: ST R,x   ; store register R to x
```

Thus both processes load the value 0 into their registers and finally store 1 into $x$ resulting in the "lost update" problem.

To avoid this problem, the statement $x = x + 1$ should be executed `atomically`. A section of the code that needs to be executed atomically is also called a *critical region* or a *critical section*. The problem of ensuring that a critical section is executed atomically is called the *mutual exclusion problem*. This is one of the most fundamental problems in concurrent computing and we will study it in detail.

The mutual exclusion problem can be abstracted as follows. We are required to implement the interface shown in Figure 2.1. A process that wants to enter the critical section (CS) makes a call to `requestCS` with its own identifier as the argument. The process or the thread that makes this call returns from this method only when it has the exclusive access to the critical section. When the process has finished accessing the critical section, it makes a call to the method `releaseCS`.

```
public interface Lock {
    public void requestCS(int pid); //may block
    public void releaseCS(int pid);
}
```

Figure 2.1: Interface for accessing the critical section

The entry protocol given by the method `requestCS` and the exit protocol given by the method `releaseCS` should be such that the mutual exclusion is not violated.

To test the Lock, we use the program shown in Figure 2.2. This program tests the Bakery algorithm that will be presented later. The user of the program may test a different algorithm for a lock implementation by invoking the constructor of that lock implementation. The program launches $N$ threads as specified by `arg[0]`. Each thread is an object of the class `MyThread`. Let us now look at the class `MyThread`. This class has two methods, `nonCriticalSection` and `CriticalSection`, and it overrides the `run` method of the `Thread` class as follows. Each thread repeatedly enters the critical section. After exiting from the critical section it spends an undetermined amount of time in the noncritical section of the code. In our example, we simply use a random number to sleep in the critical and the noncritical sections.

Let us now look at some possible protocols, one may attempt, to solve the mutual exclusion problem. For simplicity we first assume that there are only two processes, $P_0$ and $P_1$.

## 2.2 Peterson's Algorithm

Our first attempt would be to use a shared boolean variable `openDoor` initialized to `true`. The entry protocol would be to wait for `openDoor` to be true. If it is true, then a process can enter the critical section after setting it to `false`. On exit, the process resets it to `true`. This algorithm is shown in Figure 2.3.

This attempt does not work because the testing of `openDoor` and setting it to `false` is not done atomically. Conceivably, one process might check for the `openDoor` and go past the `while` statement in Figure 2.3. However, before that process could set `openDoor` to `false`, the other process starts executing. The other process now checks for the value of `openDoor` and also gets out of busy wait. Both the processes now can set `openDoor` to false and enter the critical section. Thus, mutual exclusion is violated.

In the attempt described above, the shared variable did not record who set the `openDoor` to false. One may try to fix this problem by keeping two shared variables, `wantCS[0]` and `wantCS[1]`, as shown in Figure 2.4. Every process $P_i$ first sets its own `wantCS` bit to true at line 3 and then waits until the `wantCS` for the other process is false at line 4. We have used $1 - i$ to get the process identifier of the other process when there are only two processes - $P_0$ and $P_1$. To release the critical section, $P_i$ simply resets its `wantCS` bit to false. Unfortunately, this attempt also does not work. Both processes could set their `wantCS` to true and then indefinitely loop, waiting for the other process to set its `wantCS` false.

Yet another attempt to fix the problem is shown in Figure 2.5. This attempt is based on evaluating the value of a variable `turn`. A process waits for its turn to enter the critical section. On exiting the critical section, it sets `turn` to `1-i`.

This protocol does guarantee *mutual exclusion*. It also guarantees that if both processes are trying to

```
import java.util.Random;
public class MyThread extends Thread {
    int myId;
    Lock lock;
    Random r = new Random();
    public MyThread(int id, Lock lock) {
        myId = id;
        this.lock = lock;
    }
    void nonCriticalSection() {
        System.out.println(myId + " is not in CS");
        Util.mySleep(r.nextInt(1000));
    }
    void CriticalSection() {
        System.out.println(myId + " is in CS *****");
        // critical section code
        Util.mySleep(r.nextInt(1000));
    }
    public void run() {
        while (true) {
            lock.requestCS(myId);
            CriticalSection();
            lock.releaseCS(myId);
            nonCriticalSection();
        }
    }
    public static void main(String[] args) throws Exception {
        MyThread t[];
        int N = Integer.parseInt(args[0]);
        t = new MyThread[N];
        Lock lock = new Bakery(N);//or any other mutex algorithm
        for (int i = 0; i < N; i++) {
            t[i] = new MyThread(i, lock);
            t[i].start();
        }
    }
}
```

Figure 2.2: A program to test mutual exclusion

```
class Attempt1 implements Lock {
    boolean openDoor = true;
    public void requestCS(int i) {
        while (!openDoor) ; // busy wait
        openDoor = false;
    }
    public void releaseCS(int i) {
        openDoor = true;
    }
}
```

Figure 2.3: An attempt that violates mutual exclusion

```
class Attempt2 implements Lock {
    boolean wantCS[] = {false, false};
    public void requestCS(int i) { // entry protocol
        wantCS[i] = true;   //declare intent
        while (wantCS[1 - i]) ; // busy wait
    }
    public void releaseCS(int i) {
        wantCS[i] = false;
    }
}
```

Figure 2.4: An attempt that can deadlock

```
class Attempt3 implements Lock {
    int turn = 0;
    public void requestCS(int i) {
        while (turn == 1 - i) ;
    }
    public void releaseCS(int i) {
        turn = 1 - i;
    }
}
```

Figure 2.5: An attempt with strict alternation

enter the critical section, then one of them will succeed. However, it suffers from another problem. In this protocol, both processes have to alternate with each other for getting the critical section. Thus, after process $P_0$ exits from the critical section it cannot enter the critical section again until process $P_1$ has entered the critical section. If process $P_1$ is not interested in the critical section, then process $P_0$ is simply stuck waiting for process $P_1$. This is not desirable.

By combining the previous two approaches, however, we get Peterson's algorithm for the mutual exclusion problem in a two-process system. In this protocol, shown in Figure 2.6, we maintain two flags, `wantCS[0]` and `wantCS[1]`, as in `Attempt2`, and the `turn` variable as in `Attempt3`. To request the critical section, process $P_i$ sets its `wantCS` flag to true at line 6 and then sets the `turn` to the other process $P_j$ at line 7. After that, it waits at line 8 so long as the following condition is true:

```
(wantCS[j] && (turn == j))
```

Thus a process enters the critical section only if either it is its turn to do so or if the other process is not interested in the critical section.

To release the critical section, $P_i$ simply resets the flag `wantCS[i]` at line 11. This allows $P_j$ to enter the critical section by making the condition for its **while** loop false.

Intuitively, Peterson's algorithm uses the order of updates to `turn` to resolve the contention. If both processes are interested in the critical section, then the process that updated `turn` last, loses and is required to wait.

We show that Peterson's algorithm satisfies the following desirable properties:

1. *Mutual exclusion*: Two processes cannot be in the critical section at the same time.

```
 1 class PetersonAlgorithm implements Lock {
 2     boolean wantCS[] = {false, false};
 3     int turn = 1;
 4     public void requestCS(int i) {
 5         int j = 1 - i;
 6         wantCS[i] = true;
 7         turn = j;
 8         while (wantCS[j] && (turn == j)) ;
 9     }
10     public void releaseCS(int i) {
11         wantCS[i] = false;
12     }
13 }
```

Figure 2.6: Peterson's algorithm for mutual exclusion

2. *Progress*: If one or more processes are trying to enter the critical section and there is no process inside the critical section, then at least one of the processes succeeds in entering the critical section.

3. *Starvation-freedom*: If a process is trying to enter the critical section, then it eventually succeeds in doing so.

We first prove that mutual exclusion is satisfied by Peterson's algorithm by the method of contradiction. Suppose, if possible, both processes $P_0$ and $P_1$ are in the critical section for some execution. Each of the processes $P_i$ must have set the variable *turn* to $1 - i$. Without loss of generality, assume that $P_1$ was the last process to set the variable *turn*. This means that the value of *turn* was 0 when $P_1$ checked the entry condition for the critical section. Since $P_1$ entered the critical section in spite of *turn* being 0, it must have read $wantCS[0]$ to be false. Therefore, we have the following sequence of events:
$P_0$ sets *turn* to 1, $P_1$ sets turn to 0, $P_1$ reads $wantCS[0]$ as false. However, $P_0$ sets the *turn* variable to 1 after setting $wantCS[0]$ to true. Since there are no other writes to $wantCS[0]$, $P_1$ reading it as false gives us the desired contradiction.

We give a second proof of mutual exclusion due to Dijkstra. This proof does not reason on the sequence of events; it uses an assertional proof. For the purposes of this proof, we introduce auxiliary variables *trying*[0] and *trying*[1]. Whenever $P_0$ reaches line 8, *trying*[0] becomes true. Whenever $P_0$ reaches line 9, i.e., it has acquired permission to enter the critical section, *trying*[0] becomes false.

Consider the predicate $H(0)$ defined as

$$H(0) \equiv wantCS[0] \wedge [(turn = 1) \vee ((turn = 0) \wedge trying[1])]$$

Assuming that there is no interference from $P_1$ it is clear that $P_0$ makes this predicate true after executing $(turn = 1)$ at line 7. Similarly, the predicate

$$H(1) \equiv wantCS[1] \wedge [(turn = 0) \vee ((turn = 1) \wedge trying[0])]$$

is true for $P_1$ after it executes line 7.

We now take care of interference between processes. It is sufficient to show that $P_0$ cannot falsify $H(1)$. From symmetry, it will follow that $P_1$ cannot falsify $H(0)$.

The first conjunct of $H(1)$ is not falsified by $P_0$ because it never updates the variable $wantCS[1]$. It only reads the value of $wantCS[1]$. Now, let us look at the second conjunct. Whenever $P_0$ falsifies $(turn = 0)$ by setting $turn = 1$, it makes $(turn = 1) \wedge trying[0]$ true. So, the only case left is falsification of $(turn =$

$1) \wedge trying[0]$. Since $wantCS[1]$ is also true, we look at falsification of $(turn = 1) \wedge trying[0] \wedge wantCS[1]$. $P_0$ can falsify this only by setting $trying[0]$ to false (i.e., by acquiring the permission to enter the critical section). But, $(turn = 1) \wedge (wantCS[1])$ implies that the condition for the while statement at line 8 is true, so $P_0$ cannot exit the while loop.

Now, it is easy to show mutual exclusion. If $P_0$ and $P_1$ are in critical section, we get $\neg trying[0] \wedge H(0) \wedge \neg trying[1] \wedge H(1)$, which implies $(turn = 0) \wedge (turn = 1)$, a contradiction.

It is easy to see that the algorithm satisfies the progress property. If both the processes are forever checking the entry protocol in the while loop, then we get

$$wantCS[1] \wedge (turn = 1) \wedge wantsCS[0] \wedge (turn = 0)$$

which is clearly false because $(turn = 1) \wedge (turn = 0)$ is false.

The proof of freedom from starvation is left as an exercise. The reader can also verify that Peterson's algorithm does not require strict alternation of the critical sections—a process can repeatedly use the critical section if the other process is not interested in it.

## 2.3 Filter Algorithm

We now extend Peterson's algorithm for more than two processes. The first difficulty that we face is that in Peterson's algorithm, a process would set the *turn* variable to that of the other process before checking the entry condition. Now that there are more than two processes, it is not clear how to set the *turn* variable. Therefore, instead of the *turn* variable, we use the variable *last* which stores the pid of the last process that wrote to that variable. Now a process can use *last* to wait whenever there is a conflict. In a system of two processes, one of them will wait and the other one can enter the critical section. If there are $N > 2$ processes, then one of them will wait and the remaining can get through. Note that we have managed to reduce the number of contending processes from $N$ to $N - 1$. By applying this idea $N - 1$ times, we can reduce the number of active processes from $N$ to 1.

Figure 2.7 shows the Filter Algorithm. We have $N - 1$ gates numbered $1 \ldots N - 1$. For each thread $i$, variable $gate[i]$ stores the gate that thread is trying to enter. Initially, $gate[i]$ is 0 for each $i$. For each gate $k$, we use $last[k]$ to store the last process that tries to enter the gate (i.e. writes on the variable $last[k]$). In `requestCS` method, $P_i$ goes through a gate $k$ as follows. $P_i$ checks whether there is any process $P_j$ which is at that gate or a higher numbered gate and $P_i$ is the last one to arrive at that gate. Under this condition, $P_i$ waits by continually rechecking the condition. It will exit from the *while* loop only if $gate[j]$ becomes lower than $gate[i]$ when $P_j$ exits the critical section or the variable $last[k]$ is changed.

Since this algorithm is a minor generalization of Peterson's algorithm, we leave its proof of correctness as an exercise.

## 2.4 Lamport's Bakery Algorithm

A crucial disadvantage of Peterson's algorithm is that it uses shared variables that may be written by multiple writers. Specifically, the correctness of Peterson's algorithm depends on the fact that concurrent writes to the *last* variables result in a valid value.

We now describe Lamport's bakery algorithm, which overcomes this disadvantage. The algorithm is similar to that used by bakeries in serving customers. Each customer who arrives at the bakery receives a number. The server serves the customer with the smallest number. In a concurrent system, it is difficult to ensure that every process gets a unique number. So in case of a tie, we use process ids to choose the smaller process.

```
import java.util.Arrays;

class PetersonN implements Lock {
    int N;
    int[] gate;
    int[] last;
    public PetersonN(int numProc) {
        N = numProc;
        gate = new int[N]; //We only use gate[1]..gate[N-1]; gate[0] is unused
        Arrays.fill(gate, 0);
        last = new int[N];
        Arrays.fill(last, 0);
    }
    public void requestCS(int i) {
        for (int k = 1; k < N; k++) {
            gate[i] = k;
            last[k] = i;
            for (int j = 0; j < N; j++) {
                while ((j != i) &&  // there is some other process
                       (gate[j] >= k) &&  // that is ahead or at the same level
                       (last[k] == i)) // and I am the last to update last[k]
                {};// busy wait
            }
        }
    }
    public void releaseCS(int i) {
        gate[i] = 0;
    }
}
```

Figure 2.7: PetersonN.java

The algorithm shown in Figure 2.8 requires a process $P_i$ to go through two main steps before it can enter the critical section. In the first step (lines 15–21), it is required to choose a number. To do that, it reads the numbers of all other processes and chooses its number as one bigger than the maximum number it read. We will call this step the *doorway*. In the second step the process $P_i$ checks if it can enter the critical section as follows. For every other process $P_j$, process $P_i$ first checks whether $P_j$ is currently in the doorway at line 25. If $P_j$ is in the doorway, then $P_i$ waits for $P_j$ to get out of the doorway. At lines 26–29, $P_i$ waits for the $number[j]$ to be 0 or $(number[i], i) < (number[j], j)$. When $P_i$ is successful in verifying this condition for all other processes, it can enter the critical section.

```
 1  class Bakery implements Lock {
 2      int N;
 3      boolean[] choosing; // inside doorway
 4      int[] number;
 5      public Bakery(int numProc) {
 6          N = numProc;
 7          choosing = new boolean[N];
 8          number = new int[N];
 9          for (int j = 0; j < N; j++) {
10              choosing[j] = false;
11              number[j] = 0;
12          }
13      }
14      public void requestCS(int i) {
15          // step 1: doorway: choose a number
16          choosing[i] = true;
17          for (int j = 0; j < N; j++)
18              if (number[j] > number[i])
19                  number[i] = number[j];
20          number[i]++;
21          choosing[i] = false;
22
23          // step 2: check if my number is the smallest
24          for (int j = 0; j < N; j++) {
25              while (choosing[j]) ; // process j in doorway
26              while ((number[j] != 0) &&
27                        ((number[j] < number[i]) ||
28                        ((number[j] == number[i]) && j < i)))
29                  ; // busy wait
30          }
31      }
32      public void releaseCS(int i) { // exit protocol
33          number[i] = 0;
34      }
35  }
```

Figure 2.8: Lamport's bakery algorithm

We first prove the assertion:

(A1) If a process $P_i$ is in critical section and some other process $P_k$ has already chosen its number, then $(number[i], i) < (number[k], k)$.

Let $t$ be the time when $P_i$ read the value of $choosing[k]$ to be $false$. If $P_k$ had chosen its number before $t$, then $P_i$ must read $P_k$'s number correctly. Since $P_i$ managed to get out of the $k$th iteration of the *for* loop, $((number[i], i) < (number[k], k))$ at that iteration. If $P_k$ had chosen its number after $t$, then $P_k$ must have read the latest value of $number[i]$ and is guaranteed to have $number[k] > number[i]$. If $((number[i], i) < (number[k], k))$ at the $k$th iteration, this will continue to hold because $number[i]$ does not change and $number[k]$ can only increase.

We now claim the assertion:

(A2) If a process $P_i$ is in critical section, then $(number[i] > 0)$.

(A2) is true because it is clear from the program text that the value of any number is at least 0 and a process executes increment operation on its number at line 20 before entering the critical section.

Showing that the bakery algorithm satisfies mutual exclusion is now trivial. If two processes $P_i$ and $P_k$ are in critical section, then from (A2) we know that both of their numbers are nonzero. From (A1) it follows that $(number[i], i) < (number[k], k)$ and vice versa, which is a contradiction.

The bakery algorithm also satisfies starvation freedom because any process that is waiting to enter the critical section will eventually have the smallest nonzero number. This process will then succeed in entering the critical section.

It can be shown that the bakery algorithm does not make any assumptions on *atomicity* of any read or write operation. Note that the bakery algorithm does not use any variable that can be written by more than one process. Process $P_i$ writes only on variables $number[i]$ and $choose[i]$.

There are two main disadvantages of the bakery algorithm: (1) it requires $O(N)$ work by each process to obtain the lock even if there is no contention, and (2) it requires each process to use timestamps that are unbounded in size.

## 2.5 Lower Bound on the Number of Shared Memory Locations

In this section, we show that any algorithm that solves the mutual exclusion problem for $n$ processes must use at least $n$ memory locations. The key idea in showing the lower bound is that the system never gets into an *inconsistent* state in which a thread is not able to determine by reading shared locations whether the critical section is empty or not.

Consider a system with two processes, $P$ and $Q$. Suppose that there is a protocol that uses a single shared location $A$ to coordinate access to the critical section. It is clear that a thread must write to $A$ before entering the critical section otherwise the other thread would not be able to distinguish this state from the state in which the critical section is empty. Now, we consider an adversarial schedule as follows. Suppose that $P$ is about to write to $A$ before entering the CS. We now let process $Q$ execute its protocol, possibly writes on the location $A$, and enter the CS. We now resume process $P$ which writes on location $A$ overwriting anything that $Q$ may have written. At this point, the system is in an inconsistent state because even though $Q$ is in the CS, it is indistinguishable from the state in which the CS is available.

It is instructive to extend this argument for three processes $P$, $Q$ and $R$ using two shared locations $A$ and $B$. By previous argument any correct protocol for two processes must use both the locations. By letting $P$ run three times, we know that there exists an execution in which $P$ writes on some location, say $A$, twice. We first run $P$ till it is ready to write to $A$ for the first time. Then, we run $Q$ till it is ready to write in a separate location, say $B$, for the first time before entering the CS. If $Q$ does not write at any location other than $A$ and enters CS, $P$ can overwrite what $Q$ wrote and also enter the CS. At this point $Q$ is about to write on $B$ and $P$ is about to write on $A$. We now run $P$ again. Since $P$ overwrites on $A$ and nothing has been written on $B$; it cannot tell whether $Q$ has taken any step so far. We let $P$ enter the CS and then request CS again till it is about to write on $A$ again. At this point both $A$ and $B$ are in state consistent with no process in the CS. Next, we let $R$ run and enter the CS. Then, we run $P$ and $Q$ for one step thereby overwriting any change that $R$ may have done. One of them must be able to enter the CS to keep the algorithm deadlock-free. We have a violation of mutual exclusion in that state because $R$ is already in the CS.

## 2.6 Fischer's Algorithm

Our lower bound result assumed that processes are asynchronous. We now give an algorithm that uses timing assumptions to provide mutual exclusion with a single shared variable *turn*. The variable *turn* is either $-1$ signifying that the critical section is available or has the identifier of the process that has the right to enter the critical section. Whenever any process $P_i$ finds that *turn* is $-1$, it must set *turn* to $i$ in at most $c$ time units. It must then wait for at least $d$ units of time before checking the variable *turn* again. The algorithm requires $d$ to be greater than $c$. If *turn* is still set to $i$, then it can enter the critical section.

```
public class Fischer implements Lock {
        int N;
        int turn;
        int delta;

        public Fischer (int numProc) {
                N = numProc;
                turn = -1;
                delta = 5;
        }
        public void requestCS(int i) {
                while (true) {
                        while (turn != -1) {}; // wait for the door to open
                        turn = i; // write my id on turn
                        try { // Assume that delta is bigger than time to update turn
                                Thread.sleep(delta);
                        }
                        catch (InterruptedException e){};
                        if (turn == i) return;
                }
        }
        public void releaseCS(int i) {
                turn = -1;
        }
}
```

Figure 2.9: Fischer's mutual exclusion algorithm

We first show mutual exclusion. Suppose that both $P_i$ and $P_j$ are in the CS. Suppose that *turn* is $i$. This means that *turn* must have been $j$ when $P_j$ entered and then later *turn* was set to $i$. But $P_i$ can set *turn* to $i$ only within $c$ time units of $P_j$ setting *turn* to $j$. However, $P_j$ found *turn* to be $j$ even after $d \geq c$ time units. Hence, both $P_i$ and $P_j$ cannot be in CS.

We leave the proof of deadlock-freedom as an exercise.

## 2.7 A Fast Mutex Algorithm

We now present an algorithm due to Lamport that allows fast accesses to critical section in absence of contention. The algorithm uses an idea called *splitter* that is of independent interest.

### 2.7.1 Splitter

A *splitter* is a method that splits processes into three disjoint groups: *Left*, *Right*, and *Down*. We can visualize a splitter as a box such that processes enter from the top and either move to the left, the right or go down which explains the names of the groups. The key property a splitter satisfies is that at most one process goes in the down direction and not all processes go in the left or the right direction.

The algorithm for the splitter is shown in Fig. 2.10.

```
P_i::
 var
     door: {open, closed} initially open
     last : pid initially -1;

 last := i;
 if (door == closed)
     return Left;
 else
     door := closed;
     if (last == i) return Down;
     else return Right;
 end
```

Figure 2.10: Splitter Algorithm

A splitter consists of two variables: *door* and *last*. The door is initially open and if any process finishes executing splitter the door gets closed. The variable *last* records the last process that executed the statement *last := i*.

Each process $P_i$ first records its pid in the variable *last*. It then checks if the door is closed. All processes that find the door closed are put in the group *Left*. We claim

**Lemma 2.1** *There are at most $n - 1$ processes that return* Left.

**Proof:** Initially, the door is open. At least one process must find the door to be open because every process checks the door to be open before closing it. Since at least one process finds the door open, it follows that $|Left| \leq n - 1$.

■

Process $P_i$ that find the door open checks if the last variable contains its pid. If this is the case, then the process goes in the *Down* direction. Otherwise, it goes in the *Right* direction.

We now have the following claim.

**Lemma 2.2** *At most one process can return* Down.

**Proof:** Suppose that $P_i$ be the first process that finds the door to be open and *last* equal to $i$ (and then later returns *Down*). We have the following order of events: $P_i$ wrote *last* variable, $P_i$ closed the door, $P_i$ read *last* variable as $i$. During this interval, no process $P_j$ modified the last variable. Any process that modifies *last* after this interval will find the door closed and therefore cannot return *Down*. Consider any process $P_j$ that modifies *last* before this interval. If $P_j$ checks *last* before the interval, then $P_i$ is not the first process then finds last as itself. If $P_j$ checks *last* after $P_i$ has written the variable last, then it cannot find itself as the last process since its pid was overwritten by $P_i$.

■

**Lemma 2.3** *There are at most $n-1$ processes that return* Right.

**Proof:** Consider the last process that wrote its index in *last*. If it finds the door closed, then that process goes left. If it finds the door open then it goes down.

■

Note that the above code does not use any synchronization. In addition, the code does not have any loop.

## 2.7.2 Lamport's Fast Mutex Algorithm

Lamport's fast mutex algorithm shown in Fig. 2.11 uses two shared registers $X$ and $Y$ that every process can read and write. A process $P_i$ can acquire the critical section either using a *fast* path when it finds $X = i$ or using a *slow* path when it finds $Y = i$. It also uses $n$ shared single-writer-multiple-reader registers $flag[i]$. The variable $flag[i]$ is set to value *up* if $P_i$ is actively contending for mutual exclusion using the fast path. The shared variable $X$ plays the role of the variable *last* in the splitter algorithm. The variable $Y$ plays the role of door in the splitter code. When $Y$ is $-1$, the door is open. A process $P_i$ closes the door by updating $Y$ with $i$.

Processes that are in the group *Left* of the splitter, simply retry. Before retrying, they lower their flag and wait for the door to be open (i.e. $Y$ to be $-1$). A process that is in the group *Down* of the splitter succeeds in entering the critical section. Note that at most process may succeed using this route. Processes that are in the group *Right* of the splitter first wait for all flags to go down. This can happen only if no process returned *Down*, or if the process that returned *Down* releases the critical section. Now consider the last process in the group *Right* to update $Y$. That process will find its pid in $Y$ and can enter the critical section. All other processes wait for the door to be open again and then retry.

**Theorem 2.4** *Fast Mutex algorithm in Fig. 2.11 satisfies Mutex.*

**Proof:** Suppose $P_i$ is in the critical section. This means that $Y$ is not $-1$ and $P_i$ exited either with $X = i$ or $Y = i$.

Any process that finds $Y \neq -1$ gets stuck till $Y$ becomes $-1$ so we can focus on processes that found $Y$ equal to $-1$.

Case 1: $P_i$ entered with $X = i$

Its flag stays up and thus other processes stay blocked.

Case 2: $P_i$ entered with $Y = i$

Consider any $P_j$ which read $Y == -1$. $X$ is not equal to $j$ otherwise $P_j$ would have entered CS and $P_i$ would have been blocked

Since $Y = i$, $P_j$ would get blocked waiting for $Y$ to become $-1$.

■

```
 var
     X, Y: int initially -1;
     flag: array[1..n] of {down, up};

acquire(int i)
{
 while (true)
     flag[i] := up;
     X := i;
     if (Y != -1) { // splitter's left
         flag[i] := down;
         waitUntil(Y == -1)
         continue;
     }
     else {
         Y := i;
         if (X == i) // success with splitter
             return; // fast path
         else {// splitter's right
             flag[i] := down;
             forall j:
                 waitUntil(flag[j] == down);
             if (Y == i) return; // slow path
             else {
                 waitUntil(Y == -1);
                 continue;
             }
         }
 }
}

 release(int i)
 {
     Y := -1;
     flag[i] := down;
 }
```

Figure 2.11: Lamport's Fast Mutex Algorithm

**Theorem 2.5** *Fast Mutex algorithm in Fig. 2.11 satisfies deadlock-freedom.*

**Proof:**
Consider processes that found the door open, i.e., $Y$ to be $-1$. Let $Q$ be the set of processes that are stuck that found the door open. If any one of them succeeded in "last-to-write-X" we are done; otherwise, the last process that wrote $Y$ can enter the CS.

■

## 2.8 Locks with Get-and-Set Operation

Although locks for mutual exclusion can be built using simple read and write instructions, any such algorithm requires as many memory locations as the number of threads. By using instructions with higher atomicity, it is much easier to build locks. For example, the `getAndSet` operation (also called `testAndSet`) allows us to build a lock as shown in Fig. 2.12.

```java
import java.util.concurrent.atomic.*;

public class GetAndSet implements MyLock {
    AtomicBoolean isOccupied = new AtomicBoolean(false);
    public void lock() {
        while (isOccupied.getAndSet(true)) {
            Thread.yield();
           // skip();
        }
    }
    public void unlock() {
        isOccupied.set(false);
    }
}
```

Figure 2.12: Building Locks Using GetAndSet

This algorithm satisfies the mutual exclusion and progress property. However, it does not satisfy starvation freedom. Developing such a protocol is left as an exercise.

Most modern machines provide the instruction `compareAndSet` which takes as argument an expected value and a new value. It atomically sets the current value to the new value if the current value equals the expected value. It also returns true if it succeeded in setting the current value; otherwise, it returns false. The reader is invited to design a mutual exclusion protocol using `compareAndSet`.

We now consider an alternative implementation of locks using getAndSet operation. In this implementation, a thread first checks if the lock is available using the `get` operation. It calls the `getAndSet` operation only when it finds the critical section available. If it succeeds in `getAndSet`, then it enters the critical section; otherwise, it goes back to *spinning* on the get operation. The implementation called GetAndGetAndSet (or testAndTestAndSet) is shown in Fig. 2.13.

Although the implementations in Fig. 2.12 and Fig. 2.13 are functionally equivalent, the second implementation usually results in faster accesses to the critical section on current multiprocessors. Can you guess why?

The answer to the above question is based on the current architectures that use a shared bus and a local cache with each core. Since an access to the shared memory via bus is much slower compared to an access to the local cache, each core checks for a data item in its cache before issuing a memory request. Any

```
import java.util.concurrent.atomic.*;

public class GetAndGetAndSet implements MyLock {
    AtomicBoolean isOccupied = new AtomicBoolean(false);
    public void lock() {
        while (true) {
            while (isOccupied.get()) {
            }
            if (!isOccupied.getAndSet(true)) return;
        }
    }
    public void unlock() {
        isOccupied.set(false);
    }
}
```

Figure 2.13: Building Locks Using GetAndGetAndSet

data item that is found in the local cache is termed as a *cache hit* and can be served locally. Otherwise, we get a *cache miss* and the item must be served from the main memory or cache of some other core. Caches improve the performance of the program but require that the system ensures coherence and consistency of cache. In particular, an instruction such as getAndSet requires that all other cores should invalidate their local copies of the data item on which getAndSet is called. When cores spin using getAndSet instruction, they repeatedly access the bus resulting in high contention and a slow down of the system. In the second implementation, threads spin on the variable `isOccupied` using get. If the memory location corresponding to `isOccupied` is in cache, the thread only reads cached value and therefore avoids the use of the shared data bus.

Even though the idea of getAndGetAndSet reduces contention of the bus, it still suffers from high contention whenever a thread exits the critical section. Suppose that a large number of threads were spinning on their *cached* copies of `isOccupied`. Now suppose that the thread that had the lock leaves the critical section. When it updates `isOccupied`, the cached copies of all spinning threads get invalidated. All these threads now get that `isOccupied` is false and try to set it to true using `getAndSet`. Only, one of them succeeds but all of them end up contributing to the contention on the bus. An idea that is useful in reducing the contention is called *backoff.* Whenever a thread finds that it failed in getAndSet after a successful get, instead of continuing to get the lock, it backs off for a certain random period of time. The *exponential* backoff doubles the maximum period of time a thread may have to wait after any unsuccessful trial. The resulting implementation is shown in Fig. 2.14.

Another implementation that is sometimes used for building locks is based on getting a ticket number similar to Bakery algorithm. This implementation is also not scalable since it results in high contention for currentTicket.

## 2.9 Queue Locks

Our previous approaches to solve mutual exclusion require threads to spin on the shared memory location `isOccupied`. As we have seen earlier, any update to this variable results in multiple threads getting cache invalidation. Even when threads backoff, we have the issue of deciding the duration of time to back off. If we do not back off for a sufficient period, the bus contention is still there. On the other hand, if the period for the backoff is large, then the threads may be sleeping even when the critical section is not occupied. In this section, we present alternate methods to avoid spinning on the same memory location. All three

```java
import java.util.concurrent.atomic.*;

public class MutexWithBackOff{
    AtomicBoolean isOccupied = new AtomicBoolean(false);
    public void lock() {
        while (true) {
           while (isOccupied.get()) {
           }
           if (!isOccupied.getAndSet(true)) return;
           else {
                int timeToSleep = calculateDuration();
                Thread.sleep(timeToSleep);
           }
        }
    }
    public void unlock() {
        isOccupied.set(false);
    }
}
```

Figure 2.14: Using Backoff during Lock Acquisition

```java
import java.util.concurrent.atomic.*;

public class TicketMutex {
    AtomicInteger nextTicket = new AtomicInteger(0);
    AtomicInteger currentTicket = new AtomicInteger(0);
    public void lock() {
        int myticket = nextTicket.getAndIncrement();
        while (myticket != currentTicket.get()) {
           // skip();
        }
    }
    public void unlock() {
        int temp = currentTicket.getAndIncrement();
    }
}
```

Figure 2.15: Using Tickets for Mutex

methods maintain a queue of threads waiting to enter the critical section. Anderson's lock uses a fixed size array, CLH lock uses an implicit linked list and MCS lock uses an explicit linked list for the queue. One of the key challenges in designing these algorithms is that we cannot use locks to update the queue.

### 2.9.1 Anderson's Lock

Anderson's lock uses a circular array `Available` of size $n$ which is at least as big as the number of threads that may be contending for the critical section. The array is circular so that the index $i$ in the array is always a value in the range $0..n-1$ and is incremented modulo $n$. Different threads waiting for the critical section spin on the different slots in this array thus avoiding the problem of multiple threads spinning on the same variable. An atomic integer `tailSlot` (initialized to 0) is maintained which points the next available slot in the array. Any thread that wants to lock, reads the value of the `tailSlot` in its local variable `mySlot` and advances it in one atomic operation using `getAndIncrement()`. It then spins on `Available[mySlot]` until the slot becomes available. Whenever a thread finds that the entry for its slot is true, it can enter the critical section. The algorithm maintains the invariant that distinct processes have distinct slots and also that there is at most one entry in `Available` that is true. To unlock, the thread sets its own slot as false and the entry in the next slot to be true. Whenever a thread sets the next slot to be true, any thread that was spinning on that slot can then enter the critical section. Note that in Anderson lock, only one thread is affected when a thread leaves the critical section. Other threads continue to spin on other slots which are cached and thus result only in accesses of local caches.

Note that the above description assumes that each slot is big enough so that adjacent slots do not share a cache line. Hence even though we just need a single bit to store `Available[i]`, it is important to keep it big enough by padding to avoid the problem of *false sharing*. Also note that since Anderson's lock assigns slots to threads in the FCFS manner, it guarantees fairness and therefore freedom from starvation.

A problem with Anderson lock is that it requires a separate array of size $n$ for each lock. Hence, a system that uses $m$ locks shared among $n$ threads will use up $O(nm)$ space.

```
// pseudo-code for Anderson Lock
public class AndersonLock {
    AtomicInteger tailSlot = new AtomicInteger(0);
    boolean [] Available;
    ThreadLocal<Integer> mySlot ;//initialize to 0

    public AndersonLock(int n) { // constructor
     // all Available false except Available[0]
    }
    public void lock() {
      mySlot.set(tailSlot.getAndIncrement() % n);
      spinUntil(Available[mySlot]);
    }
    public void unlock() {
        Available[mySlot.get()] = false;
        Available[(mySlot.get()+1) % n] = true;
    }
}
```

Figure 2.16: Anderson Lock for Mutex

### 2.9.2 CLH Queue Lock

We can reduce the memory consumption in Anderson's algorithm by using a dynamic linked list instead of a fixed size array. The idea of CLH Lock (shown in Fig. 2.17) is due to Travis Craig, Erik Hagersten and Anders Landin. For each lock, we maintain a linked list. Any thread that wants to get that lock inserts a node in that linked list. Just as we had a shared variable `tailSlot` in Anderson's algorithm, we maintain a shared variable `tailNode` that points to the last node inserted in the linked list. Suppose a thread wants to insert `myNode` in the linked list. This insertion in the linked list must be atomic; therefore, the thread uses `pred = tailNode.getAndSet(myNode)` to insert its node in the linked list as well as get the pointer to the previous node in the variable `pred`. Each node has a field `locked` which is initially set to true when the node is inserted in the linked list. A thread spins on the field `locked` of the predecessor node `pred`. Whenever a thread releases the critical section, it sets the `locked` field to false. Any thread that was spinning on that field can now enter the critical section.

It is important to note that if the thread that exits the critical section wants to enter the critical section again, it cannot reuse its own node because the next thread may still be spinning on that node's field. However, it can reuse the node of its predecessor.

Also note that we do not maintain an explicit pointer in each node. The linked list is virtual. A thread that is spinning has the variable `pred` that points to the predecessor node in the linked list. The node itself has a single field: `locked`.

```java
import java.util.concurrent.atomic.*;
public class CLHLock implements MyLock {
     class Node {
         boolean locked;
     }
     AtomicReference<Node> tailNode;
     ThreadLocal<Node> myNode;
     ThreadLocal<Node> pred;

     public CLHLock() {
        tailNode = new AtomicReference<Node>(new Node());
        tailNode.get().locked = false;
        myNode = new ThreadLocal<Node> () {
           protected Node initialValue() {
                 return new Node();
           }
        };
        pred = new ThreadLocal<Node> ();

     }
     public void lock() {
       myNode.get().locked = true;
       pred.set(tailNode.getAndSet(myNode.get()));
       while(pred.get().locked) { Thread.yield(); };
     }
     public void unlock() {
         myNode.get().locked = false;
         myNode.set(pred.get()); // reusing predecessor node for future use
     }
}
```

Figure 2.17: CLH Lock for Mutex

### 2.9.3 MCS Queue Lock

In many architectures, each core may have local memory such that access to its local memory is fast but access to the local memory of another core is slower. In such architectures, CLH algorithm results in threads spinning on remote locations. We now show a method due to John Mellor-Crummey and Michael Scott called MCS lock that avoids remote spinning, i.e. spinning on memory of a different core.

In MCS lock (shown in Fig. 2.18), we maintain a queue based on an explicit linked list. Any thread that wants to access the critical section inserts its node in the linked list at the tail. Each node has a field `locked` which is initialized to true when the node is inserted in the linked list. A thread spins on that field waiting for it to become false. It is the responsibility of the thread that exits the critical section to set this field to false.

When a thread $p$ exits the critical section, it first checks if there is any node linked next to its node. If there is such a node, then it makes the `locked` field false and removes its node from the linked list by making its next pointer null. The tricky case is when it finds that there is no node linked next to its node. There can be two reasons for this. First, there is no thread that has inserted its node in the linked list using `tailNode` yet. In this case, `tailNode` is still pointing to its node. In this case, the thread must simply set the tailNode to null. The second case is that a thread $q$ has called tailNode.getAndSet but not yet set the next pointer of $p$'s node to $q$'s node. In this case, the thread $p$ waits until its next field is not null. It can then set the locked field for that node to be false as before.

## 2.10 Problems

2.1. Show that any of the following modifications to Peterson's algorithm makes it incorrect:

   (a) A process in Peterson's algorithm sets the *turn* variable to itself instead of setting it to the other process.

   (b) A process sets the *turn* variable before setting the *wantCS* variable.

2.2. Show that Peterson's algorithm also guarantees freedom from starvation.

2.3. Show that the bakery algorithm does not work in absence of *choosing* variables.

2.4. Prove the correctness of the Filter algorithm in Figure 2.7. (Hint: Show that at each level, the algorithm guarantees that at least one processes loses in the competition.)

2.5. Consider the software protocol shown in Figure 2.19 for mutual exclusion between two processes. Does this protocol satisfy (a) mutual exclusion, and (b) livelock freedom (both processes trying to enter the critical section and none of them succeeding)? Does it satisfy starvation freedom?

2.6. Modify the bakery algorithm to solve $k$-mutual exclusion problem, in which at most $k$ processes can be in the critical section concurrently.

2.7. Give a mutual exclusion algorithm that uses atomic swap instruction.

2.8. Give a mutual exclusion algorithm that uses TestAndSet instruction and is free from starvation.

*2.9. Give a mutual exclusion algorithm on $N$ processes that requires $O(1)$ time in absence of contention.

```java
import java.util.concurrent.atomic.*;

public class MCSLock implements MyLock {
    class QNode {
        boolean locked;
        QNode next;
        QNode() {
          locked = true;
          next = null;
        }
    }
    AtomicReference<QNode> tailNode = new AtomicReference<QNode>(null);
    ThreadLocal<QNode> myNode;

    public MCSLock() {
       myNode = new ThreadLocal<QNode> () {
          protected QNode initialValue() {
               return new QNode();
          }
       };
    }
    public void lock() {
      QNode pred = tailNode.getAndSet(myNode.get());
      if (pred != null){
          myNode.get().locked = true;
          pred.next = myNode.get();
          while (myNode.get().locked){ Thread.yield(); };
      }
    }
    public void unlock() {
        if (myNode.get().next == null) {
            if (tailNode.compareAndSet(myNode.get(), null)) return;
            while (myNode.get().next == null) { Thread.yield(); };
        }
        myNode.get().next.locked = false;
        myNode.get().next = null;
    }
}
```

Figure 2.18: MCS Lock for Mutex

```
class Dekker implements Lock {
    boolean wantCS[] = {false, false};
    int turn = 1;
    public void requestCS(int i) { // entry protocol
        int j = 1 - i;
        wantCS[i] = true;
        while (wantCS[j]) {
            if (turn == j) {
                wantCS[i] = false;
                while (turn == j) ;// busy wait
                wantCS[i] = true;
            }
        }
    }
    public void releaseCS(int i) { // exit protocol
        turn = 1 - i;
        wantCS[i] = false;
    }
}
```

Figure 2.19: Dekker.java

## 2.11 Bibliographic Remarks

The mutual exclusion problem was first introduced by Dijkstra [Dij65a]. Dekker developed the algorithm for mutual exclusion for two processes. Dijkstra [Dij65b] gave the first solution to the problem for $N$ processes. The bakery algorithm is due to Lamport [Lam74], and Peterson's algorithm is taken from a paper by Peterson [Pet81].