# 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 transitiors on a chip increase, multiprocessor chips will become fairly common. With enough parallelism available in applications, such systems will easily beat sequential systems in performance. 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 decreases prices of computer processors and the availability of high-bandwidth links to connect them. Programming these 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 not? At the hardware level, we would expect that the prevalent model would 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.
- *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 completely different type than 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.

- *Geographical structure*: The geographical 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 purely a distributed one? The reasons for keeping a parallel system at each node are mainly of a technological nature. With the current technology it is generally faster to update a shared memory location than to send a message to some other 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 heavy-weight processes that communicate using messages (or remote method invocations). Each heavy-weight process consists of multiple light-weight 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 heavy-weight process, we get the usual model of a parallel system. Why would the system 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 only be accessed through an explicit message (or a remote procedure call). The object orientation promotes reusability as well as design simplicity. Conversely, threads are also useful to provide efficient objects. For many applications such as servers, it is useful to have a large shared data structure. It is a programming burden to split the data structure across multiple heavy-weight processes.

## 1.3 Characteristics of Parallel and Distributed Systems

Recall that we distinguish between parallel and distributed systems based on 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 upon whether multiple processor work in a lock step manner or not. 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 due to 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 it based on a tighly-coupled architecture is rarely a good idea due to loss of performance due to synchronization. In this boo, we will assume that all or 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. 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 etc. In this book we will see how failure detectors can be built to alleviate some of these problems.

## 1.4 Design Goals

The experience in large parallel and distributed software systems has shown that their design should take the following concepts into cosideration [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 upon 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 should not have to deal with unnecessary details. For example, in a heterogeneous distributed system the differences in the internal representation of data (for example, 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 adheres to a fixed set rules for syntax and semantics, preferebly 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 how big is the cache, which pages are to be cached, and how long should those pages remain in the cache. Such questions may be answered better by the user and therefore it is better for the user to build his own caching policy on the 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 service form the server goes up. Generally, the system is either completely decentralized using distributed algorithms or partially decentralized using a hierarchy of servers.

## 1.5 Problems

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

## 1.6 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], 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 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. There are also some edited books available. 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 [Mul94] covers many other topics such as protection, fault-tolerance and real-time communications.

# Chapter 2

# Parallel Programming: Process and Tasks Specification

## 2.1 Introduction

In this chapter we cover the programming constructs for shared memory-based languages. It should be noted that the issues of concurrency arises 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 chapter as *concurrent* programming.

Before we embark upon 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 it is concurrent we get multiple processes. A process can be viewed as consisting of three portions 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, i.e. code and data, then they are called *lightweight processes* or *threads.* When process has its own code and data, it is called a *heavywight process*, or simply a process. Heavywight processes may share data through files.

Any programming language that supports concurrent programming must have a way to specify the process structure, how various processes communicate and synchronize with each other. We will discuss these next.

## 2.2 Process and Thread Declaration

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 in 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.

```
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 the 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 woken up with the pid of the child process as the return value from the wait call. This way, the parent process can determine which of its child terminated.

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

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

```
cobegin
       S1
      ||
       S2
coend
```

The above construct says that $S1$ and $S2$ must be executed in parallel. Further, if one of them finishes earlier then the other then it should wait for the other one to finish. Combining the cobegin-coend with the sequencing operator semicolon (;), we can create any series-parallel task structure. For example, $S0; cobegin S1 || S2 coend; S3$ starts off with one process that excutes $S0$. When $S0$ is finished we have two processes (or threads) that execute $S1$ and $S2$. When both the statements are done, only then $S3$ is started. It should be noted that the activity precedence graph for a program constructed with *cobegin-coend* and sequencing can only be a series-parallel graph.

Yet another method 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 "Hello World" can be launched as follows:

```
class HelloWorldThread extends Thread {
   public void run() {
System.out.println("Hello World");
   }
}

class HelloDriver {
   public static void main(String[] args) {
   HelloWorldThread t = new HelloWorldThread();
   t.start();
   }
}
```

In this simple 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 follows:

```
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<1000;i++)
  System.out.println(getName() + ": Hello World");

     }
}

class FooBarDriver {
   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();
   }
}
```

## 2.3 Thread Scheduling

In our previous example, we had two threads. The same Java program will work for a single CPU machine and also for a multi-processor machine. In a single CPU machine, if both threads are runnable then which one would be picked by the system to run? The answer to this question depends upon the *priority* and the *scheduling policy* of the system. Generally, it is not a good idea to make the correctness of the algorithm dependent on the scheduling policy of the system. One should assume only the weakest guarantee specified as follows. Let $R$ be the set of all runnable processes with $p$ as the highest priority of processes in $R$. Let $R'$ be the subset of processes of $R$ with priority $p$. Then, every runnable process in $R'$ will eventually get to run. A java thread that is running may block by calling *sleep*, *wait* or any systemfunction that is blocked. 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.

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 only use a value between these two constants. By defualt, a thread has the priority `NORM_PRIORITY`.

## 2.4 Problems

2.1. Consider the following sequential program:

```
S0: read(a,b);
S1: x := 2*a;
S2: y := b*b;
S3: w := x - y;
S4: z:=y*y;
S5: write(w,z);
```

The program has six activities S0-S5. Draw a diagram showing precedence constraints in their execution.

2.2. Write a java object 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 $-1$.

2.3. Consider the class shown below.

```
class Schedule {
  static int x = 0;
  static int y = 0;
  public static void op1(){x = 1;return y;}
  public static void 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`.

## 2.5 Bibliographic Remarks

# Chapter 3

# Mutual Exclusion: Software solutions

## 3.1 Mutual Exclusion

When processes share the data, it is important to synchronize their access to the data so that updates do not get lost due to concurrent accesses. This can be see from the following example. Assume that the initial value of a shared variable $x$ is 0 and that there are two processes such that each one of them increments $x$ by the following code `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 code `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 note that the execution of two processes may get interleaved as follows:

```
p1: LD R, x; load register R from x
p1: INC R  ; increment register R
p2: LD R, x; load register R from x
p2: INC R  ; increment register R
p1: ST R,x  ; store register R to x
p2: 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, `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 design an entry protocol and an exit protocol in the following code such that mutual exclusion is not violated.

```
 while (true) {
    <critical section entry protocol>
    criticalSection();
    <critical section exit protocol>
```

```
   nonCriticalSection();
}
```

In the above code, multiple processes repeatedly want to enter the critical section. After exiting from the critical section a process spends an undetermined amount of time in the Non-critical section of the code. It then goes into the while loop again.

Let us now look at some possible protocols, one may attempt, to solve this problem. For simplicity we first assume that there are only two processes $p1$ and $p2$.

### 3.1.1 Mutual Exclusion for Two Processes

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`.

```
boolean openDoor = true;

 while (!openDoor); // busy wait
 openDoor = false;
 criticalSection();
  openDoor = true;  // exit protocol
  nonCriticalSection();
```

The above attempt does not work because the testing of `openDoor` and setting it to `false` is not done atomically. It can happen that one process checks for the `openDoor` and gets out of the busy wait. However, before that process could set `openDoor` to `false` it is context switched. 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 above attempt, the shared variable did not record who set the `openDoor` to false. One may try to fix the above problem by keeping two shared variables `wantCS1` and `wantCS2` as follows.

```
boolean wantCS1 = false;
boolean wantCS2 = false;

// code for process 1
 wantCS1 = true;
 while (wantCS2) ; // busy wait
 criticalSection();
  wantCS1 = false;
  nonCriticalSection();
```

Unfortunately, this attempt also does not work. Both processes could set their `wantCS` to true and then indefinitely loop waiting for other process to sets its `wantCS` false.

Yet another attempt to fix the problem could be

```
int turn = 1;

// code for process 1
  while (turn==2);
 criticalSection();
  turn=2;
  nonCriticalSection();
```

This protocol does guarantee *mutual exclusion*. It also guarantees that it both processes are trying to 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 $p1$ exits from the critical section it cannot enter the critical section again till process $p2$ has entered the critical section. If process $p2$ is not interested in the critical section, then process $p1$ is simply stuck waiting for process $p2$. This is not desirable.

By combining, the previous two approaches we get Peterson's algorithm for mutual exclusion problem in a two process system. In this protocol, a process enters the critical section only if either it is its turn to enter the critical section or if the other process is not interested in the critical section.

```
  boolean wantCS1 = false;
  boolean wantCS2 = false;
  int turn = 1;

// code for process 1
    wantCS1 = true;
    turn = 2;
    while (wantCS2 && (turn==2));
   criticalSection();
    wantCS1 = false;

    nonCriticalSection();
```

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.

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 must succeed in entering the critical section.

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

We first show that mutual exclusion is satisfied by the above algorithm. Assume without loss of generality that $p1$ was the first one to enter the critical section. For $p1$ to enter the critical section it must have either read $wantCS2$ as false or $turn$ as 1. We do a case analysis.

*case 1*: $p1$ read $wantCS2$ as false

If $wantCS2$ is false, then for $p2$ to enter the critical section it would have to set $wantCS2$ to true. From this case, we get the following order of events: $p1$ reads $wantCS2$ as false $<$ $p2$ sets the value of $wantCS2$ as true.

This order of events implies that $p2$ would set $turn = 1$ before checking the entry condition and after the event of $p1$ reading $wantCS2$. On the other hand, $p1$ set $turn = 2$ before reading $wantCS2$. Therefore, we have the following order of events in time: $p1$ sets $turn$ to 2 $<$ $p1$ reads $wantCS2$ as false $<$ $p2$ sets $wantCS2$ as true $<$ $p2$ sets $turn$ to 1 $<$ $p2$ reads $turn$. Clearly, $turn$ can only be 1 when $p2$ reads it.

Now let us look at what values of $wantCS1$ $p2$ can possibly read. Because $p1$ sets $wantCS1$ before reading $wantCS2$, and $p2$ reads $wantCS1$ after writing the value of $wantCS2$, we get that $p2$ reads $wantCS1$ after $p1$ has set it to true. Therefore, $p2$ can only read $wantCS1$ as true.

Because $p2$ reads $turn$ as 1 and $wantCS1$ as true, it cannot enter the critical section.

*case 2*: $p1$ read $turn$ as 1.

This implies the following order: $p2$ sets $turn = 1$ between $p1$ setting $turn = 2$ and $p1$ reading the value of $turn$. Since $p2$ reads the value of turn only after setting $turn = 1$, we know that it can only read it as 1. Also $wantCS1$ is set before $p1$ set $turn = 2$. Therefore, $p1$ wrote $wantCS1$ before $p2$ set $turn = 1$. This implies that $p2$ can only read the value of $wantCS1$ as true. Therefore, $p2$ cannot enter the critical section.

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

$$wantCS2 \wedge (turn = 2) \wedge wantsCS1 \wedge (turn = 1)$$

which is clearly false because $(turn = 2) \wedge (turn = 1)$ 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.

### 3.1.2 Mutual Exclusion for $N$ Processes

Although Peterson's algorithm satisfies all the properties that we initially required from the protocol, it suffers from the following disadvantages.

1. It works only for two processes. Although the algorithm can be extended to $N$ processes by repeated invocation of the entry protocol, the resulting algorithm is more complex.

2. Peterson's algorithm uses variables that can be written by multiple writers. A solution in which there is at most a single writer to any shared variable is more desirable. The correctness of Peterson's algorithm depends on the fact that concurrent writes to the turn variable results in a valid value.

We now describe Lamport's bakery algorithm that overcomes these two disadvantages. The algorithm is similar to that used by bakeries in serving customers. Each customer that arrives at bakery receives a number. The server serves customer with the smallest number.

In a concurrent environment, 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.

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

```
class Bakery implements Lock {
    int N;
    boolean [] choosing;
    int [] number;
    public Bakery(int numProc){
N = numProc;
choosing = new boolean[N];
number = new int[N];
for (int j=0; j<N; j++){ choosing[j] = false; number[j] = 0; }
    }

    public void requestCS(int i){
       choosing[i] = true; // first choose a number
       for (int j = 0; j < N; j++)
            if (number[j] > number[i])
                number[i] = number[j];
        number[i]++;
        choosing[i] = false;
        for (int j = 0; j < N; j++) {
            while (choosing[j]) ;
            while ((number[j] !=0) &&
                   ((number[j] < number[i]) ||
                    ((number[j] == number[i]) && j < i))) ;
        }
    }

    public void releaseCS(int i) { // exit protocol
     number[i] = 0;
    }
}
```

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 $(number[i], i) < number[k], k)$.

If the process $p_i$ is in critical section, then it managed to get out of the $k^{th}$ iteration of the *for* loop. This implies that either $(number[k] = 0)$ or $(number[i], i) < number[k], k)$ at that iteration. First assume that process $p_i$ read $number[k]$ as 0. This means that process $p_k$ must not have finished choosing the number yet. There are two cases. Either $p_k$ has not entered the doorway or it has entered the doorway but not exited yet. If $p_k$ has not entered the doorway, it will read the latest value of $number[i]$ and is guaranteed to have $number[k] > number[i]$. If it had entered the doorway, then this entry must be after $p_i$ had checked $choosing[k]$ because $p_i$ waits for $p_k$ to finish choosing before checking the condition $(number[k] = 0) \vee ((number[i], i) < number[k], k))$. This again means that that $p_k$ will read the latest value of $number[i]$ and therefore $(number[i] < number[k])$.

If $(number[i], i) < number[k], k)$ at $k^{th}$ iteration, then this will continue to hold because $number[i]$ does not change and $number[k]$ can only increase.

We can also claim the assertion:
(A2) If a process $p_i$ is in critical section, then $(number[i] > 0)$.

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

Now showing that the bakery algorithm satisfies mutual exclusion is 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 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.

## 3.2 Problems

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

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

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

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

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

3.4. Consider the following software protocol for mutual exclusion between two processes.

```
   boolean wantCS1 = false;
   boolean wantCS2 = false;
   int turn = 1;

  // code for process 1
     wantCS1 = true;
     while (wantCS2) {
         if (turn==2) {
wantCS1 = false;
                while (turn==2);// busy wait
wantCS1 = true;
 }
     }
    criticalSection();

     turn = 2;
     wantCS1 = false;

     nonCriticalSection();
```

Does this protocol satisfy (1) mutual exclusion, and (2) livelock-freedom (both processes trying to enter the critical section and none of them succeeding)? Does it satisfy starvation-freedom?

# Chapter 4

# Mutual Exclusion: Hardware solutions and Semaphores

## 4.1 Synchronization Hardware

As we have seen, pure software solutions to mutual exclusion can be quite complex and expensive. However, mutual exclusion can be provided quite easily with help of hardware. We discuss some techniques below.

### 4.1.1 Disabling interrupts

In a single CPU system, a process may disable all the interrupts before entering the critical section. This means that the process cannot be context switched (because context switching occurs when the currently running thread receives a clock interrupt when its current time slice is over). On exiting the critical section, the process enables interrupts. Although this method can work for a single CPU machine, it has many undesirable features. First, it is infeasible for multiple CPU systems in which even if interrupt is disabled in one CPU, another CPU may execute. Disabling interrupts of all CPU's is very expensive. Also many system facilities such as clock registers are maintained using hardware interrupts. If interrupts are disabled then these registers may not show correct values.

### 4.1.2 Instructions with Higher Atomicity

Most machines provide instructions with higher level of atomicity than read or write. The testAndSet instruction provided by some machines does both read and write in one atomic instruction. This instruction reads and returns the old value of a memory location while replacing it with a new value.

```
 public synchronized boolean testAndSet(boolean memVariable, boolean newValue)
     boolean oldValue = memVariable; //reads a memory location
     memVariable = newValue; //updates the memory location
     return oldValue;
}
```

If testAndSet instruction is available, then one can develop a very simple protocol for mutual exclusion.

```
boolean lock = false;
while (testAndSet(lock,true));
```

```
      criticalSection();
  lock = false;
```

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

Sometimes machine provide the instruction `swap`, that can swap two memory locations in one atomic step. Its semantics is given below:

```
  public synchronized void swap(boolean mem1, boolean mem2)
     boolean temp = mem1;
     mem1 = mem2;
     mem2 = temp;
 }
```

The reader is invited to design a mutual exclusion protocol using `swap`.

## 4.2 Semaphores

All of our previous solutions to the mutual exclusion problem were wasteful in one regard. If a process is unable to enter the critical section, it repeatedly checks for the entry condition to be true. While a process is doing this no useful work can be accomplished. Instead of checking the entry condition repeatedly, if the process checked the condition only when it could have become true, it would not waste CPU cycles. Accomplishing this requires support from the operating system.

Dijkstra proposed the object *semaphore* that solves the problem of busy wait. A semaphore has two fields associated with it—its value and a queue of blocked processes and two operations associated with it — $P()$ and $V()$. A semaphore can have only two values $false$ or *true*. It can be initialized to either value. The queue of blocked processes is initially empty and a process may add itself to the queue when it makes a call to $P()$. When a process calls $P()$ and the value is *true*, then the value of the semaphore becomes $false$. However, if the value of the semaphore is $false$, then the process gets blocked until it becomes *true*. The `wait()` call in the code for BinarySemaphore achieves this. When the value becomes *true*, the process can make it $false$ and return from $P()$. The call to $V()$ makes the value *true* and also notifies a process if the queue of processes sleeping on that semaphore is nonempty.

```
public class BinarySemaphore {
  boolean value;

  BinarySemaphore(boolean initValue){ value = initValue;}

  public synchronized void P() {
     if (value==false)
        try {
          wait();
        } catch (InterruptedException e){ }
     value = false;
  }

  public synchronized void V(){
    value = true;
    notify();
  }
}
```

Now, mutual exclusion is almost trivial to implement.

```
 BinarySemaphore mutex = new BinarySemaphore(true);

 mutex.P();
 criticalSection();
 mutex.V();
```

Another variant of semaphore allows it to take arbitrary integer as its value. These semaphore are sometimes called *counting semaphores.* Their semantics is as follows:

```
public class CountingSemaphore {
  int value;

  CountingSemaphore(int initValue){ value = initValue; }

  public synchronized void P() {
     value-- ;
     try {
          if (value < 0) {
           wait();
        }
     }
     catch (InterruptedException e){ }
  }

  public synchronized void V(){
    value++;
    if (value <= 0) notify();
  }
}
```

Semaphores can be used to solve a wide variety of synchronization problems.

### 4.2.1 Producer Consumer Problem

We first consider the bounded-buffer problem. In this problem, there is a shared buffer between two processes called the producer and the consumer. The producer produces items which are *deposited* in the buffer and the consumer *fetches* items from the buffer and consumes them. For simplicity, we assume that our items are of type `double`. Since the buffer is shared it is required that each of the processes accesses the buffer in a mutually exclusive fashion. We use an array of `double` of size `sizeBuf` as our buffer. The array is used for the implementation of a `queue`. The queue has two pointers `inBuf` and `outBuf` which point to the indices in the array for depositing an item and fetching an item respectively. The variable `count` keeps track of the number of items currently in the buffer. In the problem, we see that besides mutual exclusion, there are are two additional synchronization constraints that need to be satisfied.

1. The consumer should not fetch any item from an empty buffer.
2. The producer should not deposit any item in the buffer if it is full. The buffer can get full if the producers is producing items at a greater rate than the rate at which the items are consumed by the consumer.

Such form of synchronization constraints are sometimes called *conditional synchronization.* It requires a process to wait for some condition to become true (such as the buffer to become nonempty) before continuing its operations.

```
class BoundedBuffer {
   final int sizeBuf = 10;
   double[] buffer = new double[sizeBuf];
   int inBuf = 0, outBuf = 0, count = 0;
   BinarySemaphore mutex = new BinarySemaphore(true);
   CountingSemaphore isEmpty = new CountingSemaphore(0);
   CountingSemaphore isFull = new CountingSemaphore(sizeBuf);

   public void deposit(double value) {
      isFull.P();
      mutex.P();
      buffer[inBuf] = value;
      inBuf = (inBuf + 1) % sizeBuf;
      count++;
      mutex.V();
      isEmpty.V();
   }

   public double fetch() {
      double value;
      isEmpty.P();
      mutex.P();
      value = buffer[outBuf];
      outBuf = (outBuf + 1) % sizeBuf;
      count--;
      mutex.V();
      isFull.V();
      return value;
   }
}
```

The class `BoundedBuffer` can be exercised through the following producer consumer program.

```
import java.util.Random;
class Producer implements Runnable {
   BoundedBuffer b = null;

   public Producer(BoundedBuffer initb) {
      b = initb;
      new Thread(this).start();
   }

   public void run() {
      double item;
      Random r = new Random();
      while (true) {
         item = r.nextDouble();
         System.out.println("produced item " + item);
         b.deposit(item);
         try { Thread.sleep(10); } catch (InterruptedException e) { return; }
      }
   }
}


class Consumer implements Runnable {
   BoundedBuffer b = null;

   public Consumer(BoundedBuffer initb) {
      b = initb;
      new Thread(this).start();
   }

   public void run() {
      double item;
      while (true) {
         item = b.fetch();
         System.out.println("fetched item " + item);
         try { Thread.sleep(100); } catch (InterruptedException e) { return; }
      }
   }
}


class ProducerConsumer {
  public static void main(String[] args) {
    BoundedBuffer buffer = new BoundedBuffer();
    Producer producer = new Producer(buffer);
    Consumer consumer = new Consumer(buffer);
    try { Thread.sleep(1000); } catch (InterruptedException e) { return; }
  }
}
```

### 4.2.2 The Readers Writers Problem

Next we show the solution to readers writers problem. This problem requires us to design a protocol to coordinate access to a shared database. The requirements are:

1. **No read-write conflict:** The protocol should ensure that a reader and a writer do not access the database at the same time.
2. **No write-write conflict:** The protocol should ensure that two writers do not access the database at the same time.

Further, we would like multiple readers to be able to access the database at the same time.

```
class ReaderWriter {
   int numReaders = 0;
   BinarySemaphore mutex = new BinarySemaphore(true);
   BinarySemaphore wlock = new BinarySemaphore(true);

   public void startRead() {
      mutex.P();
      numReaders++;
      if (numReaders == 1) wlock.P();
      mutex.V();
   }

   public void endRead() {
      mutex.P();
      numReaders--;
      if (numReaders == 0) wlock.V();
      mutex.V();
   }

   public void startWrite() { wlock.P(); }

   public void endWrite() { wlock.V(); }
}
```

### 4.2.3 The Dining Philosophers Problem

This problem, first posed and solved by Dijkstra is useful in bringing out problems associated with concurrent programming and symmetry. The dining problem consists of multiple philosophers who spend their time thinking and eating spaghetti. However, a philosopher requires shared resources, such as forks, to eat spaghetti. We are required to devise a protocol to coordinate access to the shared resources. A computer-minded reader may substitute processes for philosophers and files for forks. The task of eating would then correspond to an operation that requires access to shared files.

The first attempt to solve this problem could be as follows.

```
class DiningPhilosopher implements Resource {
   int n = 0;
   BinarySemaphore[] fork = null;

   public DiningPhilosopher(int initN) {
      n = initN;
      fork = new BinarySemaphore[n];
      for (int i = 0; i < n; i++) {
         fork[i] = new BinarySemaphore(true);
      }
   }

   public void acquire(int i) {
      fork[i].P();
      fork[(i+1)%n].P();
   }

   public void release(int i) {
      fork[i].V();
      fork[(i+1)%n].V();
   }

   public static void main(String[] args) {
      DiningPhilosopher dp = new DiningPhilosopher(5);
      for (int i = 0; i < 5; i++)
         new Philosopher(i, dp);
   }
}
```

This solution can deadlock when each of the philosopher is able to grab its left fork and then waits for its right neighbor to release its fork.

It is left as an exercise for the reader to design a protocol that is free from deadlocks.

## 4.3 Problems

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

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

4.3. Show that if the P and V operations of a binary semaphore are not executed atomically, then mutual exclusion may be violated.

4.4. Show that a counting semaphore can be implemented using binary semaphores. (Hint: Use a shared variable of type integer and two binary semaphores)

4.5. Give a starvation-free solution to readers writers problem using semaphores.

4.6. The following problem is known as the sleeping barber problem. There is one thread called barber. The barber cuts the hair of any waiting customer. If there is no customer, the barber goes to sleep. There are multiple customer threads. A customer waits for the barber if there is any chair left in the barber room. Otherwise, the customer leaves immediately. If there is a chair available, then the cutomer occupies it. If the barber is sleeping, then the customer wakes the barber. Assume that there are $n$ chairs in the barber shop.

Write a java class for SleepingBarber using semaphores that allows the following methods:

```
runBarber() // called by the barber thread; runs forever
hairCut() // called by the customer thread
```

How will you extend your algorithm to work for the barber shop with multiple barbers.

4.7. Give a deadlock-free solution to dining philosophers problem using semaphores. Assume that one of the philosophers picks forks in a different order.

4.8. Assume that there are three threads $P$, $Q$ and $R$ that repeatedly print "P", "Q", and "R" respectively. Use semaphores to coordinate the printing such that the number of "R" printed is always less than or equal to the sum of "P" and "Q" printed.

# Chapter 5

# Monitors

## 5.1 Monitors

The `Monitor` is a high-level object-oriented construct for synchronization in concurrent programming. A monitor can be viewed as a `class` that can be used in concurrent programs. As any `class`, a monitor has data variables and methods to manipulate that data. Because multiple threads can access the shared data at the same time, monitors support the notion of *entry* methods to guarantee mutual exclusion. It is guaranteed that at most one thread can be executing in any entry method at any time. Sometimes the phrase "thread $t$ is inside the monitor" is used to denote that thread $t$ is executing an entry method. It is clear that at most one thread can be in the monitor at any time. Thus associated with every monitor object is a queue of threads that are waiting to enter the monitor.

As we have seen before, concurrent programs also require *conditional synchronization*, that is, require that a thread wait for certian condition to become true. To address conditional synchronization, the monitor construct supports the notion of *condition variables*. A condition variable has two operations defined on it *wait()* and *signal()*. For any condition variable $x$, any thread, say $t1$, that makes a call to $x.wait()$ is blocked and put into a queue associated with $x$. When another thread, say $t2$, makes a call to $x.signal()$, if the queue associated with $x$ is nonempty, a thread is removed from the queue and made *runnable*. Since at most one thread can be in the monitor, this immediately poses a problem: Which thread should continue after the signal operation—the one that called the $signal()$ method or the thread that was waiting. There are two possible answers:

1. One of the threads that was waiting on the condition variable continues execution. Monitors that follow this rule are called *Hoare* Monitors.

2. The thread that made the signal call continues its execution. When this thread goes out of the monitor, then other threads can enter the monitor. This is the sematics followed in the Java.

One of the advantages of Hoare's Monitor is that the thread that was signalled on the condition starts its execution without intervention of any other thread. Therefore, the state in which this thread starts executing is teh same as when the signal was issued. On waking up, it can assume that the condition is true. Therefore, using Hoare's mointor, a thread's code may be

```
if (!B) x.wait();
```

Assuming that $t2$ signals only when $B$ is true, we know that $t1$ can assume $B$ on waking up. In Java-style Monitor, even though $t2$ issues the signal, it continues its execution. Thereofore when $t1$ gets its turn to execute, the condition $B$ may not be true any more. Therefore, when using Java, the threads ususally wait for the condition as

```
while (!B) x.wait();
```

The thread $t1$ can take a `signal` only as a hint that $B$ may be true. Therefore, it explicitly needs to check for truthness of $B$ when it wakes up. If $B$ is actually false, it issues the `wait()` call again.

Let us solve the problems introduced earlier with monitors.

```
class BoundedBufferMonitor {
   final int sizeBuf = 10;
   double[] buffer = new double[sizeBuf];
   int inBuf = 0, outBuf = 0, count = 0;

   public synchronized void deposit(double value) {
      while (count == sizeBuf)
         try {wait();} catch (InterruptedException e){}
      buffer[inBuf] = value;
      inBuf = (inBuf + 1) % sizeBuf;
      count++;
      if (count==1) notify();
   }

   public synchronized double fetch() {
      double value;
      while (count == 0)
         try {wait();} catch (InterruptedException e){}
      value = buffer[outBuf];
      outBuf = (outBuf + 1) % sizeBuf;
      count--;
      if (count==sizeBuf-1) notify();
      return value;
   }
}
```

Now let us revisit the dining philosophers problem.

```
class DiningMonitor implements Resource {
   int n = 0;
   int state[] = null;
   static final int THINKING = 0, HUNGRY = 1, EATING = 2;

   public DiningMonitor(int initN) {
      n = initN;
      state = new int[n];
      for (int i = 0; i < n; i++) state[i] = THINKING;
   }

   int left(int i) { return (n + i - 1) % n; }
   int right(int i) { return (i + 1) % n; }

   public synchronized void acquire(int i) {
      state[i] = HUNGRY;
      test(i);
      while (state[i] != EATING)
         try {wait();} catch (InterruptedException e){}
   }

   public synchronized void release(int i) {
      state[i] = THINKING;
      test(left(i));
      test(right(i));
   }

   void test(int i) {
      if ((state[left(i)] != EATING) &&
          (state[i]==HUNGRY) &&
          (state[right(i)] != EATING)) {
               state[i] = EATING;
               notifyAll();
          }
   }


   public static void main(String[] args) {
      DiningMonitor dm = new DiningMonitor(5);
      for (int i = 0; i < 5; i++)
         new Philosopher(i, dm);

   }
}
```

## 5.2 Problems

5.1. Write a monitor for the sleeping barber problem.

5.2. Write a monitor that allows a process to sleep till a counter reaches certain value. The counter class allows two operations:`increment()` and `sleepUntil(int x)`.

5.3. Write a Java class for *BoundedCounter* with a minimum and a maximum value. This class provides two methods: `increment()` and `decrement()`. Decrement at the minimum value and increment at the maximum value result in the calling thread waiting until the operation can be performed without violating the bounds on the counter.

# Chapter 6

# Synchronization of Threads in Java

## 6.1 Examples in Java

In this section we give several examples of concurrent programming in Java. First we give a thread-safe implementation of a queue based on a linked list.

```
public class ListQueue {
        Node head = null, tail = null;

        public synchronized void Enqueue(String data) {
                Node temp = new Node();
                temp.data = data;
                temp.next = null;
                if (tail == null){
                        tail = temp;
                        head = tail;
                } else {
                        tail.next = temp;
                        tail = temp;
                }
                notify();
        }

        public synchronized String Dequeue() {
                while (head == null)
                   try {wait();} catch (InterruptedException e){}
                String returnval = head.data;
                head = head.next;
                return returnval;
        }
}
```

In java, associated with each object is a lock. Whenever a thread needs to execute a synchronized method, it needs to get that lock. The keyword `synchronized` can be used with any statement as `synchronized (expr) statement`. The expression `expr` must result in a reference to an object on evaluation. The semantics of the above construct is that the `statement` can be executed only when the thread has the lock for the object given by the `expr`. Thus a synchronized method

```
public synchronized void method()
{
 body();
}
```

can simply be viewed as a short form for

```
public void method() {
  synchronized (this) {
     body();
  }
}
```

Just as non-static methods can be `synchronized`, so can the static methods. A `synchronized` static method results in a classwide lock.

One also needs to be careful with inheritance. When an extended class overrides a `synchronized` method with an unsynchronized method, the method of the original class stays synchronized. Thus, any call to `super.method()` will result in synchronization.

## 6.2 Dangers of Deadlocks

Since every synchronized call requires a lock, if the programmer is not careful he can introduce deadlocks. For example, consider the following class that allows a cell to be swapped with the other cell.

```
class BCell { // can result in deadlocks
     int value;
     public synchronized int getValue() { return value;}
     public synchronized void setValue(int i) { value = i;}
     public synchronized void swap(BCell x){
     int temp = getValue();
        setValue(x.getValue());
        x.setValue(temp);
     }
}
```

The program that avoids the deadlock is given below.

```
class Cell {
     int value;
     public synchronized int getValue() { return value;}
     public synchronized void setValue(int i) { value = i;}
     protected synchronized void doSwap(Cell x){
     int temp = getValue();
        setValue(x.getValue());
        x.setValue(temp);
     }
     public void swap(Cell x){
        if (this == x) return;
        else if (System.identityHashCode(this)
               < System.identityHashCode(x))
```

```
                doSwap(x);
            else
                x.doSwap(this);
        }
}
```

Some other useful concepts in Java are as follows.

1. The *join()* method allows one thread to wait for the other thread to die. If thread $t1$ calls `t2.join()`, then $t1$ will get blocked till $t2$ dies.
2. The `interrupt()` method allows a thread to be interrupted. If thread $t1$ calls $t2.interrupt()$, then $t2$ gets an InterruptedException.
3. The `yield()` method allows a thread to yield the CPU to other threads temporarily. It does not require any interaction with other threads and a program without `yield()` would be functionally equivalent to `yield()` call. A thread may chose to `yield()` if it is waiting for some data to become available from say `InputStream`.
4. The keyword `volatile` declares a variable to be possibly updated by other thread. This means that if the compiler sees some value assigned to that variable, it cannot assume that a future read will return that value.
5. The method `holdsLock(x)` returns true if the current thread holds the monitor lock of the object $x$.

## 6.3 Thread Groups

Sometimes a programmer may prefer to work on groups of threads instead of individual threads. The `ThreadGroup` class in Java allows programmer to do this. A ThreadGroup consists of one or more threads.

# Chapter 7

# Consistency Conditions

## 7.1 Introduction

In the presence of concurrency, one needs to revisit the correctness conditions of executons, i.e., which behaviors are correct when multiple processes invoke methods concurrently on a shared object. Let us define a concurrent object as one that allows multiple processes to execute its operations concurrently. For example, a concurrent queue in a shared memory system may allow multiple processes to invoke *enqueue* and *dequeue* operations. The natural question, then, is to define which behavior of the object under concurrent operations is consistent (or correct). Consider the case when a process $P$ first enqueues $x$ and then dequeues while process Q concurrently enqueues $y$. Is the queue's behavior acceptable if process $P$ gets $y$ as the result of dequeue? The objective of this chapter is to clarify such questions.

The notion of consistency is also required when objects are *replicated.* There are two reasons for replicating objects— fault tolerance and efficiency. If an object has multiple copies, then even when a node that contains one of the copies of the object goes down, the system may still be able to function correctly by using other copies. Second, accessing a remote object may incur a large overhead because of communication delays. Suppose that we knew that most accesses of the object are for *read-only.* In this case, it may be better to replicate that object. A process can read the value from the replica that is closest to it in the system. Of course, when we perform a *write* on this object, we have to worry about consistency of data. This again requires us to define data consistency. Observe that any system that uses *caches*, such as multi-processor systems, has to grapple with similar issues.

This chapter is organized as follows. In Section 7.2, we define our system model formally. Section 7.3 describes *sequential consistency* condition. Section 7.4 describes a stronger consistency condition called *linearizability* that has the attractive property that it can be implemented on a per-object basis, that is, if implementation of every object in the system is linearizable, then the entire collection is also linearizable. Finally, in Section 7.5 we describe some consistency conditions that are weaker than sequential consistency.

## 7.2 System Model

**Objects and Processes**

A *concurrent system* consists of a finite set of *sequential processes* that communicate through *concurrent objects.* Each object has a name and a type. The type defines the set of possible values for objects of this type and the set of primitive operations that provide the only means to manipulate objects of this type. Execution of an operation takes some time; this is modeled by two events, namely an *invocation* event and a *response* event. Let *op(arg)* be an operation on object $x$ issued at $P$; *arg* and *res* denote *op*'s input and output parameters, respectively. Invocation and response events $inv(op(arg))$ $on\ x\ at\ P$ and $resp(op(res))$ $from\ x\ at\ P$ will be abbreviated as *inv(op)* and *resp(op)* when parameters, object name and process identity are not necessary. For any operation $e$, we use $proc(e)$ to denote the process and $object(e)$ to denote the set of objects associated with the operation. In this chapter, we assume that all operations are applied by a single process on a single object. In the problem set, we explore generalizations to operations that span multiple objects.

**Histories**

A *history* is an execution of a concurrent system modeled by a directed acyclic graph $(H, <_H)$, where $H$ is the set of operations and $<_H$ is an irreflexive transitive relation that captures the "occurred before" relation between operations. Sometimes we simply use $H$ to denote the history when $<_H$ is clear from the context. Formally, for any two operations $e$ and $f$:

$$e <_H f \quad \text{if} \quad resp(e) \text{ occurred before } inv(f) \text{ in real-time.}$$

Observe that this relation includes the following relations.

`Process Order`: $(proc(e) = proc(f)) \wedge (resp(e)$ occurred before $inv(f))$.

`Object Order`: $(object(e) \cap object(f) \neq \emptyset) \wedge (resp(e)$ occurred before $inv(f))$.

A process subhistory $H|P$ ($H$ at $P$) of a history $H$ is the subposet of all events $e$ in $H$ such that $proc(e) = P$. An object's subhistory is defined in a similar way for an object $x$; denoted by $H|x$ ($H$ at $x$). Two histories $H$ and $H'$ are *equivalent* if they are composed of exactly the same set of invocation and response events.

A history $(H, <_H)$ is a *sequential* history if $<_H$ is a total order. Such a history would happen if there was only one sequential process in the system. A sequential history is *legal* if it meets the sequential specification of all the objects. For example, if we are considering a read-write register $x$ as a shared object, then a sequential history is legal if for every read operation that returns its value as $v$, there exists a write on that object with value $v$, and there does not exist another write operation on that object with a different value between the write and the read operation. For a sequential queue, if the queue is nonempty then a `dequeue` operation should return the item that was enqueued earliest and has not been already dequeued. If the queue is empty, then the dequeue operation should return null.

Our goal is to determine whether a given *concurrent* history is correct.

## 7.3 Sequential Consistency

**Definition 7.1 (Sequentially Consistent)** *A history $(H, <_H)$ is sequentially consistent if there exists a sequential history $S$ equivalent to $H$ such that $S$ is legal and it satisfies process order.*

Thus a history is sequentially consistent if its execution is equivalent to a legal sequential execution and each process's behavior is identical in the concurrent and sequential execution. In the following histories, $P, Q$, and $R$ are processes operating on shared registers $x$, $y$, and $z$. We assume that all registers have 0 initially. The response of a read operation is denoted by $ok(v)$, where $v$ is the value returned, and the response of a write operation is denoted by $ok()$.

1. $H_1 = P\ write(x, 1),\ Q\ read(x),\ Q\ ok(0),\ P\ ok()$.
   Note that $H_1$ is a concurrent history. $Q$ invokes the $read(x)$ operation before the $write(x, 1)$ operation is finished. Thus $write(x, 1)$ and $read(x)$ are concurrent operations in $H_1$.

   $H_1$ is sequentially consistent because it is equivalent to the following legal sequential history.
   $Q\ read(x),\ Q\ ok(0),\ P\ write(x, 1),\ P\ ok()$.

2. $H_2 = P\ write(x, 1),\ P\ ok(),\ Q\ read(x),\ Q\ ok(0)$
   Somewhat surprisingly, $H_2$ is also sequentially consistent. Even though $P$ got the response of its write before $Q$, it is okay for $Q$ to have read an old value. Note that $H_2$ is a sequential history but not legal. However, it is equivalent to the following legal sequential history.
   $Q\ read(x),\ Q\ ok(0),\ P\ write(x, 1),\ P\ ok()$.

3. $H_3 = P\ write(x,1),\ Q\ read(x),\ P\ ok(),\ Q\ ok(0),\ P\ read(x),\ P\ ok(0)$
   $H_3$ is not sequentially consistent. Any sequential history equivalent to $H_3$ must preserve process order. Thus the *read* operation by $P$ must come after the *write* operation. This implies that the *read* cannot return 0.

4. $H_4 = P\ write(x,1),\ Q\ read(x),\ P\ ok(),\ Q\ ok(2)$
   $H_4$ is also not sequentially consistent. There is no *legal* sequential history equivalent to $H_4$ because the read on $Q$ returns 2 which was never written on the register (and was not the initial value).

## 7.4 Linearizability

Linearizability is a stronger consistency condition than sequential consistency. Intuitively, an execution of a concurrent system is linearizable if it could appear to an external observer as a sequence composed of the operations invoked by processes that respect object specifications and real-time precedence ordering on operations. So, linearizability provides the illusion that each operation on shared objects issued by concurrent processes takes effect instantaneously at some point between the beginning and the end of its execution. Formally,

**Definition 7.2 (Linearizable)** *A history $(H, <_H)$ is linearizable if there exists a sequential history $(S, <)$ equivalent to $H$ such that $S$ is legal and it preserves $<_H$.*

Since $<_H$ includes process order, it follows that a linearizable history is always sequentially consistent.
Let us reexamine some histories that we saw earlier.

1. $H_1 = P\ write(x,1),\ Q\ read(x),\ Q\ ok(0),\ P\ ok()$
   $H_1$ is linearizable because the following legal sequential history,
   $Q\ read(x),\ Q\ ok(0),\ P\ write(x,1),\ P\ ok()$
   preserves $<_H$.

2. $H_2 = P\ write(x,1),\ P\ ok(),\ Q\ read(x),\ Q\ ok(0)$
   $H_2$ is sequentially consistent but not linearizable. The legal sequential history used for showing sequential consistency does not preserve $<_H$.

A key advantage of linearizability is that it is a local property, i.e., if for all objects $x$, $H|x$ is linearizable, then $H$ is linearizable. Sequential consistency does not have this property. For example, consider two concurrent queues $s$ and $t$. Process $P$ enqueues $x$ in $s$ and $t$. Process $Q$ enqueues $y$ in $t$ and then in $s$. Now $P$ gets $y$ from deq on $s$ and $Q$ get $x$ when it does deq on $t$.

Consider the following history:
$H = P\ s.enq(x), P\ s.ok(), Q\ t.enq(y),\ Q\ t.ok(), P\ t.enq(x),\ P\ t.ok(), Q\ s.enq(y),\ Q\ s.ok(),$
$P\ s.deq(),\ P\ s.ok(y), Q\ t.deq(),\ P\ t.ok(x)$

If we focus simply on the queue $s$, we get the history:
$H|s = P\ s.enq(x), P\ s.ok(), Q\ s.enq(y),\ Q\ s.ok(), P\ s.deq(),\ P\ s.ok(y).$

If we focus simply on the queue $t$, we get the history:
$H|t = Q\ t.enq(y),\ Q\ t.ok(), P\ t.enq(x),\ P\ t.ok(), Q\ t.deq(),\ P\ t.ok(x)$

Both $H|s$ and $H|t$ are sequentially consistent but $H$ is not.

To see the linearizability is a local property assume that $(S_x, <_x)$ is a linearization of $H|x$, that is, $(S_x, <_x)$ is a sequential history that is equivalent to $H|x$. We construct an acyclic graph that orders all operations on any object and also preserves occurred before order $<_H$. Any sort of this graph will then serve as a linearization of $H$. The graph is constructed as follows. The vertices are all the operations. The edges are all the edges given by union of $<_x$'s and $<_H$. This graph totally orders all operations on any object. Moreover, it preserves $<_H$. The only thing that remains to be shown is that it is acyclic. Since $<_x$'s are acyclic, it follows that any cycle, if it exists, must involve at least two objects.

We will show that cycle in this graph implies a cycle in $<_H$. If any two consecutive edges in the cycle are due to just $<_x$ or just $<_H$, then they can be combined. Note that $e <_x f <_y g$ for distinct objects $x$ and $y$ is not possible because all operations are unary ($e <_x f <_y g$ implies that $f$ operates on both $x$ and $y$). Now consider, any sequence

of edges such that $e <_H f <_x g <_H h$.
$e <_H f$ implies $res(e)$ precedes $inv(f)$ { definition of $<_H$ }
$f <_x g$ implies $inv(f)$ precedes $res(g)$ { $<_x$ is a total order }
$g <_H h$ implies $res(g)$ precedes $inv(h)$ { definition of $<_H$ }.

These can be combined to give that $res(e)$ precedes $inv(h)$. Therefore, $e <_H h$. Thus any cycle in the graph can be reduced to a cycle in $<_H$, a contradiction because $<_H$ is irreflexive.

So far we have only looked at consistency conditions for complete histories, i.e. histories in which every *invocation* operation has a corresponding *response* operation. We can generalize the consistency conditions for partial histories as follows. A partial history $H$ is linearizable if there exists a way of completing the history by appending response events such that the complete history is linearizable. For example, consider the following history:
$H_3 = P\ write(x, 1),\ Q\ read(x),\ Q\ ok(0)$
is linearizable because $P\ write(x, 1),\ Q\ read(x),\ Q\ ok(0), P\ ok()$
is linearizable. This generalization allows us to deal with systems in which some processes may fail and consequently some response operations may be missing.

## 7.5 Other Consistency Conditions

Although we have focused on sequential consistency and linearizability, there are many consistency conditions that are weaker than sequential consistency. A weaker consistency condition allows more efficient implementation at the expense of increased work by the programmer, who has to ensure that the application works correctly despite weaker consistency condition.

Consider a program consisting of two processes $P$ and $Q$ with two variables $x$ and $y$. Assume that the initial values of $x$ and $y$ are both 0. $P$ writes 1 in $x$ and then reads the value of $y$; $Q$ writes 1 in $y$ and then reads the value of $x$. Strong consistency conditions like sequential consistency or linearizability prohibit the results of both reads from being 0. However, if we assume that the minimum possible time to read plus the minimum possible time to write is less than the communication latency, then both reads must return 0. The latency is the information delivery time and each processor cannot possibly know of the events that have transpired at the other processor. So, no matter what the protocol is, if it implements sequential consistency, it must be slow.

Causal consistency is weaker than sequential consistency. Causal consistency allows for implementation of read and write operations in a distributed environment that do not always incur communication delay, that is, causal consistency allows for cheap read/write operations.

With sequential consistency, all processes agree on the same legal sequential history $S$. The agreement defined by causal consistency is weaker. Given a history $H$, it is not required that two processes $P$ and $Q$ agree on the same ordering for the write operations, which are not ordered in $H$. The reads are, however, required to be legal. Each process considers only those operations that can affect it, that is, its own operations and only write operations from other processes. Formally, for read-write objects causal consistency can be defined as follows.

**Definition 7.3 (Causally Consistent)** *A history $(H, <_H)$ is* causally consistent *if for each process $P_i$, there is a legal sequential history $(S_i, <_{S_i})$ where $S_i$ is the set of all operations of $P_i$ and all write operations in $H$, and $<_{S_i}$ respects the following order:*

`Process Order`*: If $P_i$ performs operation $e$ before $f$, then $e$ is ordered before $f$ in $S_i$.*

`Object Order`*: If any process $P$ performs a write on an object $x$ with value $v$ and another process $Q$ reads that value $v$, then the write by $P$ is ordered before read by $Q$ in $S_i$.*

Intuitively, causal consistency requires that causally related writes are seen by all processes in the same order. The concurrent writes may be seen in different order by different processes.

It can be proved that sequential consistency implies causal consistency but the reverse does not hold. As an example consider history $H_1$ in which $P_1$ does $w_1(x, 1),\ r_1(x, 2)$ and $P_2$ does $w_2(x, 2),\ r_2(x, 1)$.

The history is causally consistent because the following serializations exist:
$S_1 = w_1(x, 1), w_2(x, 2), r_1(x, 2)$
$S_2 = w_2(x, 2), w_1(x, 1), r_2(x, 1)$

| **Consistency** | **Legal History** | **Order Preserved** |
|---|---|---|
| Linearizability | Global | Occurred before order |
| Sequential | Global | Process order |
| Causal | Per process | Process, object order |
| FIFO (Problem 7.4) | Per process | Process order |

Figure 7.1: Summary of consistency conditions

Thus we only require that there is a legal sequential history for every process and not one for the entire system. $P_1$ orders $w_1$ before $w_2$ in $S_1$ and $P_2$ orders $w_2$ before $w_1$ but that is considered causally consistent because $w_1$ and $w_2$ are concurrent writes. It can be easily proved that history $H_1$ is not sequentially consistent.

The following history is not even causally consistent. Assume that the initial value of $x$ is 0. The history at process $P$ is

$H|P = P\ r(x,4), P\ w(x,3)$.

The history at process $Q$ is

$H|Q = Q\ r(x,3), Q\ w(x,4)$.

Since, $Q$ reads the value 3 and then writes the value of $x$, the write by $Q$ should be ordered after the write by $P$. $P$'s read is ordered before its write; therefore, it cannot return 4 in a causally consistent history.

The table in Figure 7.1 summarizes the requirements of all consistency conditions considered in this chapter. The second column tells us whether the equivalent legal history required for the consistency condition is global or not. The third column tells us the requirement on the legal history in terms of the order preserved. For example, linearizability requires that there be a single equivalent legal history that preserves the occured before order.

## 7.6 Problems

7.1. Consider a concurrent stack. Which of the following histories are linearizable? Which of the them are sequentially consistent? Justify your answer.
a) $P\ push(x), P\ ok(), Q\ push(y), Q\ ok(), P\ pop(), P\ ok(x)$
b) $P\ push(x), Q\ push(y), P\ ok(), Q\ ok(), Q\ pop(), Q\ ok(x)$

7.2. Assume that all processors in the system maintain a cache of a subset of objects accessed by that processor. Give an algorithm that guarantees sequential consistency of reads and writes of the objects.

7.3. Assume that you have an implementation of a concurrent system that guarantees causal consistency. Show that if you ensure that the system does not have any concurrent writes, then the system also ensures sequential consistency.

7.4. FIFO consistency requires that the writes done by the same process must be seen in the same order. Writes done by different processes may be seen in different order. Show a history that is FIFO consistent but not causally consistent.

## 7.7 Bibliographic Remarks

Sequential consistency was first proposed by Lamport in [Lam79]. The notion of linearizability for read/write registers was also introduced by Lamport in [Lam86] under the name of atomicity. The concept was generalized to arbitrary data types and termed as linearizability by Herlihy and Wing in [HW90]. Causal consistency was introduced by Hutto and Ahamad [HA90].

# Chapter 8

# Wait-Free Synchronization

## 8.1 Introduction

The synchronization mechanisms that we have discussed so far are based on locking data structures during concurrent accesses. The lock-based synchronization mechanisms are inappropriate in fault-tolerance and real-time applications. When we use lock-based synchronization, if a process fails inside the critical section, then all other processes cannot perform their own operations. Even if no process ever fails, lock-based synchronization is bad for real-time systems. Consider a thread serving a request with short deadline. If another thread is inside the critical section and is slow, then this thread may have to wait and therefore miss its deadline. Using locks also require the programmer to worry about deadlocks. In this chapter, we introduce synchronization mechanisms which do not use locks and are therefore called *lock-free*. If a lock-free synchronization also guarantees that each operation finishes in a bounded number of steps, then it is called *wait-free* synchronization.

To illustrate lock-free synchronization, we will implement various concurrent objects. The implementation may use other simpler concurrent objects. One dimension of simplicity of an object is based on whether it allows multiple readers or writers. We use SR, MR, SW, and MW to denote single reader, multiple reader, single writer and multiple writer respectively. The other dimension is the consistency condition satisfied by the register. For a single writer register, Lamport has defined the following consistency conditions. A register is *safe* if a read that does not overlap with the write returns the most recent value. If the read overlaps with the write, then it can return any value. A register is *regular* if it is safe and when the read overlaps with one or more writes, it returns either the value of the most recent write that preceded the read or the value of one of the overlapping writes. A register is *atomic* if its histories are linearizable. Clearly, atomicity implies regularity which implies safety.

Surprisingly, it is possible to build a multiple reader multiple writer atomic multivalued register from single reader single writer safe boolean register. This can be achieved by the following chain of constructions:

1. SRSW safe boolean register to
2. SRSW regular boolean register to
3. SRSW regular register to
4. SRSW atomic register to
5. MRSW atomic register to
6. MRMW atomic register

We show some of these constructions next.

## 8.2 Building regular SRSW Register from a safe SRSW Register

Assume that we have a safe single reader single writer boolean register. We show that a regular SRSW register can be built from it.

```
class RegularBoolean {
  boolean prev; // not shared
  boolean value; // assume safe

  public boolean getValue() { return value; }

  public void setValue(boolean b) {
     if (prev != b) { value = b; prev = b; }
  }
}
```

The new register ensures that the writer does not access *value* if the previous value was the same. Thus an overlapping read will return the correct value. If the new value is different, then the read can return arbitrary value from {true, false}, but that is still acceptable because one of them is the previous value and the other is the new value.

## 8.3 Building SRSW Multivalued Register from SRSW Boolean Register

Assume that we have a single reader single writer boolean register. This register maintains a single bit and guarantees that in spite of concurrent accesses by a single reader and a single writer it will result only in linearizable concurrent histories. We now show that using this register, we can implement a multi-valued SRSW register. The implementation is shown below.

```
class MultiValued {
  int n = 0;
  boolean A[] = null;

  public MultiValued(int maxVal, int initVal){
    n = maxVal;
    A = new boolean[n];
    for (int i=0; i<n; i++) A[i] = false;
    A[initVal] = true;
  }

  public int getValue() {
    int j = 0;
    while (!A[j]) j++;
    int v=j;
    for (int i=j-1; i>=0; i--)
      if (A[i]) v = i;
    return v;
  }

  public void setValue(int x){
      A[x] = true;
      for (int i=x-1; i>=0; i--)
        A[i]= false;
  }
}
```

The implementation requires an array of $maxVal$ SRSW boolean registers to allow values in the range $0..maxVal-1$. To write the value $x$, the writer makes the $x$ bit true and then makes all the previous values $false$. The idea is that the reader should return the index of the first true bit. The straightforward solution of the writer updating the array in the forward direction till it reaches the required index and the reader also scanning in the forward direction for the first true bit does not work. It can happen that the reader does not find any true bit (come up with an execution to show this!). So the first idea we will use is that the writer will first set the required bit to true and then traverse the array in th *backward* direction setting all previous bits to false. The reader scans for the true bit in the forward direction. With this strategy, the reader is guaranteed to find at least one bit to be true. Further this bit would correspond to the most recent write before the read or one of the concurrent writes. Therefore, this will result in at least a *regular* register. However, a single scan by the reader does not result in a linearizable implementation. To see this assume that the initial value of the register is 5 and the writer first writes the value 1 and then the value 4. These steps will results in:
1. writer sets A[1] to true.
2. writer sets A[4] to true.
3. writer sets A[1] to false.

Now assume that concurrent with these two writes, a reader performs two read operations. Since the initial value of A[1] is false, the first read may read $A[4]$ as the first bit to be true. The second read may happen between steps 2 and 3 above resulting in the second read returning 1. The resulting concurrent history is not linearizable.

In our implementation, the reader first does a forward scan and then does a backward scan to find the first bit which is true. Two scans are sufficient to guarantee linearizability.

## 8.4 Building MRSW Register from SRSW Registers

We now build a MRSW from a SRSW register. The simplest strategy would be to have an array of $n$ SRSW registers, say $V[n]$ for $n$ readers. The writer would write to all $n$ registers and the reader $r$ can read from its own register. This does not result in a linearizable implementation. Assume that initially all registers are 5, the initial value of the MRSW register. Assume that writer is writing a value 3. Concurrent to this write, two reads occur. It is possible for the first read to get the new value 3 and the second read to get the old value 5 contradicting linearizability.

To solve this problem, we require a reader to read not only the value written by the writer but also all the values read by other readers so far. The reader is supposed to return the most recent value. How does the reader determine the most recent value? To solve this problem we use a sequence number associated with each value. The writer maintains the sequence number and writes this sequence number with any value that it writes.

The algorithm is shown below.

```
class SRSW {
    public int val=0;
    public int ts=0;
    public void setValue(int x, int seq){ val = x; ts = seq;}
}

class MultiReader {
  int n = 0;
  SRSW V[] = null;// value written by the writer for reader i
  SRSW Comm[][] = null; // for communication between readers
  int seqNo = 0;

  public MultiReader(int readers, int initVal){
    n = readers;
    V = new SRSW[n];
    for (int i=0; i<n; i++) V[i].setValue(initVal,0);
    Comm = new SRSW[n][n];
    for (int i=0; i<n; i++)
       for (int j=0; j<n; j++)
           Comm[i][j].setValue(initVal,0);
  }

  public int getValue(int r) {//reader r reads
    SRSW tsv = V[r]; // tsv is local
    for (int i=0; i<n; i++)
        if (Comm[i][r].ts > tsv.ts) tsv = Comm[i][r];
    for (int i=0; i<n; i++) Comm[r][i].setValue(tsv.val, tsv.ts);
    return tsv.val;
  }

  public void setValue(int x){
     seqNo++;
     for (int i=0; i<n; i++)
       V[i].setValue(x,seqNo);
  }
}
```

Since we can only use SRSW objects we are forced to keep $O(n^2)$ $Comm$ registers for informing readers what other readers have read. $Comm[i][j]$ is used by the reader $i$ to inform the value it read to the reader $j$.

The reader simply reads its own register and also what other readers have read and returns the latest write. The writer simply increments the sequence number and writes the value in all $n$ registers.

## 8.5 Building MRMW Register from MRSW Registers

The construction of MRMW register from MRSW is simpler. We use $n$ MRSW registers for $n$ writers. Each writer writes in its own register. The only problem for the reader to solve is which of the write it should choose. We use the idea of sequence numbers as in the previous implementation. There is only one problem. Previously, there was a single writer and therefore we could guarantee that all writes had different sequence numbers. Now we have multiple writers, so how do we assign unique sequence number to each write. First, we require a writer to read all the sequence numbers and then choose its sequence number to be larger than the maximum sequence number it read. Second, to avoid the problem of two concurrent writers coming up with the same sequence number, we attach process id with the sequence number. Now two writes with the same sequence number can be ordered based on process ids. Furthermore, we have the following guarantee: if one write completely precedes another write then the sequence number associated with the first write will be smaller than that with the second write. The reader reads all the values written by various writers and chooses the one with the highest timestamp. The algorithm is shown below.

```
class MRSW {
    public int val=0, ts=0, pid=0;
    public void setValue(int x, int seq, int id){ val = x; ts = seq; pid=id;}
}
class MultiWriter {
  int n = 0;
  MRSW V[] = null;// value written by the writer i

  public MultiWriter(int writers, int initVal){
    n = writers;
    V = new MRSW[n];
    for (int i=0; i<n; i++) V[i].setValue(initVal,0,0);
  }
  public int getValue() {
    MRSW tsv = V[0]; // tsv is local
    for (int i=1; i<n; i++)
      if ((tsv.ts < V[i].ts)||((tsv.ts == V[i].ts)&&(tsv.pid < V[i].pid)))
           tsv = V[i];
    return tsv.val;
  }
  public void setValue(int w, int x){ // writer w
     int maxseq = V[0].ts;
     for (int i=1; i<n; i++)
       if (maxseq < V[i].ts) maxseq = V[i].ts;
     V[w].setValue(x,maxseq+1, w);
  }
}
```

## 8.6 Atomic Snapshots

All our algorithms so far handled single values. Consider an array of values that we want to read in an atomic fashion. We will assume that there is a single reader and a single writer but while the array is being read, the writes to individual locations may be going on concurrently. Intuitively, we would like our readArray operation to behave as if it took place instantaneously. A simple scan of the array will not work. Assume that the array has three locations initially all with value 0 and that a readArray operation and concurrently two writes take place. The first write updates the first location to 1 and the second write updates the second location to 1. A simple scan may return the value of array as $[0, 1, 0]$. However, the array went through the transitions $[0, 0, 0]$ to $[1, 0, 0]$, and then to $[1, 1, 0]$. Thus, the value returned by readArray is not consistent with linearizability. A construction that provides a readArray operation with consistent values in spite of concurrent writes with it is called an atomic snapshot operation. Such an operation can be used in getting a checkpoint for fault-tolerance applications.

We first show a *lock-free* construction of atomic snapshots. This construction is extremely simple. First, to determine if a location in the array has changed view append each value with the sequence number as seen earlier. Now, the readArray operation reads the array twice. If none of the sequence numbers changed, then we know that there exists a time interval in which the array did not change. Hence the copy read is consistent. This construction is not wait free because if a conflict is detected the readArray operation has to start all over again. There is no upper bound on the number of times this may have to be done.

```
public class LockFreeSnapshot {
   int n =0;
   SRSW[] V;
   public void LockFreeSnapshot(int initN) {
      n = initN;
      V = new SRSW[n];
   }

   public void writeLoc(int k, int x){
       int seq = V[k].ts;
       V[k].setValue(x,seq+1);
   }

   public SRSW [] readArray(){
     SRSW[] W = new SRSW[n];// W is local
     boolean done = false;
     while (!done) {
        for (int i=0;i<n; i++) W[i].setValue(V[i].val, V[i].ts);
        done = true;
        for (int i=0;i<n; i++)
           if (W[i].ts != V[i].ts) {
                 done = false;
                 break;
           }
      }
      return W;
    }
}
```

The above construction is not wait-free because a readArray operation may be starved by the update operation. We do not go into details here but this and many other lock-free constructions can be turned into wait-free constructions by using the notion of "helping" moves. The main idea is that a thread tries to help pending operations. For example, if the thread wanting to perform an *update* operation helps another concurrent thread which is trying to do a readArray operation, then we call it a helping move. Thus one of the ingredients in constructing waitFreesnapshot would require the update operation to also scan the array.

## 8.7 Problems

8.1.

## 8.8 Bibliographic Remarks

The notion of *safe*, *regular*, and *atomic* registers was first proposed by Lamport in [Lam86] who also gave many of the constructions provided here. The reader should also consult [AW98] (chapter 10).

# Chapter 9

# Wait-free Consensus

## 9.1 Introduction

So far we have given many wait-free constructions of a concurrent object using other simpler concurrent objects. The natural question that arises is whether it is always possible to build a concurrent object from concurrent objects. Are there concurrent objects powerful enough so that they can be used to implement all other concurrent objects?

It turns out that *consensus* is a fundamental problem useful for analyzing such problems. The consensus problem requires a given set of processes to agree on an input value. Each process has a value input to it that it can propose. For simplicity, we will restrict the range of input values to a single bit. The processes are required to run a protocol so that they decide on a common value. Thus, any object that implements consensus supports the following methods.

```
public interface Consensus {
   public void propose(int pid, int value);
   public int decide(int pid);
}
```

The requirements are

- **Agreement**: Two processes cannot decide different values.
- **Validity**: The decided value must be one of the proposed values.
- **Wait-free**: Each process decides the value after a finite number of steps. This should be true without any assumption on relative speeds of processes.

Every concurrent object $O$ is defined to have a consensus number equal to the largest number of processes that can use $O$ to solve the consensus problem. If any number of processes can solve the consensus problem using $O$, then its consensus number is $\infty$ and the object $O$ is called the universal object.

Now if we could show that some concurrent object $O$ has consensus number $m$ and another concurrent object has consensus number $m' > m$, then it is clear that there can be no wait-free implementation of $O'$ using $O$. Surprisingly, the converse is true as well: If $O'$ has consensus number $m' \leq m$, then $O'$ can be implemented using $O$.

We begin by showing that linearizable (or atomic) registers have consensus number 1. Clearly, an atomic register can be used to solve the consensus problem for a single process. The process simply decides its own value. Therefore, the consensus number is at least 1. Now we show that there does not exist any protocol to solve the consensus using atomic registers.

The argument for nonexistence of a consensus protocol hinges on the concepts of a *bivalent state* and a *critical state*. A protocol is in a bivalent state if both the values are possible as decision values starting from that global state. A bivalent state is a critical state if all possible moves from that state result in non-bivalent states. Any initial state in which processes have different proposed value is bivalent because there exist at least two runs from that state which result in different decision values. In the first run, the process with input 0 gets to execute and all other processes are very slow. Due to wait-freedom, this process must decide and it can only decide on 0 to ensure validity. A similar run exists for a process with its input as 1.

Starting from a bivalent initial state, and letting any process move that keeps the state as bivalent, we must hit a critical state; otherwise, the protocol can run forever. We show that even in a 2-process system, atomic registers cannot be used to go to non-bivalent states in a consistent manner. We do a case analysis of events that can be done by two processes, say $P$ and $Q$ in a critical state $S$. Let $e$ be the event at $P$ and event $f$ be at $Q$ be such that $e(S)$ has a different decision value from $f(S)$. We now do a case analysis:

- *Case 1:* $e$ and $f$ are on different registers
  Then, both $ef$ and $fe$ are possible in the critical state $S$. Further, the state $ef(S)$ is identical to $fe(S)$ and therefore cannot have different decision values. But, we assumed that $f(S)$ and $e(S)$ have different decision values which implies that $e(f(S))$ and $f(e(S))$ have different values.

- *Case 2:* Either $e$ or $f$ is a read.
  Assume that $e$ is a read. Then the state of $Q$ does not change when $P$ does $e$. Therefore, the decision value for $Q$ from $f(S)$ and $e(S)$, if it ran alone, would be the same; a contradiction.

- *Case 3:* Both $e$ and $f$ are writes on the same register
  Again the state $f(S)$ and $f(e(S))$ are identical for $Q$ and should result in the same decision value.

This implies that there is no consensus protocol for 2 processes that uses just atomic registers. Therefore, the consensus number for atomic registers is 1.

Now let us look at a concurrent object which can do both read and write in one operation. Let us define a `TestAndSet` object as one that provides the test and set instruction discussed in Chapter 4. The implementation is shown below.

```
public class TestAndSet {
  int myValue = 0;

  public synchronized int testAndSet(int newValue) {
      int oldValue = myValue;
      myValue = newValue;
      return oldValue;
 }
}
```

We now show that two threads can indeed use this object to achieve consensus.

```
class TestSetConsensus implements Consensus {
    TestAndSet x;
    int proposed[] = {0,0};

    // assumes pid is 0 or 1
    public void propose(int pid, int v){ proposed[pid] = v; }
    public int decide(int pid){
       if (x.testAndSet(pid) == -1) return proposed[pid];
       else return proposed[1-pid];
    }
}
```

By analyzing the testAndSet operations on the critical state one can show that the TestAndSet cannot be used to solve the consensus problem on three processes.

Finally, we show universal objects. One such universal object is CompareAndSwap. The semantics of this object is shown below. An object of type CompSwap can hold a value `myValue`. It supports a single operation `compSwapOp` operation that takes two arguments: `prevValue` and `newValue`. It replaces the value of the object only if the old value of the object matches the `preValue`.

```
public class CompSwap {
  int myValue = 0;

  public CompSwap(int initValue) {myValue = initValue;}
  public synchronized int compSwapOp(int prevValue, int newValue) {
      int oldValue = myValue;
      if (myValue == prevValue) myValue = newValue;
      return oldValue;
 }
}
```

Processes use a CompSwap object $x$ for consensus as follows. It is assumed to have the initial value of $-1$. Each process tries to update $x$ with its own `pid`. They use the inital value $-1$ as `prevValue`. It is clear that only the first process will have the right `prevValue`. All other processes will get the `pid` of the first process when they perform this operation. The program to implement consensus is given below.

```
public class CompSwapConsensus implements Consensus {
    CompSwap x= new CompSwap(-1);
    int proposed[];

    public CompSwapConsensus(int n){ proposed = new int[n];}
    public void propose(int pid, int v){ proposed[pid] = v; }
    public int decide(int pid){
       int j = x.compSwapOp(-1,pid);
       if (j == -1) return proposed[pid];
       else return proposed[j];
    }
}
```

## 9.2 Problems

9.1. Show that `TestAndSet` cannot be used to solve the consensus problem for three processes. (Hint: show that TestAndSet by two processes in any order result in the same state and the third process cannot distinguish between the two cases.)

9.2. Consider a concurrent FIFO queue class that allows two threads to concurrently dequeue. Show that the consensus number of such an object is 2. (Hint: Assume that queue is initialized with two values 0 and 1. Whichever process dequeues 0 wins.)

9.3. Consider a concurrent object of type `Swap` that holds an integer. It supports a single method `swapOp(int v)` that sets the value with $v$ and returns the old value. Show that `Swap` has consensus number 2.

# Chapter 10

# Universal Construction

## 10.1 Introduction

Now that we have seen universal objects, let us use them to build other concurrent objects. We first show a construction for concurrent queue that allows multiple processes to enque and deque concurrently.

Our construction will almost be mechanical. We first begin with a sequential implementation of a queue.

```
public class SeqQueue
{
        public class Element {
                public String data; public Element next;
                public Element(String s, Element e){data = s; next=e;}
        }
        public Element head, tail;
        public SeqQueue() { head = null; tail = null; }
        public SeqQueue(SeqQueue copy){
                Element node;
                head = copy.head;
                tail = copy.tail;
                for (Element index = head; index != null; index=index.next)
                        node = new Element(index.data, index.next);
        }

        public void Enqueue(String data) {
                Element temp = new Element(data,null);

                if (tail == null){ tail = temp; head = tail; }
                else { tail.next = temp; tail = temp; }
        }

        public String Dequeue() {
                if (head == null) return null;
                String returnval = head.data;
                head = head.next;
                return returnval;
        }
}
```

We now use the load-linked and store-conditional (LLSC) object to design a concurrent queue. An LLSC object provides two atomic operations. The load-linked operation allows a thread to load the value of a pointer to an object. The store-conditional operation allows the pointer to an object to be updated if the pointer has not changed since the last load-linked operation. Thus the semantics of LLSC are given below.

```
public class LLSC {
   ObjPointer p;
   public LLSC(Object x) {p.obj = x; p.version = 0;}

   public synchronized void load_linked(ObjPointer local) {
      local.obj = p.obj;
      local.version = p.version;
   }

   public synchronized boolean store_conditional(ObjPointer local, Object newObj) {
      if ((p.obj == local.obj) && (p.version == local.version)) {
           p.obj = newObj;
           p.version++;
           return true;
      }
      return false;
   }
}
```

Now we use LLSC to implement a concurrent queue using a pointer swinging technique. In the object pointer swinging technique any thread that wishes to perform an operation on an object goes through the following steps:

1. The thread makes a copy of that object.
2. It performs its operation on the copy of the object instead of the original object.
3. It swing the pointer of the object to the copy if the original pointer has not changed. This is done atomically.
4. If the original pointer has changed, then some other thread has succeeded in performing its operation. This thread starts all over again.

Using these ideas, we can now implement a concurrent queue class as follows.

```
public class CQueue{
        private LLSC x;

        public CQueue(){ x = new LLSC(new SeqQueue()); }

        public void Enqueue(String data) {
                SeqQueue new_queue;
                ObjPointer local = new ObjPointer();
                while (true) {
                        x.load_linked(local);
                        new_queue = new SeqQueue((SeqQueue) local.obj);
                        new_queue.Enqueue(data);
                        if (x.store_conditional(local, new_queue))
```

```
                                return;
                }
        }

        public String Dequeue() {
                SeqQueue new_queue;
                ObjPointer local = new ObjPointer();
                String returnval;
                while (true) {
                        x.load_linked(local);
                        new_queue = new SeqQueue((SeqQueue) local.obj);
                        returnval = new_queue.Dequeue();
                        if (x.store_conditional(local, new_queue))
                                return returnval;
                }
        }
}
```

While the above technique can be applied for lock-free construction of any concurrent object, it may be inefficient in practice because every operation requires a copy.

We now show that more efficient algorithms may exist for concurrent objects. The following algorithm for a concurrent queue is due to Michael and Scott. This algorithm uses pointer swinging technique. To distinguish between two pointers we use `count` asscoiated with the pointer. The `Pointer` class is shown below. It assumes two atmoic operations: `copyTo` and `CAS`. The first operation updates a pointer atomically. The second operation is our usual compare and swap.

```
   public class Element { public String data; public Pointer next; }

   public class Pointer {
     public Element ptr;
     public int count;
     public Pointer(Element initItem, int initCount){
         ptr = initItem; count = initCount;
     }
     public synchronized void copyTo(Pointer x){
         // assumes that x is accessed by at most one process
         x.ptr = ptr;
         x.count = count;
     }
   public synchronized boolean CAS(Pointer localPtr, Element newPtr, int newCount) {
     if ((ptr == localPtr.ptr) && (count == localPtr.count)) {
           ptr = newPtr; count = newCount;
           return true;
     } else return false;
   }
   }
```

Now, we show the algorithm for the concurrent queue.

```
public class ConcQueue {
   private Pointer head, tail;
   public ConcQueue(){ // set head and tail to a dummy node
     head = new Pointer(new Element(),0);
     head.ptr.next = new Pointer(null,0);
     tail = new Pointer(head.ptr,head.count);
   }

   public void Enqueue(String data) {
     Pointer local_tail = new Pointer(null,0);
     Pointer local_next = new Pointer(null,0);

     Element node = new Element();
     node.data = data; node.next = new Pointer(null,0);

     while (true) {
             tail.copyTo(local_tail); // make a local copy of tail
             local_tail.ptr.next.copyTo(local_next); // make a local copy of tail.next;
             if (local_next.ptr == null) {
                     // if successful swing next to new node, then enqueue is done
                     if (local_tail.ptr.next.CAS(local_next, node, local_next.count +1))
                             break;
             } else  // Need to swing tail to last node
                     tail.CAS(local_tail, local_next.ptr, local_tail.count+1);
     }
     // Try to swing tail to inserted node
     tail.CAS(local_tail, node, local_tail.count + 1);
   }

   public String Dequeue() {
     Pointer local_tail = new Pointer(null,0);
     Pointer local_head = new Pointer(null,0);
     Pointer local_next = new Pointer(null,0);

     while (true){
             tail.copyTo(local_tail); // make a local copy of tail
             head.copyTo(local_head); // make a local copy of head
             local_head.ptr.next.copyTo(local_next); // make a local copy of head.next;

             if (local_head.ptr == local_tail.ptr) { //queue empty or tail is behind
                 if (local_next.ptr == null) return null; // queue is empty
                 tail.CAS(local_tail, local_next.ptr, local_tail.count + 1); //advance tail
             } else { String return_val = local_next.ptr.data;
                   if (head.CAS(local_head, local_next.ptr, local_head.count+1))
                         return return_val; // dequeue is done
             }
     }
   }
}
```

# Chapter 11

# Transactions

## 11.1 Introduction

The concept of a transaction has been very useful in allowing concurrent processing of data with consistent results. A transaction is a sequence of operations such that that entire sequence appears as one indivisible operation. For any observer, it appears as if the entire sequence has been executed or none of the operations in the sequence have been executed. This property of indivisibility is preserved by a transaction despite the presence of *concurrency* and *failures*. By concurrency, we mean that multiple transactions may be going on at the same time. We are guaranteed that the transactions do not interefere with each other. The concurrent execution of multiple transactions is equivalent to a serial execution of those transactions. Further, if the transaction has not committed and there is failure then everything should be restored to appear as if none of the operations of the transaction happened. If the transcation has committed, then the results of the transaction must become permanent even when there are failures.

As an example of a transaction, consider transfer of money from account $A$ to account $B$. The transaction, say $T_1$ can be written as

```
transaction begin
        withdraw $x$ from Account $A$;
        deposit $x$ to Account $B$;
end
```

Once the two operations of withdrawing and depositing have been grouped as a transaction, they become atomic to the rest of the world. Assume for example that another transaction $T_2$ is executed concurrently with this transaction. $T_2$ simply adds the balances in accounts $A$ and $B$. The semantics of the transaction guarantees that $T_2$ will not add the balance of account $A$ after the withdrawal and the balance of account $B$ before the deposit. The all-or-nothing property is also guaranteed when there is a failure. If the second operation cannot be performed for some reason (such as account $B$ does not exist), then the effects of the first operation are also not visible, i.e., the balance is restored in account $A$.

A transaction can be viewed as implementing the following primitives.

1. **begin_transaction**: This denotes the beginning of a transaction.

2. **end_transaction**: This denotes the end of a transaction. All the statements between the beginning and the end of the transaction constitute the transaction. Execution of this primitive results in committing the transaction and its effects must persist.

3. **abort_transaction**: The user has the ability to call abort_transaction in the middle of a transaction. This requires that all values prior to the transaction are restored.

4. **read**: Within a transaction, the program can read objects.

5. **write**: Within a transaction, the program can also write objects.

## 11.2 ACID properties

Sometimes the guarantees provided by a transaction are called ACID properties where ACID stands for atomicity, consistency, isolation and durability. These terms are explained next.

- *Atomicity*: This property refers to all-or-nothing property explained earlier.
- *Consistency*: A transaction should not violate integrity constraints of the system. The typical example is that of a financial system where the transaction of money transfer from one account to the other should keep the total amount of money in the system constant. It is the responsibility of the programmer writing the transaction to ensure that such constraints are not violated after the transaction has taken place.
- *Isolation*: This means that transactions are isolated from effects of concurrent transactions. Thus, in spite of concurrency, the net effect is that it appears that all transactions excuted sequentially in some order.
- *Durability*: This property refers to the aspect of committing a transaction. It says that once a transaction has been committed its effects must become permanent even if there are failures.

## 11.3 Concurrency Control

The isolation property of transaction is also called ***serializability*** condition. A concurrent history $H$ of transactions $T_1, T_2, \ldots, T_n$ is ***serializable*** if it is equivalent to a serial history. As an example, suppose that there are two transactions $T_1$ and $T_2$. $T_1$ transfers hundred dollars from account $A$ to account $B$ and $T_2$ transfers two hundred dollars from account $B$ to account $C$. Assuming that all accounts have thousand dollars initially, the final balances for $A$, $B$ and $C$ should be 900, 900 and 1200 respectively. $T_1$ could be implemented as:

```
begin_trans;
  x = read(A);
  x = x-100;
  write x to A;
  x = read(B);
  x = x+100;
  write x to B;
end_trans;
```

$T_2$ could be implemented as:

```
begin_trans;
  y = read(B);
  y = y-200;
  write y to B;
  y = read(C);
  y = y+200;
  write y to C;
end_trans;
```

It is clear that all concurrent histories are not serializable. In the above example, assume that the following history happened.

```
  x = read(A);
  x = x-100;
  write x to A;
  x = read(B);
   y = read(B);
   y = y-200;
   write y to B;
   y = read(C);
   y = y+200;
   write y to C;
  x = x+100;
  write x to B;
```

In this case, the final values would be 900, 800 and 1200 which is clearly wrong. One way to ensure serializability would be by locking the entire database when a transaction is in progress. However, this will allow only serial histroies.

A more practical technique frequently employed is called **two phase locking**. In this technique, a transaction is viewed as consisting of two phases - locking phase and unlocking phase. In the locking phase (sometimes called a growing phase), a transaction can only lock data items and in the unlocking phase (sometimes called a shrinking phase) it can only unlock them. With this technique, the implementation of $T_1$ would be

```
begin_trans;
  lock(A);
  x = read(A);
  x = x-100;
  write x to A;
  lock(B);
  x = read(B);
  x = x+100;
  write x to B;
  unlock(A);
  unlock(B);
end_trans;
```

## 11.4 Dealing with Failures

There are primarily two techniques used for dealing with failures called **private workspace** and **logging**. In the private workspace approach, a transaction does not change the original primary copy. Any object that is affected by the transaction is kept in a separate copy called a shadow copy. If the transaction aborts, then private or shadow copies are discarded and nothing is lost. If the transaction commits, then all the shadow copies become the primary copies. Note that this technique is different from the technique for non-blocking synchronization that we studied earlier. In the private workspace approach, we do not make copy of the entire database before a transaction is started. Instead, a copy of only those objects (or pages) are made which have been updated by the transaction. This technique can be implemented as follows. Assume that objects can only be accessed through pointers in the index table. Let $S$ be primary index table of objects. At the beginning of a transaction, a copy of this table, $S'$, is made and all read and write operations go through $S'$. Since, reads do not change the object, both $S$ and $S'$ point to the same copy and thus all the read operations still go to the primary copy. For a write, a new copy of that object is made and the pointer in the table $S'$ is changed to the updated version. If the transaction aborts, then the table $S'$ can be discarded; otherwise, $S'$ becomes the primary table. Observe that this scheme requires locking of the objects for transactions to be serializable.

In the logging scheme, all the updates are performed on a single copy. However, a trail of all the writes are kept so that in case of a failure, one can go to the log and undo all the operations. For example, if an operation changed the value of object $x$ from 5 to 3, then in the log it is maintained that $x$ is changed from 5 to 3. If the transaction has to abort, then it is easy to undo this operation.

## 11.5 Problems

11.1. A contracted two-phase locking scheme is one in which the second phase is contracted, that is, all locks are unlocked at the same time. What are the advantages and disadvantages of this scheme compared with ordinary two-phase locking scheme ?

11.2. Write a class that provides the following services:
*lock*(String varname; int pid); returns 1 if allowed by two phase locking scheme, returns 0 otherwise
*unlock*(String varname, int pid);returns 1 if locked by the process, returns 0 otherwise
Assume that the processor never crashes.

11.3. Which of the following schedules are serializable ?
(i) $r_1(a,b); w_1(b); r_2(a); w_1(a); w_2(a)$.
(ii) $r_1(a,b); w_1(a); r_2(a); w_2(a); r_1(b)$.
(iii) $r_2(a); r_1(a,b); w_2(c); w_1(a); w_1(c)$.
(iv) $r_1(a), r_2(b), w_1(a), w_2(b), r_1(b), w_1(b), r_2(c), w_2(c)$

For each of the serializable schedules, show a possible two phase locking history.

11.4. Assume that you have two floats representing checking balance and savings balance stored on disk. Write a program that transfers hundred dollars from the checking account to the savings account. Assume that the processor can crash at anytime but disks are error-free. This means that you will also have to write a crash recovery procedure. You are given the following primitives:

```
class Stable {
 float val;
  synchronized void get(float x) // copies the value of disk object val to memory object x
  synchronized void set(float x) // copies the value of memory object x to disk object val
}
```

# Chapter 12

# Distributed Programming

## 12.1 Introduction

In this chapter, we will learn primitives provided in the Java programming language for building distributed applications. We will primarily see two programming styles - sockets and remote method invocations. Sockets provide a lower-level interface for building distributed programs but they are more efficient and flexible.

In this chapter we first describe the class InetAddress which is useful for network programming no matter which style of primitives are used. Then we discuss primitives for programming using sockets. We give an example of a simple name server that is implemented using sockets. Finally, we discuss programming using remote method invocations.

## 12.2 InetAddress Class

For any kind of distributed application, we need the notion of an internet address. Any computer connected on the Internet (called *host*) is identified by a unique address called IP address. Since addresses are difficult to remember, each host also has a hostname. It is the task of a Domain Name System (DNS) Server to provide the mapping from a hostname to its address. Java provides a class *Java.net.Inetaddress* which can be used for this translation. The relevant methods for the class Inetaddress are given below:

```
public byte[] getAddress()
  Returns the raw IP address of this InetAddress object.
public static InetAddress getByName(String)
  Determines the IP address of a host, given the host's name.
public String getHostAddress()
  Returns the IP address string "%d.%d.%d.%d"
public String getHostName()
  Returns the fully qualified host name for this address.
public static InetAddress getLocalHost()
  Returns the local host.
```

## 12.3 Sockets

Sockets are useful in writing programs based on communication using messages. A socket is an object that can be used to send and receive messages. There are primarily two protocols used for sending and receiving messages

- Universal Datagram Protocol (UDP) and Transmission Control Protocol (TCP). The UDP provides a low-level connection less protocol. This means that packets sent using UDP are not guaranteed to be received in the order sent. In fact, the UDP protocol does not even guarantee reliability, i.e. a packet may get lost. The protocol does not use any handshaking mechanisms (such as acknowledgments) to detect loss of packets. Why is UDP useful then ? Because, even though UDP may lose packets, in practice, this is rarely the case. Since there are no overheads associated with error checking, UDP is an extremely efficient protocol.

The TCP protocol is a reliable connection oriented protocol. It also guarantees ordered delivery of packets. Needless to say, TCP is not as efficient as UDP.

## 12.4 Datagram Sockets

The first class that will be of use to us is DatagramSocket. This class represents a socket for sending and receiving datagram packets. A datagram socket is the sending or receiving point for a connectionless packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from a machine to another may be routed differently, and may arrive in any order. This class provides a very low level interface for sending and receiving messages. There are few guarantees associated with datagram sockets. An advantage of datagram sockets is that it allows fast data transmission.

The details for the methods in this class are given below. To construct a DatagramSocket, we can use one of the following constructors:

```
public DatagramSocket() throws SocketException
public DatagramSocket(int port) throws SocketException
public DatagramSocket(int port, InetAddress laddr) throws SocketException
```

The first constructor constructs a datagram socket and binds it to any available port on the local host machine. Optionally, a port may be specified as in the second constructor. The last constructor creates a datagram socket, bound to the specified local address.

These constructors throw SocketException if the socket could not be opened, or the socket could not bind the specified local port. The method

```
public void close()
```

closes a datagram socket. To get the information about the socket, one can use

```
public int getLocalPort()
```

which returns the port number on the local host to which this socket is bound. The method

```
public InetAddress getLocalAddress()
```

gets the local address to which the socket is bound.

Receiving a packet is accomplished by

```
public void receive(DatagramPacket p) throws IOException
```

The method `receive` receives a datagram packet from this socket. When this method returns, the DatagramPacket's buffer is filled with the data received. The datagram packet also contains the sender's IP address, and the port number on the sender's machine.

This method blocks until a datagram is received. The length field of the datagram packet object contains the length of the received message. If the message is longer than the buffer length, the message is truncated. It throws IOException if an I/O error occurs. The blocking can be avoided by setting the timeout.

Sending a packet is accomplished by

```
public void send(DatagramPacket p) throws IOException
```

It sends a datagram packet from this socket. The DatagramPacket includes information indicating the data to be sent, its length, the IP address of the remote host, and the port number on the remote host.

## 12.5 DatagramPacket Class

The above class required data to be sent as datagram packets. The class java.net.DatagramPacket is used for that. Its definition is given below.

```
public  final  class  java.net.DatagramPacket
    extends  java.lang.Object
{
        // Constructors
    public DatagramPacket(byte  ibuf[], int  ilength);
    public DatagramPacket(byte  ibuf[], int  ilength,
                        InetAddress  iaddr, int iport);

        // Methods
    public InetAddress getAddress();
    public byte[] getData();
    public int getLength();
    public int getPort();
    public void setAddress(InetAddress)
    public void setData(byte[])
    public void setLength(int)
    public void setPort(int)
}
```

The first constructor

public DatagramPacket(byte ibuf[], int ilength)

constructs a DatagramPacket for receiving packets of length ilength. The parameter ibuf is the buffer for holding the incoming datagram, and ilength is the number of bytes to read.

The constructor for creating a packet to be sent is

public DatagramPacket(byte ibuf[], int ilength, InetAddress iaddr, int iport)

It constructs a DatagramPacket for sending packets of length ilength to the specified port number on the specified host. The parameters iaddr and iport are used for the desination address and the destination port number respectively. The method `getAddress` returns the IP address of the machine to which this datagram is being sent, or from which the datagram was received. The method `getData` returns the data received, or the data to be sent. The method `getLength` returns the length of the data to be sent, or the length of the data received. Similarly, the method `getPort` returns the port number on the remote host to which this datagram is being sent, or from which the datagram was received. The `set` methods are used to set the IP address, port number etc. appropriately.

### 12.5.1 Example of Using Datagrams

We give a simple example of a program that uses Datagrams. This example consists of two processes - a server and a client. The client reads input from the user and sends it to the server. The server receives the datagram packet and then echoes back the same data. The program for the server is given below.

```
import java.net.*;
import java.io.*;
public class DatagramServer {
  public static void main(String[] args) {
    DatagramPacket datapacket, returnpacket;
    int port = 2018;
    int len = 1024;
      try {
      DatagramSocket datasocket = new DatagramSocket(port);
      byte[] buffer = new byte[len];
      datapacket = new DatagramPacket(buffer, buffer.length);
      while (true) {
        try {
          datasocket.receive(datapacket);
          returnpacket = new DatagramPacket(datapacket.getData(), datapacket.getLength(),
                                  datapacket.getAddress(), datapacket.getPort());
          datasocket.send(returnpacket);
        }
        catch (IOException e) { System.err.println(e); }
      }
    }
    catch (SocketException se) { System.err.println(se); }
  }
}
```

The client process reads a line of input from `System.in`. It then creates a datagram packet and sends it to the server. On receiving a response from the server it displays the message received. The program for the client is given below.

```
import java.net.*;
import java.io.*;
public class DatagramClient {
  public static void main(String[] args) {
    String hostname;
    int port = 2018;
    int len = 1024;
    DatagramPacket sPacket, rPacket;

    if (args.length > 0) hostname = args[0];
    else hostname = "localhost";
      try {
      InetAddress ia = InetAddress.getByName(hostname);
      DatagramSocket datasocket = new DatagramSocket();
      DataInputStream stdinp = new DataInputStream(System.in);

      while (true) {
        try {
          String echoline = stdinp.readLine();
          if (echoline.equals("done")) break;
          byte [] buffer = new byte[echoline.length()];
          echoline.getBytes(0, echoline.length(), buffer, 0);
          sPacket = new DatagramPacket(buffer, buffer.length, ia, port);
          datasocket.send(sPacket);
          byte[] rbuffer = new byte[len];
          rPacket = new DatagramPacket(rbuffer, rbuffer.length);
          datasocket.receive(rPacket);
          String retstring = new String(rPacket.getData(), 0, 0,
rPacket.getLength());
          System.out.println(retstring);
        }
catch (IOException e) { System.err.println(e); }
      } // while
    }
    catch (UnknownHostException e) { System.err.println(e); }
    catch (SocketException se) { System.err.println(se); }
  }  // end main

}
```

## 12.6 Sockets based on TCP

The second style of inter-process communication that we discuss is based on the notion of streams. In this style, a connection is set up between the sender and the receiver. This style allows better error recovery and guarantees on the delivery of packets. Thus, in a stream the data is received in the order it is sent.

The socket class in Java extends the Object class. We will give only a subset of constructors and methods available for Socket.

public Socket(String host, int port) throws UnknownHostException, IOException

This constructor creates a stream socket and connects it to the specified port number on the named host. Here we have used the name of the host. Alternatively, IP address can be used in the form of the class InetAddress as below: public Socket(InetAddress address, int port) throws IOException

The methods for the socket are

- public InetAddress getInetAddress()

  returns the remote IP address to which this socket is connected.

- public InetAddress getLocalAddress()

  Gets the local address to which the socket is bound.

- public int getPort()

  Returns the remote port to which this socket is connected.

- public InputStream getInputStream() throws IOException

  returns an input stream for reading bytes from this socket.

- public OutputStream getOutputStream() throws IOException

  returns an output stream for writing bytes to this socket.

- public synchronized void close() throws IOException
  Figure **??** illustrates these terms.

  Closes this socket.

Note that many of the methods above throw IOException if an I/O error occurs when applying the method to the socket.

### 12.6.1 Server Sockets

On the server side the class that is used is called ServerSocket. A way to create a server socket is public ServerSocket(int port) throws IOException

This call creates a server socket on a specified port. Various methods on a server socket are:
public InetAddress getInetAddress()
It returns the address to which this socket is connected, or null if the socket is not yet connected.
public int getLocalPort()
returns the port on which this socket is listening.
public Socket accept() throws IOException
Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.
public void close() throws IOException
Closes this socket.

#### Example

We now give a simple name server implemented using server sockets. The name server creates a server socket with the specified port. It then listens to any incoming connections by the method `accept`. The `accept` method returns the socket whenever a connection is made. It then handles the request which arrives on that socket by the method `handleclient`. We call `getInputStream` and `getOutputStream` to get input and output streams associated with the socket. Now we can simply use all methods associated for reading and writing input streams to read and write data from the socket.

In our implementation of the name server, at most one client is handled at a time. Once a request is handled the main loop of the name server accepts another connection. For many applications this may be unacceptable if the procedure to handle a request takes a long time. For these applications, it is quite common for the server to be multi-threaded. The server accepts a connection and then spawns a thread to handle the request. However, it must be observed that since the data for the server is shared among multiple threads, it is the responsibility of the programmer to ensure that the data is accessed in a safe manner (for example, by using `synchronized` methods).

```
import java.net.*;
import java.io.*;
import java.util.*;
public class NameServer {
 static final int debug = 1;
  final int numProc = 100;
  private String [] names = new String [numProc];
  private String [] hosts = new String [numProc];
  private int [] ports = new int [numProc];
  private int dirsize = 0;
  private String hname = new String();
    private String nullhost = new String("nullhost");

  int search(String s) {
    for(int i=0; i < dirsize; i++)
      if (ports[i] != -1)
        if (names[i].equals(s)) {
          hname = hosts[i];
          return ports[i];
        }
    hname = nullhost;
    return -1;
  }

  void handleclient(Socket theClient) {
    try {
      BufferedReader din = new BufferedReader(new InputStreamReader(theClient.getInputStream()));
      PrintWriter pout = new PrintWriter(theClient.getOutputStream());
      String getline = din.readLine();
      if (debug==1) System.out.println(getline);
      StringTokenizer st = new StringTokenizer(getline);
      String command = st.nextToken();
      String name = st.nextToken();
      int portnum = search(name);
      if (command.equals("search"))  pout.println(portnum + " " + hname);
      else if (command.equals("insert")) {
        if (portnum == -1) {
          names[dirsize] = new String(name);
          hosts[dirsize] = new String(st.nextToken());
          ports[dirsize] = Integer.parseInt(st.nextToken());
          dirsize++;
          pout.println(1);
} else pout.println(0);
      }
      pout.flush();
    }
    catch (IOException e) {System.err.println(e);}
  }

  public static void main(String [] args) {
    NameServer ns = new NameServer();
    try {
      ServerSocket listener = new ServerSocket(Symbols.ServerPort);
      while (true) {
        Socket aClient = listener.accept();
        ns.handleclient(aClient);
        aClient.close();
      }
    }
    catch(IOException e) { System.err.println("Server aborted:" + e);}
  }
}
```

The following client program can be used to test the above server.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;
 public class Name {
   BufferedReader din;
   PrintStream pout;
```

```
  public void getSocket() throws IOException {
        Socket server = new Socket(Symbols.nameServer, Symbols.ServerPort);
      din = new BufferedReader(new InputStreamReader(server.getInputStream()));
      pout = new PrintStream(server.getOutputStream());
  }

  public int insertName(String name, String hname, int portnum) throws IOException {
        getSocket();
        pout.println("insert " + name + " " + hname + " " + portnum); pout.flush();
        return Integer.parseInt(din.readLine());
    }

    public PortAddr searchName(String name) throws IOException {
        getSocket();
        pout.println("search " + name); pout.flush();
        String result = din.readLine();
        StringTokenizer st = new StringTokenizer(result);
        int portnum = Integer.parseInt(st.nextToken());
        String hname = st.nextToken();
       return new PortAddr(hname, portnum);
    }


    public int deleteName(String name, String hname, int portnum) throws IOException {
        getSocket();
        pout.println("delete " + name); pout.flush();
        return Integer.parseInt(din.readLine());
    }

    public static void main(String [] args) {
       Name myClient = new Name();
        try {
            myClient.insertName("hello1", "birch.ece.utexas.edu", 1000);
            PortAddr pa =  myClient.searchName("hello1");
            System.out.println(pa.gethostname() + ":" + pa.getportnum());
        }
        catch(Exception e) { System.err.println("Server aborted:" + e); }
  }
}
```

### 12.6.2 Remote Method Invocations

Another popular way of developing distributed applications is based on the concept of remote procedure calls (RPC's) or remote method invocations (rmi's). Here the main idea is that a process can make calls to methods of a remote object as if it were on the same machine.

In Java, the package java.rmi includes an interface `remote` and two classes. The Remote interface serves to identify all remote objects. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a remote interface are available remotely.

public final class Naming extends Object

This is the bootstrap mechanism for obtaining references to remote objects based on Uniform Resource Locator

(URL) syntax. The URL for a remote object is specified using the usual host, port and name: rmi://host:port/name host = host name of registry (defaults to current host) port = port number of registry (defaults to the registry port number) name = name for remote object

The key methods are

```
  bind(String, Remote)
    Binds the name to the specified remote object.
  list(String)
    Returns an array of strings of the URLs in the registry.
  lookup(String)
    Returns the remote object for the URL.
  rebind(String, Remote)
    Rebind the name to a new object; replaces any existing binding.
  unbind(String)
    Unbind the name.
```

public final class LocateRegistry extends Object

This class is used to obtain the bootstrap Registry on a particular host (including your local host). The following example demonstrates usage (minus exception handling): Server wishes to make itself available to others:

```
SomeService service = ...; // remote object for service
Registry registry = LocateRegistry.getRegistry();
registry.bind("I Serve", service);
```

The client wishes to make requests of the above service:

```
Registry registry = LocateRegistry.getRegistry("foo.services.com");
SomeService service = (SomeService)registry.lookup("I Serve");
service.requestService(...);
```

```
 createRegistry(int)
    Create and export a registry on the local host.
  getRegistry()
    Returns the remote object Registry for the local host.
  getRegistry(int)
    Returns the remote object Registry on the current host at the specified port.
  getRegistry(String)
    Returns the remote object Registry on the specified host at a default
(i.e., well-known) port number.
  getRegistry(String, int)
    Returns the remote object Registry on the specified host at the specified port.
```

### 12.6.3 Passing Arguments

There are three ways of passing arguments (and returning results) in rmi programs. The primitive types in Java (e.g. int and boolean) are passed by values. References to objects that implement `remote` interface are passed as remote references. Suce objects, called *Remote objects*, run on the server. Finally, objects that are not remote are passed by value using object serialization. Object serialization refers to the process of converting the object state into a stream of bytes.

## 12.7 Other Useful Classes

In this chapter, we have focused on classes that allow you to write distributed programs. For cases, when a process simply needs data from a remote location, Java provides the Uniform Resource Locator (URL) class. A URL consists of six parts: protocol, hostname, port, path, filename and document section. An example of a URL is

```
http://www.ece.utexas.edu:80/classes.html#distributed
```

The java.net.URL class allows the programmer to read data from a URL by methods such as

```
public final InputStream openStream() throws java.io.IOException
```

This method returns a `InputStream` from which one can read the data. For different types of data such as images and audio clips there are methods such as
`public Image getImage(URL u, String filename)`
and
`public void play(URL u).`
We will not concern ourselves with these classes and methods.

Chapter 13

# Clocks

## 13.1 Introduction

We have defined two relations between events based on the global total order of events, and the happened before order. In this chapter we discuss mechanisms called *clocks* that can be used for tracking these relations.

The first relation we discussed on events imposes a total order on all events. Because this total order cannot be observed we give a mechanism to generate a total order that could have happened in the system (rather than that actually happened in the system). This mechanism is called a *logical clock.* The second relation, happened before, can be accurately tracked by a *vector clock.* A vector clock assigns timestamps to states (and events) such that the happened before relationship between states can be determined by using the timestamps.

This chapter is organized as follows. Section 13.2 describes logical clocks, and Section 13.3 describes vector clocks.

## 13.2 Logical Clocks

When the behavior of a distributed computation is viewed as a total order, it is impossible to determine the actual order of events in the absence of accurately synchronized physical clocks. If the system has a shared clock (or equivalently, precisely synchronized clocks), then timestamping the event with the clock would be sufficient to determine the order. Because in the absence of a shared clock the total order between events cannot be determined, we will develop a mechanism that gives a total order that *could have* happened instead of the total order that did happen.

The purpose of our clock is only to give us an order between events and not any other property associated with clocks. For example, based on our clocks one could not determine the time elapsed between two events. In fact, the number we associate with each event will have no relationship with the time we have on our watches.

As we have seen before, there are only two kinds of order information that can be determined in a distributed system—the order of events on a single process and the order between the send and the receive event of a message. On the basis of these considerations, we get the following definition.

A *logical clock* $C$ is a map from the set of events $E$ to $\mathcal{N}$ (the set of natural numbers) with the following constraint:

$$\forall s, t \in E : s \to t \Rightarrow C(s) < C(t)$$

The constraint for logical clocks models the sequential nature of execution at each process and the physical requirement that any message transmission requires a nonzero amount of time.

Availability of a logical clock during distributed computation makes it easier to solve many distributed problems. An accurate physical clock clearly satisfies the above condition and therefore is also a logical clock. However, by definition of a distributed system there is no shared clock in the system. Figure 13.1 shows an implementation of a logical clock that does not use any shared physical clock or shared memory.

It is not required that message communication be ordered or reliable. The algorithm is described by the initial conditions and the actions taken for each event type. The algorithm uses the variable $c$ to assign the logical clock. The notation $s.c$ denotes the value of $c$ in the state $s$. Let $s.p$ denote the process to which state $s$ belongs.

For any send event, the value of the clock is sent with the message and then incremented. On receiving a message, a process takes the maximum of its own clock value and the value received with the message. After taking the maximum, the process increments the clock value. On an internal event, a process simply increments its clock.

```
class LamportClock {
  int c;

  public LamportClock() { c = 1; }
  public int getValue() { return c; }

  public void sendAction() {
     // include c in the message
     c = c + 1;
  }

  public void tick() { // on internal actions
     c = c + 1;
  }

  public void receiveAction(int src, int tag) {
    c = Util.max(c, tag) + 1;
  }
}
```

Figure 13.1: A logical clock algorithm

The following claim is easy to verify:

$$\forall s, t \in S : s \to t \Rightarrow s.c < t.c$$

In some applications it is required that all events in the system be ordered totally. If we extend the logical clock with process number, then we get a total ordering on events. Recall that for any state $s$, $s.p$ indicates the identity of the process to which it belongs. Thus the timestamp of any event is a tuple $(s.c, s.p)$ and the total order $<$ is obtained as:

$$(s.c, s.p) < (t.c, t.p) \overset{\text{def}}{=} (s.c < t.c) \lor ((s.c = t.c) \ \land \ (s.p < t.p)).$$

## 13.3 Vector Clocks

### 13.3.1 Definition

We saw that logical clocks satisfy the following property:

$$s \to t \Rightarrow s.c < t.c.$$

However, the converse is not true; $s.c < t.c$ does not imply that $s \to t$. The computation $(S, \to)$ is a partial order, but the domain of logical clock values (the set of natural numbers) is a total order with respect to $<$. Thus logical clocks do not provide complete information about the *happened before* relation. In this section, we describe a mechanism called a vector clock.

**Definition 13.1 (Vector Clock)** *A* vector clock *$v$ is a map from $S$ to $\mathcal{N}^k$ (vectors of natural numbers) with the following constraint:*

$$\forall s,t : s \rightarrow t \Leftrightarrow s.v < t.v.$$

*where $s.v$ is the vector assigned to the state $s$.*

Because $\rightarrow$ is a partial order, it is clear that the timestamping mechanism should also result in a partial order. Thus the range of the timestamping function cannot be a total order like the set of natural numbers used for logical clocks. Instead, we use vectors of natural numbers. Given two vectors $x$ and $y$ of dimension $N$, we compare them as follows.

$$\begin{aligned} x < y &= (\forall k : 1 \leq k \leq N : x[k] \leq y[k]) \ \wedge \\ &\quad (\exists j : 1 \leq j \leq N : x[j] < y[j]) \\ x \leq y &= (x < y) \vee (x = y) \end{aligned}$$

It is clear that this order is only partial for $N \geq 2$. A vector clock timestamps each event with a vector of natural numbers.

### 13.3.2 Implementation

Our implementation of vector clocks uses vectors of size $N$, the number of processes in the system. The algorithm presented in Figure 13.2 is described by the initial conditions and the actions taken for each event type. A process increments its own component of the vector clock after each event. Furthermore, it includes a copy of its vector clock in every outgoing message. On receiving a message, it updates its vector clock by taking a componentwise maximum with the vector clock included in the message. It is not required that message communication be ordered or reliable. A sample execution of the algorithm is given in Figure 13.3.

We now show that $s \rightarrow t$ iff $s.v < t.v$. We first claim that if $s \neq t$, then

$$s \not\rightarrow t \Rightarrow t.v[s.p] < s.v[s.p] \tag{13.1}$$

If $t.p = s.p$, then it follows that $t$ occurs before $s$. Because, the local component of the vector clock is increased after each event, $t.v[s.p] < s.v[s.p]$. So, we assume that $s.p \neq t.p$. Since $s.v[s.p]$ is the local clock of $P_{s.p}$ and $P_{t.p}$ could not have seen this value as $s \not\rightarrow t$, it follows that $t.v[s.p] < s.v[s.p]$. Therefore, we have that $(s \not\rightarrow t)$ implies $\neg(s.v < t.v)$.

Now we show that $(s \rightarrow t)$ implies $(s.v < t.v)$. If $s \rightarrow t$, then there is a message path from $s$ to $t$. Since every process updates its vector on receipt of a message and this update is done by taking the componentwise maximum, we know the following holds:

$$\forall k : s.v[k] \leq t.v[k].$$

Furthermore, since $t \not\rightarrow s$, from Equation 13.1, we know that $s.v[t.p]$ is strictly less than $t.v[t.p]$. Hence, $(s \rightarrow t) \Rightarrow (s.v < t.v)$.

It is left as an exercise to show that if we know the processes the vectors came from, the comparison between two states can be made in constant time.

$$s \rightarrow t \Leftrightarrow (s.v[s.p] \leq t.v[s.p]) \wedge (s.v[t.p] < t.v[t.p])$$

## 13.4 Direct-Dependency Clocks

One drawback with the vector clock algorithm is that it requires $O(N)$ integers to be sent with every message. For many applications, a weaker version of the clock suffices. We now describe a clock algorithm that is used by many algorithms in distributed systems. These clocks require only one integer to be appended to each message. We call these clocks direct-dependency clocks.

```
class VectorClock {
  int [] v;
  int myId;
  int N;

  public VectorClock(int numProc, int id) {
    myId = id;
    N = numProc;
    v = new int[numProc];
    for (int i=0; i < N; i++)
      v[i] = 0;

    v[myId] = 1;
   }

  public void sendAction(int[] tag) {
     tag = v;
     v[myId]++;
  }

  public void receiveAction(int[] tag) {
    for (int i=0; i < N; i++)
       v[i] = Util.max(v[i], tag[i]);
    v[myId]++;

  }

}
```

Figure 13.2: A vector clock algorithm

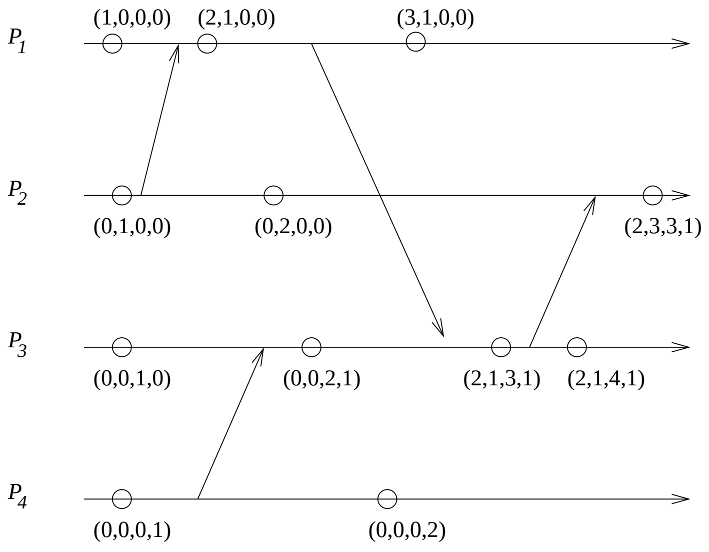

Figure 13.3: A sample execution of the vector clock algorithm

### 13.4.1 Algorithm

The algorithm shown in Figure 13.4 is described by the initial conditions and the actions taken for each event type. On a send event, the process sends only its local component in the message. It also increments its component as in vector clocks. The action for internal events is the same as that for vector clocks. When a process receives a message, it updates two components—one for itself, and the other for the process from which it received the message. It updates its own component in a manner identical to logical clocks. It also updates the component for the sender by taking the maximum with the previous value.

```
class DirectClock {
  public int [] clock;
  int myId;

  public DirectClock(int numProc, int id) {
    myId = id;
    clock = new int[numProc];
    for (int i=0; i < numProc; i++)
      clock[i] = 0;

    clock[myId] = 1;
   }
  public int getValue(int i) {return clock[i];}
  public void tick() { clock[myId]++; }
  public void receiveAction(int sender, int tag) {
    clock[sender] = Util.max(clock[sender], tag);
    clock[myId] = Util.max(clock[myId], tag) + 1;
  }

}
```

Figure 13.4: A direct-dependency clock algorithm

An example of a distributed computation and its associated direct-dependency clock is given in Figure 13.5.

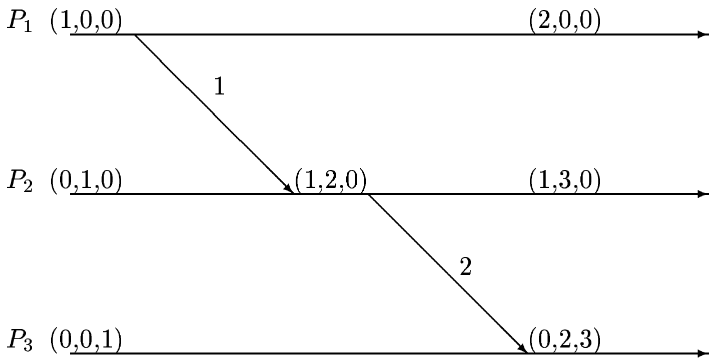

Figure 13.5: A sample execution of the direct-dependency clock algorithm.

### 13.4.2 Properties

We first observe that if we retain only the $i^{th}$ component for the $i^{th}$ process, then the above algorithm is identical to the logical clock algorithm. However, our interest in a direct-dependency clock is not due to its logical clock property (Lamport's logical clock is sufficient for that), but to its ability to capture the notion of direct-dependency. We first define a relation *directly precedes* $\rightarrow_d$, a subset of $\rightarrow$. as follows: $s \rightarrow_d t$ iff there is a path from $s$ to $t$ that uses at most one message in the happened before diagram of the computation. The following property makes direct-dependency clocks useful for many applications.

$$\forall s, t : s.p \neq t.p : (s \rightarrow_d t) \Leftrightarrow (s.v[s.p] \leq t.v[s.p])$$

## 13.5 Problems

13.1. We discussed a method by which we can totally order all events within a system. If two events have the same logical time, we broke the tie using process identifiers. This scheme always favors processes with smaller identifiers. Suggest a scheme that does not have this disadvantage. (Hint: Use the value of the logical clock in determining the priority.)

13.2. Prove the following for vector clocks: $s \rightarrow t$ iff

$$(s.v[s.p] \leq t.v[s.p]) \wedge (s.v[t.p] < t.v[t.p]).$$

13.3. Suppose that the underlying communication system guarantees FIFO ordering of messages. How will you exploit this feature to reduce the communication complexity of the vector clock algorithm? Give an expression for overhead savings if your scheme is used instead of the traditional vector clock algorithm. Assume that any process can send at most $m$ messages.

13.4. Assume that you have implemented the vector clock algorithm. However, some application needs Lamport's logical clock. Write a function *convert* that takes as input a vector timestamp and outputs a logical clock timestamp.

13.5. Give a distributed algorithm to maintain clocks for a distributed program that has a dynamic number of processes. Assume that there are the following events in the life of any process: start-process, internal, send, receive, fork, join processid, terminate. It should be possible to infer the *happened before relation* using your clocks.

## 13.6 Bibliographic Remarks

The idea of logical clocks is from Lamport [Lam78]. The idea of vector clocks in its pure form first appeared in Fidge [Fid89] and Mattern [Mat89]. However, vectors had been used before in some earlier papers such as Strom and Yemini [SY85].

# Chapter 14

# Resource Allocation

## 14.1 Introduction

In a distributed system mutual exclusion is often necessary for accessing shared resources such as data. For example, consider a table that is replicated on multiple sites. Assume that operations on the table can be issued concurrently. For their correctness we require that all operations appear *atomic* in the sense that the effect of the operations must appear indivisible to the user. For example, if an update operation requires changes to two fields $x$ and $y$, then another operation should not read the old value of $x$ and the new value of $y$. Observe that in a distributed system, there is no shared memory and therefore one could not use shared objects such as semaphores to implement the mutual exclusion.

Mutual exclusion is one of the most studied topics in distributed systems. It reveals many important issues in distributed algorithms such as safety and liveness properties. We will study three classes of algorithms—timestamp-based algorithms, token-based algorithms and quorum-based algorithms. The timestamp-based algorithms resolve conflict in use of resources based on timestamps assigned to requests of resources. The token-based algorithms use auxiliary resources such as tokens to resolve the conflicts. The quorum-based algorithms use a subset of processes to get permission for accessing the shared resource. All algorithms in this chapter assume that there are no faults in the distributed system, that is, processors and communication links are reliable.

The chapter is organized as follows. Section 14.2 describes the mutual exclusion problem. Section 14.3 presents Lamport's algorithm and its proof of correctness. Section 14.4 presents a modification of Lamport's algorithm by Ricart and Agrawala.

## 14.2 Specification of the Mutual Exclusion Problem

Let a system consist of a fixed number of processes and a shared resource called the *critical section.* An example of a critical section is the operation performed on the replicated table introduced earlier. The algorithm to coordinate access to the critical section must satisfy the following properties:

**Safety:** Two processes should not have permission to use the critical section simultaneously.

**Liveness:** Every request for the critical section is eventually granted.

**Fairness:** Different requests must be granted in the order they are made.

There are many algorithms for mutual exclusion in a distributed system. However, the least expensive algorithm for the mutual exclusion is a centralized algorithm. If we are required to satisfy just the safety and liveness properties, then a simple queue-based algorithm works. One of the processes is designated as the coordinator for the critical section. Any process that wants to enter the critical section sends a request to the coordinator. The coordinator simply puts these requests in a queue in the order it receives them. It also grants permission to the process that is at the head of the queue.

The above algorithm does not satisfy the notion of fairness, which says that requests should be granted in the order they are made and not in the order they are received. Assume that the process $P_i$ makes a request for the shared resource to the coordinator process $P_k$. After making the request, $P_i$ sends a message to the process $P_j$. Now, $P_j$ sends a request to $P_k$ that reaches $P_k$ earlier than the request made by the process $P_i$. This example shows that it is possible for the order in which requests are received by the coordinator process to be different from the order in which they are made.

## 14.3 Lamport's Algorithm

In Lamport's algorithm each process maintains a logical clock (used for timestamps) and a queue (used for storing requests for the critical section). The algorithm ensures that processes enter the critical section in the order of timestamp of their requests. The rules of the algorithm are as follows:

- To request the critical section, a process sends a timestamped message to all other processes and adds a timestamped request to the queue.
- On receiving a request message, the request and its timestamp are stored in the queue and an acknowledgment is sent back.
- To release the critical section, a process sends a release message to all other processes.
- On receiving a release message, the corresponding request is deleted from the queue.
- A process determines that it can access the critical section if and only if:
  1. It has a request in the queue with timestamp $t$, and
  2. $t$ is less than all other requests in the queue, and
  3. It has received a message from every other process with timestamp greater than $t$ (the request acknowledgments ensure this).

```
class LampMutex implements MsgHandler, Lock {
  int numProc, myId;
  Linker comm;
  DirectClock v;
  int [] requestq;

  LampMutex(int id, int N, Linker initComm) {
    myId = id; numProc = N; comm = initComm;
    v = new DirectClock(numProc, id);
    requestq = new int [numProc];
    for (int i=0; i < numProc; i++) requestq[i] = Symbols.Infinity;
  }

  public void requestCS() {
      v.tick();
      requestq[myId] = v.getValue(myId);
      for (int i=0; i < numProc; i++)
        if (i != myId) comm.sendMessage(i, "Request", requestq[myId]);
  }
```

```
 public void releaseCS() {
      requestq[myId] = Symbols.Infinity;
      for (int i=0; i<numProc; i++)
        if (i != myId) comm.sendMessage(i, "Release", v.getValue(myId));
  }

  public synchronized boolean okayCS() {
    if (requestq[myId] == Symbols.Infinity) return false;

    for(int i=0; i < numProc; i++)
      if (i != myId) {
        if (requestq[i] != Symbols.Infinity)
          if ((requestq[myId] > requestq[i]) ||
              ((requestq[myId] == requestq[i]) && (myId > i)))
            return false;

        if ((requestq[myId] > v.getValue(i)) ||
            ((requestq[myId] == v.getValue(i)) && (myId > i)))
          return false;
      }
    return true;
  }

  public void handleMessage(Message m){
  int source = m.getSrcId();
          String command  = m.getCommand();
  int timeStamp = Integer.parseInt(m.getMessage());
  v.receiveAction(source,timeStamp);
  if (command.equals("Request")){
       requestq[source] = timeStamp;
       comm.sendMessage(source, "Reply", v.getValue(myId));
          } else if (command.equals("Release")) requestq[source] = Symbols.Infinity;

  }

    public static void main(String [] args) throws Exception {
int myId = Integer.parseInt(args[1]);
int numProc = Integer.parseInt(args[2]);
Linker comm = new Linker(args[0], myId, numProc);
LampMutex lock1 = new LampMutex(myId, numProc, comm);
for (int i=0;i<numProc; i++)
   if (i != myId) (new ListenerThread(i,comm,lock1)).start();
while (true) {
System.out.println(myId + " is not in CS");
lock1.requestCS();
while (!lock1.okayCS()) Util.mySleep(500);
System.out.println(myId + " is in CS *****");
lock1.releaseCS();
}
   }
```

```
}
```

Lamport's algorithm requires $3(N-1)$ messages per invocation of the critical section: $N-1$ request messages, $N-1$ acknowledgement messages and $N-1$ release messages. There is a time delay of two serial messages to get permission for the critical section—a request message followed by an acknowledgment. The space overhead per process is the vectors $q$ and $v$ which is $O(N \log m)$ where $m$ is the maximum number of times any process enters the critical section.

## 14.4 Ricart and Agrawala's Algorithm

Ricart and Agrawala's algorithm uses only $2(N-1)$ messages per invocation of the critical section. It does so by combining the functionality of ack and release messages. In this algorithm, a process does not always send back an acknowledgement on receiving a request. It may defer the reply for a later time. The algorithm is stated by the following rules:

- To request a resource, the process sends a timestamped message to all processes.
- On receiving a request from any other process, the process sends an *okay* message if either the process is not interested in the critical section or its own request has a higher timestamp value. Otherwise, that process is kept in a pending queue.
- To release a resource, the process $P_i$ sends *okay* to all the processes in the pending queue.
- Process $P_i$ is granted the resource when it has requested the resource and it has received the *okay* message from every other process in response to its *request* message.

The algorithm is presented formally in Figure 14.1. There are two kinds of messages in the system—*request* messages and *okay* messages. Each process maintains the logical time of its request in the variable *myts*. On receiving any request with a lower timestamp than its own, it replies immediately with *okay*. Otherwise, it adds that process to *pendingQ*. The variable *numOkay* counts the number of *okay* messages received since the request was made.

The above algorithm satisfies safety, liveness, and fairness properties of mutual exclusion. To see the safety property, assume that $P_i$ and $P_j$ are in the critical section concurrently and $P_i$ has the smaller value of the timestamp for its request. $P_j$ can enter the critical section only if it received *okay* for its request. The request made by $P_j$ can only reach $P_i$ after $P_i$ has made its request; otherwise, the timestamp of $P_i$'s request would have been greater because of the rules of the logical clock. From the algorithm, $P_i$ cannot send *okay* unless it has exited from the critical section contradicting our earlier assumption that $P_j$ received *okay* from $P_i$. Thus the safety property is not violated. The process with the least timestamp for its request can never be deferred by any other process, and therefore the algorithm also satisfies liveness. Because processes enter the critical section in the order of the timestamps of the requests, the fairness is also true.

It is easy to see that every critical section execution requires $N-1$ *request* messages and $N-1$ *okay* messages.

## 14.5 Token-Based Algorithms

Token-based algorithms use the auxiliary resource *token* to resolve conflicts in a resource coordination problem. The issue in these algorithms is how the requests for the token are made, maintained, and served. A centralized algorithm is an instance of a token-based algorithm in which the coordinator is responsible for keeping the token. All the requests for the token go to the coordinator.

In a token-ring approach, all processes are organized in a ring. The token circulates around the ring. Any process that wants to enter the critical section waits for the token to arrive at that process. It then grabs the token and enters the critical section. By ensuring that a process enters the critical section only when it has the token, the algorithm guarantees the safety property trivially.

```
$P_i$::
    var
        $pendingQ$: list of process ids initially $null$;
        $myts$: integer initially $\infty$;
        $numOkay$: integer initially 0;

    $request$:
        $myts := logical\_clock$;
        send $request$ with $myts$ to all other processes;
        $numOkay := 0$;

    $receive(u, request)$:
        if $(u.myts < myts)$ then
            send $okay$ to process $u.p$;
        else $append(pendingQ, u.p)$;

    $receive(u, okay)$:
        $numOkay := numOkay + 1$;
        if $(numOkay = N - 1)$ then
            enter_critical_section;

    $release$:
        $myts := \infty$;
        for $j \in pendingQ$ do
            send okay to the process $j$;
        $pendingQ := null$;
```

Figure 14.1: Ricart and Agrawala's algorithm

## 14.6 Quorum-Based Algorithms

Token-based algorithms are vulnerable to failures of processes holding the token. We now present quorum-based algorithms, which do not suffer from such single point of failures. The main idea behind a quorum-based algorithm is that instead of asking permission to enter the critical section from either just one process as in token-based algorithms, or from all processes, as in timestamp-based algorithms in Chapter 14, the permission is sought from a subset of processes called the *request set.* If any two request sets have nonempty intersection, then we are guaranteed that at most one process can have permission to enter the critical section. A simple example of this strategy is that of requiring permission from a majority of processes. In this case, a request set is any subset of processes with at least $\lceil N + 1/2 \rceil$ processes.

We now give some examples of quorum systems.

### 14.6.1 Voting Systems

In these quorum systems, each process is assigned a number of votes. Let the total number of votes in the system be $V$. A quorum is defined to be any subset of processes with a combined number of votes exceeding $V/2$. If each process is assigned a single vote, then such a quorum system is also called a majority voting system.

When applications require *read* or *write* accesses to the critical section, then the voting systems can be generalized to two kinds of quorums—*read* quorums and *write* quorums. These quorums are defined by two parameters $R$ and $W$ such that $R + W > V$ and $W > V/2$. For a subset of processes if the combined number of votes exceeds $R$, then it is a *read* quorum and if it exceeds $W$, then it is a *write* quorum.

### 14.6.2 Crumbling Walls

To obtain quorums for crumbling walls, processes are logically arranged in rows of possibly different widths. A quorum in a crumbling wall is the union of one full row and a representative from every row below the full rows. For example, consider a system with 9 processes such that $P_1$ to $P_3$ are in row 1, $P_4$ to $P_6$ are in row 2 and $P_7$ to $P_9$ are in row 3. In this system, $\{P_4, P_5, P_6, P_9\}$ is a quorum because it contains the entire second row and a representative, $P_9$, from the third row. Let $CW(n_1, n_2, \ldots, n_d)$ be a wall with $d$ rows of width $n_1$, $n_2$, $\ldots$, $n_d$, respectively. We assume that processes in the wall are numbered sequentially from left to right and top to bottom. Our earlier example of the crumbling wall can be concisely written as $CW(3, 3, 3)$. $CW(1)$ denotes a wall with a single row of width 1. This corresponds to a centralized algorithm. The crumbling wall $CW(1, N-1)$ is called the wheel coterie because it has $N-1$ "spoke" quorums of the form $\{1, i\}$ for $i = 2, \ldots, N$ and one "rim" quorum $\{2, \ldots, N\}$. In a triangular quorum system, processes are arranged in a triangle such that the $i^{th}$ row has $i$ processes. If there are $d$ rows, then each quorum has exactly $d$ processes. In a grid quorum system, $N (= d^2)$ processes are arranged in a grid such that there are $d$ rows each with $d$ processes. A quorum consists of the union of one full row and a representative from every row below the full rows.

It is important to recognize that the simple strategy of getting permission to enter the critical section from one of the quorums can result in a deadlock. In the majority voting system, if two requests gather $N/2$ votes each (for an even value of $N$), then neither of the requests will be granted. Quorum-based systems require additional messages to ensure that the system is deadlock free. The details of ensuring deadlock freedom are left to the reader (see Problem 14.4).

## 14.7 Problems

14.1. The mutual exclusion algorithm by Lamport requires that any request message be acknowledged. Under what conditions does a process not need to send an *acknowledgement* message for a *request* message?

14.2. Some applications require two types of accesses to the critical section—*read* access and *write* access. For these applications, it is reasonable for two *read* accesses to happen concurrently. However, a *write* access cannot happen concurrently with either a *read* access or a *write* access. Modify algorithms presented in this chapter for such applications.

14.3. (due to [Ray89]) In the decentralized algorithm, a process is required to send the message to everybody to request the token. Design an algorithm in which all processes are organized in the form of a logical binary tree. The edges in the tree are directed as follows. Each node except the one with the token has exactly one outgoing edge such that if that edge is followed it will lead to the node with the token. Give the actions required for requesting and releasing the critical section. What is the message complexity of your algorithm?

14.4. (due to [Mae85]) Let all processes be organized in a rectangular grid. We allow a process to enter the critical section only if it has permission from all the processes in its row and its column. A process grants permission to another process only if it has not given permission to some other process. What properties does this algorithm satisfy? What is the message complexity of the algorithm? How will you ensure deadlock-freedom?

14.5. Compare all the algorithms for mutual exclusion discussed in this book using the following metrics: the response time and the number of messages.

14.6. Discuss how you will extend each of the mutual exclusion algorithms to tolerate failure of a process. Assume perfect failure detection of a process.

14.7. Extend all algorithms discussed in this chapter to solve $k$-mutual exclusion problem, in which at most $k$ processes can be in the critical section concurrently.

14.8. (due to [AA91]) In the tree-based quorum system, processes are organized in a rooted binary tree. A quorum in the system is defined recursively to be either the union of the root and a quorum in one of the two subtrees, or the union of quorums of subtrees. Analyze this coterie for availability and load.

## 14.8 Bibliographic Remarks

Lamport's algorithm for mutual exclusion first appeared in [Lam78]. The algorithm was presented there as an application of logical clocks. The number of messages per invocation of the critical section in Lamport's algorithm can be reduced as shown by Ricart and Agrawala [RA81]. The tree-based algorithm in the problem set is due to Raymond [Ray89]. The use of majority voting systems for distributed control is due to Thomas [Tho79], and the use of weighted voting systems with $R$ and $W$ parameters is due to Gifford [Gif79]. Maekawa [Mae85] introduced grid-based quorums and quorums based on finite projective planes. The tree-based quorum in the problem set is due to Agrawal and El-Abbadi [AA91]. The triangular quorum systems are due to Lovasz [Lov73]. The notion of *crumbling walls* is due to Peleg and Wool [PW95].

# Chapter 15

# Detecting Termination

## 15.1 Introduction

Consider a computation on a distributed system that is started by a special process called environment. This process starts up the computation by sending messages to some of the processes. Each process in the system is either passive or active. It is assumed that a passive process can become active only on receiving a message (an active process can become passive at any time). Furthermore, a message can be sent by a process only if it is in the active state. Such a computation is called a diffusing computation. Algorithms for many problems such as computing the breadth-first search-spanning tree in an asynchronous network or determining the shortest paths from a processor in a network can be structured as diffusing computations (see Problems 15.3 and 15.4).

From properties of a diffusing computation, it follows that if all processes are passive in the system and there are no messages in transit, then the computation has terminated. Our problem is to design a protocol by which the environment process can determine whether the computation has terminated. Our solution is based on an algorithm by Dijkstra and Scholten.

This chapter is organized as follows. Section 15.2 describes a variant of Dijkstra and Scholten's algorithm, and Section 15.3 gives an optimization of the variant. With the optimization, the algorithm has the same message complexity as Dijkstra and Scholten's algorithm.

## 15.2 Dijkstra and Scholten's Algorithm

We say that a process is in a green state if it is passive and all of its outgoing channels are empty; otherwise, it is in a red state. How can a process determine whether its outgoing channel is empty? This can be done if the receiver of the channel signals the sender of the channel the number of messages received along that channel. If the sender keeps a variable $D[i]$ (for deficit) for each outgoing channel $i$, which records the number of messages sent minus the number of messages that have been acknowledged via signals, it can determine that the channel $i$ is empty by checking whether $D[i] = 0$. Observe that $D[i] \geq 0$ is always true. Therefore, if $O$ is the set of all outgoing channels, it follows that

$$\forall i \in O : D[i] = 0$$

is equivalent to

$$\sum_{i \in O} D[i] = 0.$$

Thus it is sufficient for a process to maintain just one variable $D$ that represents the total deficit for the process.

It is clear that if all processes are in the green state then the computation has terminated. To check this condition, we will maintain a set $T$ with the invariant:

(I0) All red processes are part of the set $T$.

Observe that green processes may also be part of $T$—the invariant is that there is no red process outside $T$. When the set $T$ becomes empty, termination is true.

When the diffusing computation starts, the environment is the only red process initially (with nonempty outgoing channels); the invariant is made true by keeping environment in the set $T$. To maintain the invariant that all red processes are in $T$, we use the following rule. If $P_j$ turns $P_k$ red (by sending a message), and $P_k$ is not in $T$, then we add $P_k$ to $T$.

We now induce a directed graph $(T, E)$ on the set $T$ by defining the set of edges $E$ as follows. We add an edge from $P_j$ to $P_k$, if $P_j$ was responsible for addition of $P_k$ to the set $T$. We say that $P_j$ is the parent of $P_k$. From now on we use the terms *node* and *process* interchangeably. Because every node (other than the environment) has exactly one parent and an edge is drawn from $P_j$ to $P_k$ only when $P_k$ is not part of $T$, the edges $E$ form a spanning tree on $T$ rooted at the environment. Our algorithm will maintain this as invariant:

(I1) The edges $E$ form a spanning tree of nodes in $T$ rooted at the environment.

Up to now, our algorithm only increases the size of $T$. Because detection of termination requires the set to be empty, we clearly need a mechanism to remove nodes from $T$. Our rule for removal is simple: a node is removed from $T$ only if it is a green leaf node. When a node is removed from $T$, the incoming edge to that node is also removed from $E$. Thus the invariants (I0) and (I1) are maintained by this rule. To implement this rule, a node needs to keep track of the number of its children in $T$. This can be implemented by keeping a variable at each node *numchild* initialized to 0 that denotes the number of children it has in $T$. Whenever a new edge is formed, the child reports this to the parent, who can increment the count. When a leaf leaves $T$, it reports this to the parent, who decrements the count. If the node has no parent (it must be the environment) and it leaves the set $T$, then termination is detected. By assuming that a green leaf node eventually reports to its parent, we conclude that once the computation terminates, it is eventually detected. Conversely, if termination is detected, then the computation has indeed terminated on account of the invariant (I0).

Observe that the property that a node is green is not stable and hence a node, say $P_k$, that is green may become active once again on receiving a message. However, because a message can only be sent by an active process we know that some active process (which is already a part of the spanning tree) will be now responsible for the node $P_k$. Thus the tree $T$ changes with time but maintains the invariant that all active nodes are part of the tree.

## 15.3 An Optimization

The algorithm given above can be optimized by combining messages from the reporting process and the messages for detecting whether a node is green. To detect whether an outgoing channel is empty we assumed a mechanism by which the receiver tells the sender the number of messages it has received. One implementation could be based on control messages called signal. For every message received, a node is eventually required to send a signal message to the sender. To avoid the use of report messages, we require that a node not send the signal message for the message that made it active until it is ready to report to leave $T$. When it is ready to report, the signal message for the message that made it active is sent. With this constraint we get an additional property that a node will not turn green unless all its children in the tree have reported. Thus we have also eliminated the need for maintaining *numchild*: only a leaf node in the tree can be green. A node is ready to report when it has turned green, that is, it is passive and $D = 0$. The algorithm obtained after the optimization is shown in Figure 15.1

```
**var**
    $state$ : $\{passive, active\}$ initially $passive$ except for environment;
    $D$: integer initially $0$;
    $parent$: process id initially $null$;

Upon receiving a message from $P_j$:
    **if** $(parent = null)$ **then**
        $parent := P_j$;
        $state := active$;
    **else** send signal to $P_j$;

Upon receiving a signal:
    $D := D - 1$;

To send a message (**enabled if** $(state = active)$):
    $D := D + 1$;

on $(state = passive) \wedge (D = 0) \wedge (parent \neq null)$:
    send signal to $parent$;
    $parent := null$;

environment detects termination when $(D = 0)$.
```

Figure 15.1: Termination detection algorithm

## 15.4 Problems

15.1. What is the message complexity of Dijkstra and Scholten's algorithm?

15.2. Assume that all processes in the system are organized as a logical ring. Design an algorithm in which a token circulates in the ring carrying appropriate information to detect termination.

15.3. Give an algorithm based on diffusing computation to determine the breadth-first search tree from a given processor.

15.4. Consider a network of processors connected by bidirectional links with different but fixed delays. Our interest is in determining the shortest path from all processors to a fixed processor, say $P_0$. Design a diffusing computation in which each processor maintains two variables: *cost* and *parent*. The *cost* represents the cost of the shortest path from that processor to $P_0$, and the *parent* points to the neighbor that is the next node in the path to $P_0$. Assume that each node knows the delay on links to its neighbors.

15.5. Assume that each process uses a simple acknowledgment scheme. Whenever a message is received it is acknowledged. Further assume that each process maintains a variable $D$ that records the total deficit for that process. Define a process $P_i$ to satisfy the local predicates $l_i$ if it is passive and $D = 0$. Show that detecting termination is equivalent to WCP detection with this setup. What is the message complexity of centralized WCP algorithm if used in this manner?

15.6. Consider the WCP detection problem. Assume that a monitor process responsible for WCP detection runs with every application process. The application process sends its state interval index and the dependence list whenever the local predicate becomes true. Show that a termination detection algorithm can be used to solve the WCP detection problem.

15.7. Extend Dijkstra and Scholten's algorithm for the case when there can be multiple initiators of the diffusing computation.

15.8. Because termination is equivalent to conjunction of local predicates and channel predicates we can employ the algorithms in Chapter **??** for detecting termination. Compare the message complexity of those algorithms for detecting termination with the algorithm in this chapter.

## 15.5 Bibliographic Remarks

The algorithm discussed in this chapter is a slight variant of the algorithm by Dijkstra and Scholten [**?**].

**Chapter 16**

# Detecting Global Predicates

## 16.1 Introduction

One of the difficulties in a distributed system is that no process has access to the global state of the system, that is, it is impossible for a process to know the current global state of the system (unless the computation is frozen). For many applications, it is sufficient to capture a global state that happened in the *past* instead of the *current* global state. For example, in case of a failure the system can restart from such a global state. As another example, suppose that we were interested in monitoring the system for the property that the token in the system has been lost. This property is *stable*, that is, once it is true it stays true forever; therefore, we can check this property on an old global state. If the token is found to be missing in the old global state, then we can conclude that the token is also missing in the current global state. An algorithm that captures a global state is called a *global snapshot algorithm.*

A global snapshot algorithm provides us a useful tool in building distributed systems. Computing a global snapshot is beautifully exemplified by Chandy and Lamport as the problem of taking a picture of a big scene such as a sky filled with birds. The scene is so big that it cannot be captured by a single photograph, and therefore multiple photographs must be taken and composed together to form the global picture. The multiple photographs cannot be taken at the same time instant because there is no shared physical clock in a distributed system. Furthermore, the act of taking a picture cannot change the behavior of the underlying process. Thus birds may fly from one part of the sky to the other while the local pictures are being taken. Despite these problems, we require that the composite picture should be meaningful. For example, it should give us an accurate count of the number of birds. We first need to define what is meant by "meaningful" global state.

Consider the following definition of a global state: A *global state* is a set of local states that occur simultaneously. This definition is based on physical time. We use the phrase "time-based model" to refer to such a definition. A different definition of a global state is possible based on the "happened before model." In the happened before model, a global state is a set of local states that are all concurrent with each other. By concurrent, we mean that no two states have a happened before relationship with each other. A global state in the time-based model is also a global state in the happened before model; if two states occur simultaneously, then they cannot have any happened before relationship. However, the converse is not true; two concurrent states may or may not occur simultaneously in a given execution.

We choose to use the definition for the global state from the happened before model for two reasons. First, it is impossible to determine whether a given global state occurs in the time-based model without access to perfectly synchronized local clocks. For example, the statement "there exists a global state in which more than two processes have access to the critical section" cannot be verified in the time-based model. In the happened before model, however, it is possible to determine whether a given global state occurs. Second, program properties that are of interest are often more simply stated in the happened before model than in the time-based model, which makes them easier to understand and manipulate. This simplicity and elegance is gained because the happened before model inherently accounts for different execution schedules. For example, an execution that does not violate mutual exclusion in the time-based model may do so with a different execution schedule. This problem is avoided in the happened before model.

It is instructive to observe that a consistent global state is not simply a product of local states. To appreciate this, consider a distributed database for a banking application. Assume for simplicity that there are only two sites which keep the accounts for a customer. Also assume that the customer has \$500 at the first site and \$300 at the second site. In the absence of any communication between these sites, the total money of the customer can be easily computed to be \$800. However, if there is a transfer of \$200 from site A to site B, and a simple procedure is used to add up the accounts, we may falsely report that the customer has a total of \$1000 in his accounts (to the chagrin of the bank). This happens when the value at the first site is used before the transfer and the value at the second site after the transfer. It is easily seen that these two states are not concurrent. Note that \$1,000 cannot be justified even by the messages in transit (or, that "the check is in the mail").

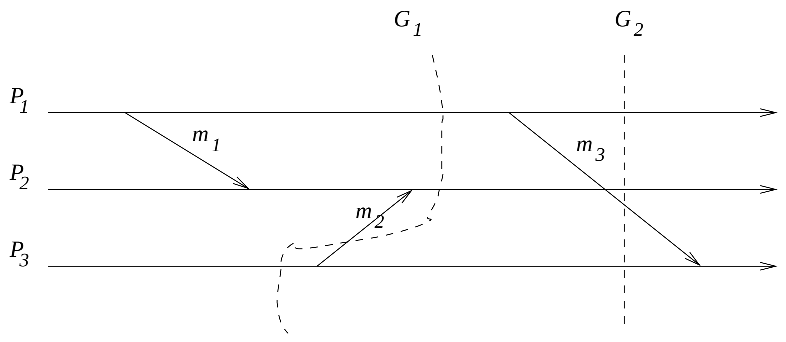

Figure 16.1: Consistent and inconsistent cuts

Figure 16.1 depicts a distributed computation. The dashed lines labeled $G_1$ and $G_2$ represent global states that consist of local states at $P_1$, $P_2$, and $P_3$ where $G_1$ and $G_2$ intersect. Because a global state can be visualized in such a figure as a *cut* across the computation, the term "cut" and "global snapshot" are used interchangeably with "global state." The cut $G_1$ in this computation is not consistent because it records the message $m_2$ as having been received but not sent. This is clearly impossible. The cut $G_2$ is consistent. The message $m_3$ in this cut has been sent but not yet received. Thus it is a part of the channel from process $P_1$ to $P_3$. Our example of the distributed database illustrates the importance of recording only the consistent cuts.

## 16.2 Global Snapshot Algorithm

In this section, we describe an algorithm to take a global snapshot of a distributed system. The algorithm computes a consistent cut or a consistent subcut as desired. The computation of the snapshot is initiated by one or more processes. We assume that all channels are unidirectional and satisfy the FIFO property. Assuming that channels are unidirectional is not restrictive because a bidirectional channel can simply be modeled by using two unidirectional channels. The assumption that channels are FIFO is essential to the correctness of the algorithm as explained later.

The algorithm is shown in Figure **??**. We associate with each process a variable called *color* that is either white or red. Intuitively, the computed global snapshot corresponds to the state of the system just before the processes turn red. All processes are initially white. After recording the local state, a process turns red. Thus the state of a local process is simply the state just before it turned red.

There are two difficulties in the design of rules for changing the color for the global snapshot algorithm. First, we need to ensure that the recorded local states are mutually concurrent. Second, we also need a mechanism to capture the state of the channels. To address these difficulties, the algorithm relies on a special message called a *marker*. Once a process turns red, it is required to send a *marker* along all its outgoing channels before it sends out any message. A process is required to turn *red* on receiving a marker if it has not already done so. Since channels are FIFO, the above rule guarantees that no white process ever receives a message sent by a red process. This in turn guarantees that local states are mutually concurrent.

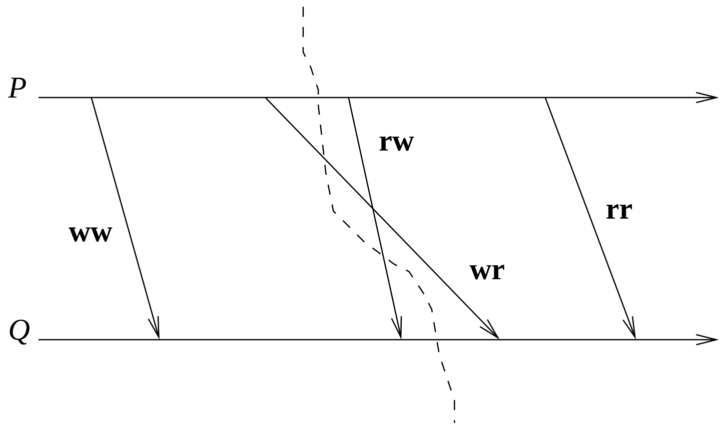

Figure 16.2: Classification of messages

Now let us turn our attention to the problem of computing states of the channels. Figure 16.2 shows that messages in the presence of colors can be of four types:

1. **(ww messages)** These are the messages sent by a white process to a white process. These messages correspond to the messages sent and received before the global snapshot.

2. **(rr messages)** These are the messages sent by a red process to a red process. These messages correspond to the messages sent and received after the global snapshot.

3. **(rw messages)** These are the messages sent by a red process received by a white process. In the figure, they cross the global snapshot in the backward direction. The presence of any such message makes the global snapshot inconsistent. The reader should verify that such messages are not possible if a *marker* is used.

4. **(wr messages)** These are the messages sent by a white process received by a red process. These messages cross the global snapshot, or cut, in the forward direction and form the state of the channel in the global snapshot because they are in transit when the snapshot is taken.

To record the state of the channel, $P_j$ starts recording all messages it receives from $P_i$ after turning red. Since $P_i$ sends a marker to $P_j$ on turning red, the arrival of the marker at $P_j$ from $P_i$ indicates that there will not be any further white messages from $P_i$ sent to $P_j$. It can, therefore, stop recording messages once it has received the marker.

The program shown in Figure **??** uses *chan*$[k]$ to record the state of the $k^{th}$ incoming channel and *closed*$[k]$ to stop recording messages along that channel. In the program, we say that $P_j$ is a *neighbor* of $P_i$ if there is a channel from $P_i$ to $P_j$.

In the algorithm, any change in the value of *color* must be reported to all neighbors. On receiving any such notification, a process is required to update its own color. This may result in additional messages because of the semantics of the event *turn_red*. The net result is that if one process turns red, all processes that can be reached directly or indirectly from that process also turn red. Thus if any process turns red, all other processes will also eventually turn red.

The above algorithm requires that a marker be sent along all channels. Thus it has an overhead of $E$ messages where $E$ is the number of unidirectional channels in the system. We have not discussed the overhead required to combine local snapshots into a global snapshot. A simple method would be for all processes to send their local snapshots to a predetermined process, say $P_0$.

```
class camera {
 int myColor = WHITE;
 static final int WHITE = 0, RED = 1;
 boolean closed[];
 int numProc;
 int myID;
 Linker comm;

  public void camera(int N, int id, Linker initComm) {
    numProc = N;
    myID = id;
    comm = initComm;
    closed = new boolean[numProc];
    for(int i=0; i < numProc; i++) closed[i] = false;
    closed[myID] = true;
  }

 public void turnRed() {
    myColor = RED;
    // recordState();
    for(int i=0; i < numProc; i++) // send Markers
      if (i != myID) comm.sendMessage(i, "marker", 0);
 }

  public void markerHandler(int source_id) {
    if (myColor == WHITE) turnRed();
    closed[source_id] = true;
    checkDone();
  }

  public void checkDone() {
    boolean done = true;
    if(myColor == WHITE) done = false;
    for(int i=0; i < numProc; i++) {
      if (closed[i] == false) done = false;
    }
    if (done) System.out.println("done");
  }

  public void recordChannel(int logID, int event, int msg, int len) {
    if (closed[logID] == false)
      chan[logID] = chan[logID] + event + msg;
  }
}
```

### 16.2.1 Application: Detecting Stable Properties

Computation of a global snapshot is useful in many contexts. It can be be used to detect a stable property of a distributed computation. To define stable predicates we use the notion of the reachability of one global state from another. For two global states $G$ and $H$, we say that $G \leq H$ if $H$ is reachable from $G$. A predicate $B$ is **stable** iff

$$\forall G, H : G \leq H : B(G) \Rightarrow B(H)$$

In other words, a property $B$ is stable if once it becomes true, it stays true. Some examples of stable properties are deadlock, termination, and loss of a token. Once a system has deadlocked or terminated, it remains in that state. A simple algorithm to detect a stable property is as follows. Compute a global snapshot $S$. If the property $B$ is true in the state $S$, then we are done. Otherwise, we repeat the process after some delay. It is easily seen that if the stable property ever becomes true, the algorithm will detect it. Conversely, if the algorithm detects that some stable property $B$ is true, then the property must have become true in the past (and is therefore also true currently).

Formally, if the global snapshot computation was started in the global state $G_i$, the algorithm finished by the global state $G_f$, and the recorded state is $G_*$, then the following is true.

1. $B(G_*) \Rightarrow B(G_f)$
2. $\neg B(G_*) \Rightarrow \neg B(G_i)$

Note that the converse of (1) and (2) may not hold.

The global snapshot algorithm can also be used for providing fault tolerance in distributed systems. On failure, the system can be restarted from the last snapshot. Finally, snapshots can also be used for distributed debugging. Inspection of intermediate snapshots may sometimes reveal the source of an error.

At this point it is important to observe some limitations of the snapshot algorithm for detection of global properties.

- The algorithm is not useful for unstable predicates. An unstable predicate may turn true only between two snapshots.
- In many applications (such as debugging), it is desirable to compute the least global state that satisfies some given predicate. The snapshot algorithm cannot be used for this purpose.
- The algorithm may result in an excessive overhead depending on the frequency of snapshots. A process in Chandy and Lamport's algorithm is forced to take a local snapshot upon receiving a marker even if it knows that the global snapshot that includes its local snapshot cannot satisfy the predicate being detected. For example, suppose that the property being detected is termination. Clearly, if a process is not terminated then the entire system could not have terminated. In this case, computation of the global snapshot is a wasted effort.

## 16.3 Unstable Predicate Detection

In this section, we discuss an algorithm to detect unstable predicates. We will assume that the given global predicate, say $B$, is constructed from local predicates using boolean connectives. We first show that $B$ can be detected using an algorithm that can detect $q$, where $q$ is a pure conjunction of local predicates. The predicate $B$ can be rewritten in its disjunctive normal form. Thus,

$$B = q_1 \vee \ldots \vee q_k \qquad k \geq 1$$

where each $q_i$ is a pure conjunction of local predicates. Next, observe that a global cut satisfies $B$ if and only if it satisfies at least one of the $q_i$'s. Thus the problem of detecting $B$ is reduced to solving $k$ problems of detecting $q$, where $q$ is a pure conjunction of local predicates.

As an example, consider a distributed program in which $x, y$ and $z$ are in three different processes. Then,

$$even(x) \wedge ((y < 0) \vee (z > 6))$$

can be rewritten as

$$(even(x) \wedge (y < 0)) \vee (even(x) \wedge (z > 6))$$

where each disjunct is a conjunctive predicate.

Note that even if the global predicate is not a boolean expression of local predicates, but is satisfied by a finite number of possible global states, then it can also be rewritten as a disjunction of conjunctive predicates. For example, consider the predicate $(x = y)$, where $x$ and $y$ are in different processes. $(x = y)$ is not a *local* predicate because it depends on both processes. However, if we know that $x$ and $y$ can only take values $\{0, 1\}$, then the above expression can be rewritten as

$$((x = 0) \wedge (y = 0)) \vee ((x = 1) \wedge (y = 1)).$$

Each of the disjuncts in this expression is a conjunctive predicate.

We have emphasized conjunctive predicates and not disjunctive predicates. The reason is that disjunctive predicates are quite simple to detect. To detect a disjunctive predicate $l_1 \vee l_2 \vee \ldots \vee l_n$, where $l_i$ denotes a local predicate in the process $P_i$, it is sufficient for the process $P_i$ to monitor $l_i$. If any of the processes finds its local predicate true, then the disjunctive predicate is true.

Formally, we define a weak conjunctive predicate (WCP) to be true for a given run if and only if there exists a consistent global cut in that run in which all conjuncts are true. Intuitively, detecting a weak conjunctive predicate is generally useful when one is interested in detecting a combination of states that is unsafe. For example, violation of mutual exclusion for a two process system can be written as: "$P_1$ is in the critical section and $P_2$ is in the critical section." We first give a necessary and sufficient condition for a weak conjunctive predicate to be true in a computation $(S_1, S_2, \ldots, S_N, \rightsquigarrow)$ where $S_i$ is the set of states on process $P_i$. Let $l_i(s)$ denote that the predicate $l_i$ is true in the state $s$.

Our aim is to detect whether $(l_1 \wedge l_2 \wedge \ldots l_n)$ holds for a given computation. We can assume $n \leq N$ (the total number of processes in the system) because $l_i \wedge l_j$ is just another local predicate if $l_i$ and $l_j$ belong to the same process. From the definition of the consistent state, we get that

$(l_1 \wedge l_2 \wedge \ldots l_n)$ is in a consistent global state of a computation iff for all $1 \leq i \leq n$, $\exists s_i \in S_i$ such that $l_i$ is true in state $s_i$, and $s_i$ and $s_j$ are concurrent for $i \neq j$.

Thus it is necessary and sufficient to find a set of incomparable states in which local predicates are true to detect a weak conjunctive predicate. We now present an algorithm to do so. This algorithm finds the *first* consistent cut for which a WCP is true.

In this algorithm, one process serves as a checker. All other processes involved in detecting the WCP are referred to as application processes. Each application process checks for local predicates. It also maintains the vector clock algorithm. Whenever the local predicate of a process becomes true for the *first* time since the most recently sent message (or the beginning of the trace), it generates a debug message containing its local timestamp vector and sends it to the checker process.

Note that a process is not required to send its vector clock every time the local predicate is detected. If two local states, say $s$ and $t$, on the same process are separated only by internal events, then they are indistinguishable to other processes so far as consistency is concerned, that is, if $u$ is a local state on some other process, then $s||u$ if and only if $t||u$. Thus it is sufficient to consider at most one local state between two external events and the vector clock need not be sent if there has been no message activity since the last time the vector clock was sent.

The checker process is responsible for searching for a consistent cut that satisfies the WCP by considering a sequence of candidate cuts. If the candidate cut either is not a consistent cut or does not satisfy some term of the WCP, the checker can efficiently eliminate one of the states along the cut. The eliminated state can never be part of a consistent cut that satisfies the WCP. The checker can then advance the cut by considering the successor to one of the eliminated states on the cut. If the checker finds a cut for which no state can be eliminated, then that cut satisfies the WCP and the detection algorithm halts. The algorithm for the checker process is shown in Figure 16.3.

The checker receives local snapshots from the other processes in the system. These messages are used by the checker to create and maintain data structures that describe the global state of the system for the current cut. The data structures are divided into two categories: queues of incoming messages and those data structures that describe the state of the processes.

```
var
    $cut$: array[1..$n$] of struct
        $v$: array[1..$n$] of integer;
        $color$: $\{red, green\}$;
    endstruct initially $(\forall i : cut[i].color = red)$;
    $detect$: boolean initially $false$;

    while $(\exists i : (cut[i].color = red))$ do
        if $(q[i] = null)$ and $P_i$ terminated then return $false$;
        else $cut[i].v := receive(q[i])$;// advance the cut
        $paintState(i)$;
    endwhile;

$detect := true$;
```

Figure 16.3: WCP detection algorithm—checker process.

The queue of incoming messages is used to hold incoming local snapshots from application processes. We require that messages from an individual process be received in FIFO order. We abstract the message passing system as a set of $n$ FIFO queues, one for each process. We use the notation $q[1 \dots n]$ to label these queues in the algorithm.

The checker also maintains information describing one state from each process $P_i$. The collection of this information is organized into a vector *cut* which is an array of structure consisting of the vector $v$ and *color*. The color of a state is either red or green and indicates whether the state has been eliminated in the current cut. A state is green only if it is concurrent with all other green states. A state is red only if it cannot be part of a consistent cut that satisfies the WCP.

The aim of advancing the cut is to find a new candidate cut. However, we can advance the cut only if we have eliminated at least one state along the current cut and if a message can be received from the corresponding process. The data structures for the processes are updated to reflect the new cut. This is done by the procedure *paintState*. This procedure is shown in Figure 16.4. The parameter $i$ is the index of the process from which a local snapshot was most recently received. The color of $cut[i]$ is temporarily set to green. It may be necessary to change some green states to red to preserve the property that all green states are mutually concurrent. Hence, we must compare the vector clock of $cut[i]$ to each of the other green states. Whenever the states are comparable, the smaller of the two is painted red.

```
$paintState(i)$
    $cut[i].color := green$;
    for $j := 1$ to $n$ do
        if $(cut[j].color = green)$ then
            if $(cut[i].v < cut[j].v)$ then $cut[i].color := red$;
            else if $(cut[j].v < cut[i].v)$ then $cut[j].color := red$;
    endfor
```

Figure 16.4: Procedure *paintState*.

### 16.3.1 Overhead Analysis

Let $n$ denote the number of processes involved in the WCP, and $m$ denote the maximum number of messages sent or received by any process.

The main time complexity is involved in detecting the local predicates and time required to maintain vector clocks. In the worst case, one debug message is generated for each program message sent, so the worst case message complexity is $O(m)$. In addition, program messages have to include time vectors. Every process sends $O(m)$ local snapshots to the checker process. With the same assumptions as made for space complexity, it follows that $O(mn)$ bits are sent by each process.

The main space requirement of the checker process is the buffer for the local snapshots. Each local snapshot consists of a vector clock that requires $O(n)$ space. Since there are at most $O(mn)$ local snapshots, $O(n^2m)$ total space is required to hold the component of local snapshots devoted to vector clocks. Therefore, the total amount of space required by the checker process is $O(n^2m)$.

We now discuss the time complexity of the checker process. Note that it takes only two comparisons to check whether two vectors are concurrent. Hence, each invocation of *paintState* requires at most $n$ comparisons. This function is called at most once for each state, and there are at most $mn$ states. Therefore, at most $n^2m$ comparisons are required by the algorithm.

## 16.4 Problems

16.1. Chandy and Lamport's algorithm requires the receiver to record the state of the channel. Since messages in real channels may get lost, it may be advantageous for senders to record the state of the channel. Give an algorithm to do so. Try to make your algorithm as practical as possible. Assume that control messages can be sent over unidirectional channels even in the reverse direction.

16.2. The original algorithm proposed by Chandy and Lamport does not require FIFO but a condition weaker than that. Specify the condition formally.

16.3. How can you use Lamport's logical clock to compute a consistent global snapshot?

16.4. Assume that the given global predicate is a simple conjunction of local predicates. Further assume that the global predicate is stable. In this scenario, both Chandy and Lamport's algorithm and the weak conjunctive algorithm can be used to detect the global predicate. What are the advantages and disadvantages of using each of them?

# Chapter 17

# Message Ordering

## 17.1 Introduction

Distributed programs are difficult to design and test because of their nondeterministic nature, that is, a distributed program may exhibit multiple behaviors on the same external input. This nondeterminism is caused by reordering of messages in different executions. It is sometimes desirable to control this nondeterminism by restricting the possible message ordering in a system.

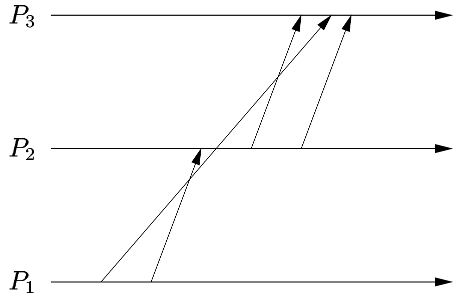

Figure 17.1: A FIFO computation that is not causally ordered

A *fully asynchronous* computation does not have any restriction on the message ordering. It permits maximum concurrency, but algorithms based on fully asynchronous communication can be difficult to design because they are required to work for all ordering of the messages. Therefore, many systems restrict message delivery to a FIFO order. This results in simplicity in design of distributed algorithms based on the FIFO assumption. For example, we used the FIFO assumption in Lamport's algorithm for mutual exclusion and Chandy and Lamport's algorithm for a global snapshot.

A FIFO-ordered computation is implemented generally by using sequence numbers for messages. However, observe that by using FIFO ordering, a program loses some of its concurrency. When a message is received out of order, its processing must be delayed.

A stronger requirement than FIFO is that of *causal ordering.* Intuitively, causal ordering requires that a single message should not be overtaken by a sequence of messages. For example, the computation in Figure 17.1 satisfies FIFO ordering of messages but does not satisfy causal ordering. A sequence of messages from $P_1$ to $P_2$ and from $P_2$ to $P_3$ overtakes a message from $P_1$ to $P_3$ in this example. Causal ordering of messages is useful in many contexts. In Chapter 14, we considered the problem of mutual exclusion. Assume that we use a centralized coordinator for granting requests to the access of the critical section. The fairness property requires that the requests be honored in the order they are made (and not in the order they are received). It is easy to see that if the underlying system guaranteed a causal ordering of messages, then the order in which requests are received cannot violate the order in which they are made. For another example of the usefulness of causal ordering, see Problem 17.1.

The relationship among various message orderings can be formally specified based on the happened before relation. For convenience, we denote the receive event corresponding to the send event $s_i$ by $r_i$ and vice versa. The message is represented as $(s_i, r_i)$. Thus $s_i \rightsquigarrow r_i$ for any $i$.

Now, FIFO and causally ordered computations can be defined as follows.

**FIFO** : Any two messages from a process $P_i$ to $P_j$ are received in the same order as they were sent. Formally, let $s_1$ and $s_2$ be any two send events and $r_1$ and $r_2$ be corresponding receive events. Then,

$$s_1 \prec s_2 \quad \Rightarrow \quad \neg(r_2 \prec r_1). \qquad \text{(FIFO)}$$

**Causally Ordered** : Let any two send events $s_1$ and $s_2$ in a distributed computation be related such that the first send happened before the second send. Then, the second message cannot be received before the first message by any process. Formally,

$$s_1 \rightarrow s_2 \quad \Rightarrow \quad \neg(r_2 \prec r_1). \qquad \text{(CO)}$$

## 17.2 Algorithm

We now describe an algorithm to ensure causal ordering of messages. We assume that a process never sends any message to itself. Each process maintains a matrix $m$ of integers. The entry $s.m[j,k]$ records the number of messages sent by process $P_j$ to process $P_k$ as known by process $P_{s.p}$ in the state $s$. The algorithm for process $P_i$ is given in Figure 17.2. Whenever a message is sent from $P_i$ to $P_j$, the matrix $m$ is piggybacked with the message. The entry $m[i,j]$ is incremented to reflect the fact that one more message has been sent from $P_i$ to $P_j$. Whenever messages are received by the communication system at $P_i$, they are first checked for eligibility before delivery to $P_i$. If a message is not eligible it is simply buffered until it becomes eligible. In the following discussion, we identify a message with the state it is sent from. A message $u$ is eligible to be received at state $s$, when the number of messages sent from any process $P_k$ to $P_i$, as indicated by the matrix $u.m$ in the message, is less than or equal to the number recorded in the matrix $s.m$. Formally, this condition is

$$\forall k : s.m[k,i] \geq u.m[k,i]$$

If for some $k$, $u.m[k,i] > s.m[k,i]$, then there is a message that was sent in the casual history of the message $u$ and has not arrived by the state $s$. Therefore, $P_i$ must wait for that message to be delivered before it can accept the message $u$.

## 17.3 Synchronous and Total Ordering

Synchronous ordering is a stronger requirement than causal ordering. A computation satisfies synchronous ordering of messages if it is equivalent to a computation in which all messages are logically instantaneous. Figure 17.3 gives an example of a synchronously ordered computation and Figure 17.4 an example of a computation that does not satisfy synchronous ordering.

Algorithms for synchronous systems are easier to design than those for causally ordered systems. The model of synchronous message passing lets us reason about a distributed program under the assumption that messages are instantaneous or "points" rather then "intervals" (i.e., we can always draw the time diagrams for the distributed programs with the message arrows being vertical). If we assume messages as points instead of intervals, we can order the messages as a partial order and therefore, we can have vector clocks with respect to messages. One of the applications for synchronous ordering of messages is that it enables us to reason about distributed objects as if they were centralized. Assume that a process invokes an operation on a remote object by sending a message. If

```
$P_i$::
    var
        $m$:array[1..$N$, 1..$N$] of integer initially $\forall j,k : m[j,k] = 0$;

    To send a message to $P_j$:
        piggyback $m$ as part of the message;
        $m[i,j] := m[i,j] + 1$;

    To receive a message $u$ from $P_j$enabled if $\forall k : m[k,i] \geq u.m[k,i]$
        $m := max(m, u.m)$;
        $m[j,i] := m[j,i] + 1$;
```

Figure 17.2: An algorithm for causal ordering of messages at $P_i$

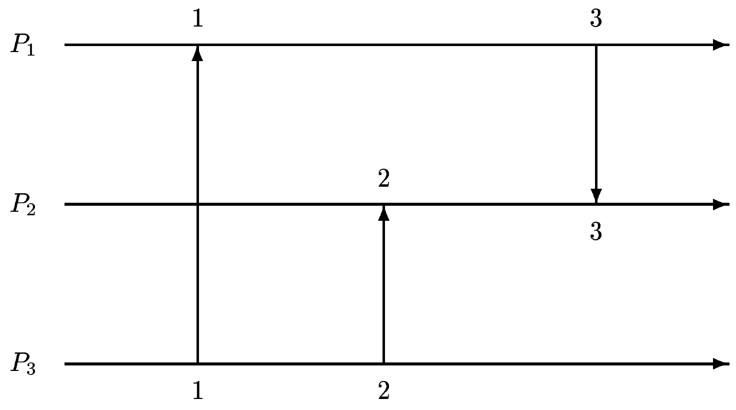

Figure 17.3: A computation that is synchronously ordered

synchronous ordering of messages is assumed, then all operations on the objects can be ordered based on when the messages are sent because messages can be considered instantaneous.

*Total ordering* of messages is useful when messages may be multicast. One of the applications of total ordering of messages is in implementing an object that is replicated across multiple sites. If different processes invoke different operations on the object, the sequence of operations applied at each site must be identical. Implementing this constraint in a distributed system is significantly easier if there is a total ordering of messages.

### 17.3.1 Synchronous Ordering

A computation is synchronous if its time diagram can be drawn such that all message arrows are vertical, that is, all external events can be assigned a timestamp such that time increases within a single process and for any message its send and receive are assigned the same timestamp. Formally, let $\mathcal{E}$ be the set of all external events. Then, a computation is synchronous iff there exists a mapping T from $\mathcal{E}$ to the set of natural numbers such that for all $s, r, e, f \in \mathcal{E}$:

$$s \leadsto r \Rightarrow \mathrm{T}(s) = \mathrm{T}(r)$$

and

$$e \prec f \Rightarrow \mathrm{T}(e) < \mathrm{T}(f).$$

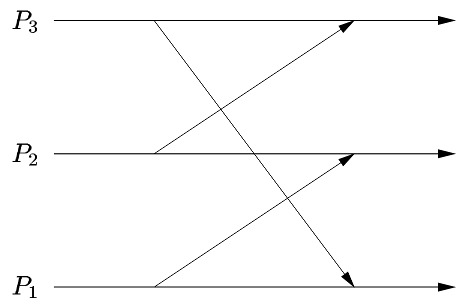

Figure 17.4: A computation that is not synchronously ordered

We call this condition (SYNC). It is easy to see that, for any two external events $e$ and $f$

$$(e \rightarrow f) \land \neg(e \rightsquigarrow f) \quad \Rightarrow \quad \mathrm{T}(e) < \mathrm{T}(f). \tag{17.1}$$

### 17.3.2 Relationship Among Message Orderings

We show that the hierarchy associated with the various message orderings is

$$\text{Synchronous} \subseteq \text{Causally Ordered} \subseteq \text{FIFO} \subseteq \text{Asynchronous}.$$

*FIFO* $\subseteq$ *Asynchronous* is obvious. A causally Ordered conputation satisfies FIFO because

$$s_1 \prec s_2 \quad \Rightarrow \quad s_1 \rightarrow s_2.$$

We only need to show that if a computation is synchronous then it is also causally ordered. Because the communication is synchronous, there exists a function T satisfying SYNC.

For any set of send events $s_1, s_2$ and receive events $r_1, r_2$ such that $s_1 \rightsquigarrow r_1$, $s_2 \rightsquigarrow r_2$ and $s_1 \rightarrow s_2$:

$$\mathrm{T}(s_1) = \mathrm{T}(r_1), \quad \mathrm{T}(s_2) = \mathrm{T}(r_2), \quad \text{and} \quad \mathrm{T}(s_1) < \mathrm{T}(s_2).$$

It follows that $\mathrm{T}(r_1) < \mathrm{T}(r_2)$. Therefore, (17.1) implies

$$\neg(r_2 \rightarrow r_1).$$

### 17.3.3 Algorithm

The algorithm implements the desired ordering using control messages. Note that control messages are not required to satisfy synchronous ordering. Thus $\mathcal{E}$ includes the send and receive events only of application messages. It does not include send and receive of control messages sent by the algorithm to ensure synchronous ordering.

The algorithm shown in Figure 17.5 is for the process $P_i$. All processes have the same algorithm. Observe that the protocol to implement synchronous message ordering cannot be completely symmetric. If two processes desire to send messages to each other, then there is no symmetric synchronous computation that allows this—one of them must succeed before the other. To introduce asymmetry, we use process numbers to totally order all processes. Each send event, $s$, is either from a bigger process (denoted by $big(s)$) or from a smaller process (denoted by $small(s)$). We assume that processes do not send messages to themselves.

In our algorithm, a process can be in two states—*active* or *passive*. Every process is initially active. We first consider the algorithm for a send event corresponding to $big(s)$. A process is allowed to send a message to a smaller process only when it is active. After sending the message it turns passive until it receives an *ack* message from the receiver of the message. While passive, it cannot send any other message nor can it accept any other message. Note that the protocol for a message from a bigger process requires only one control message (*ack*).

To send a message to a bigger process, say $P_j$, $P_i$ first needs permission from $P_j$. It can request the permission at any time. $P_j$ can grant permission only when it is active. Furthermore, after granting the permission, $P_j$ turns passive until it receives the message for which it has granted the permission. Thus the protocol for a message from a smaller process requires two control messages (*request* and *permission*).

```
$P_i$ ::
    var
        $state$ : $\{active, passive\}$ initially $active$;

    To send $m$ to $P_j$, $(j < i)$
        enabled if $(state = active)$:
        send $m$ to $P_j$
        $state := passive$;

    Upon receive $m$ from $P_j$, $(j > i)$
        enabled if $(state = active)$:
            send ack to $P_j$;

    Upon receive ack:
        $state := active$;

    To send a message $(message\_id, m)$ to $P_j$, $(j > i)$
        send request(message_id) to $P_j$;

    Upon receive request(message_id) from $P_j$, $(j < i)$
        enabled if $(state = active)$:
        send permission(message_id) to $P_j$
        $state := passive$;

    Upon receive permission(message_id) from $P_j$, $(j > i)$
        enabled if $(state = active)$:
        send $m$ to $P_j$;

    Upon receive $m$ from $P_j$, $(j < i)$
        $state := active$;
```

Figure 17.5: The algorithm at $P_i$ for synchronous ordering of messages

## 17.4 Total Order for Multicast Messages

So far we have assumed that messages were point-to-point. In many applications, where a message may be sent to multiple processes, it is desirable that all messages are delivered in the same order at all processes. For example, consider a server that is replicated at multiple sites for fault tolerance. If a client makes a request to the server, then all copies of the server should handle requests in the same order. The total ordering of messages can formally be specified as:

For all messages $x$ and $y$ and all processes $P$ and $Q$, if $x$ is received at $P$ before $y$, then $y$ is not received before $x$ at $Q$. (**Total Order**)

We require that $y$ not be received before $x$, rather than that $x$ be received before $y$, to address the case where $x$ is not sent to $Q$. Observe that we do not require that a message be broadcast to all processes.

In this section we discuss algorithms that will provide the total ordering of messages. Observe that the property of total order of messages does not imply causal or even FIFO property of messages. Consider the case when $P$ sends messages $m_1$ followed by $m_2$. If all processes receive $m_2$ before $m_1$, then the total order is satisfied even though FIFO is not. If messages satisy causal order in addition to the total order, then we will call this ordering of messages *causal total order.*

The algorithms for ensuring total order are very similar to mutual exclusion algorithms. After all, mutual exclusion algorithms ensure that all accesses to the critical section form a total order. If we ensure that messages are received in the "critical section" order, then we are done. We will discuss a centralized algorithm and a distributed algorithm for causal total ordering of messages.

### 17.4.1 Centralized Algorithm

We first modify the centralized algorithm for mutual exclusion to guarantee causal total ordering of messages. We assume that channels between the coordinator process and other processes satisfy the FIFO property. A process that wants to multicast a message simply sends it to the coordinator. This step corresponds to requesting the lock in the mutual exclusion algorithm. Furthermore, in that algorithm, the coordinator maintains a request queue and whenever a request by a process becomes eligible, it sends the lock to that process. In the algorithm for total ordering of messages, the coordinator will simply multicast the message corresponding to the request instead of sending the lock. Since all multicast messages originate from the coordinator, and the channels are FIFO, the total order property holds.

In the above algorithm, the coordinator has to perform more work than the other nodes. One way to perform load balancing over time is by suitably rotating the responsibility of the coordinator among processes. This can be achieved using the notion of a token. The token assigns sequence numbers to broadcasts, and messages are delivered only in the sequence order.

### 17.4.2 Lamport's Algorithm for Total Order

We modify Lamport's algorithm for mutual exclusion to derive an algorithm for total ordering of messages. As in that algorithm, we assume FIFO ordering of messages. We also assume that a message is broadcast to all processes. To simulate multicast, a process can simply ignore a message that is not meant for it. Each process maintains a logical clock (used for timestamps) and a queue (used for storing undelivered messages). The algorithm is given by the following rules.

- To send a broadcast message, a process sends a timestamped message to all other processes including itself. This step corresponds to requesting the critical section in mutual exclusion algorithm.
- On receiving a broadcast message, the message and its timestamp are stored in the queue and an acknowledgment is returned.
- A process can deliver the message from the request queue with the smallest timestamp, $t$, if it has received a message from every other process with timestamp greater than $t$. This step corresponds to executing the critical section for the mutual exclusion algorithm.

In this algorithm, the total order of messages delivered is given by the logical clock of send events of the broadcast messages.

### 17.4.3 Skeen's Algorithm

The distributed algorithm of Skeen is given by the following rules. It also assumes that processes have access to Lamport's logical clock.

- To send a multicast message, a process sends a timestamped message to all the destination processes.
- On receiving a message, a process marks it as *undeliverable* and sends the value of the logical clock as the proposed timestamp to the initiator.
- When the initiator has received all the proposed timestamps, it takes the maximum of all proposals and assigns that timestamp as the final timestamp to that message. This value is sent to all the destinations.
- Upon receiving the final timestamp of a message, it is marked as deliverable.
- A deliverable message is delivered to the site if it has the smallest timestamp in the message queue.

In this algorithm, the total order of message delivery is given by the final timestamps of the messages.

### 17.4.4 Application: Replicated State Machines

Assume that we are interested in providing a fault-tolerant service in a distributed system. The service is expected to process *requests* and provide *outputs*. We would also like the service to tolerate up to $t$ faults where each fault corresponds to a crash of a processor. We can build such a service using $t + 1$ processors in a distributed system as follows. We structure our service as a *deterministic* state machine. This means that if each nonfaulty processor starts in the same initial state and executes the requests in the same order, then each will produce the same output. Thus by combining outputs of the collection we can get a $t$ fault-tolerant service. The key requirement for implementation is that all state machines process all requests in the same order. The total ordering of messages (for example, Lamport's algorithm) satisfies this property.

## 17.5 Problem

17.1. Assume that you have replicated data for fault tolerance. Any file (or a record) may be replicated at more than one site. To avoid updating two copies of the data, assume that a token-based scheme is used. Any site possessing the token can update the file and broadcast the update to all sites which have that file. Show that if the communication is guaranteed to be causally ordered, then the above scheme will ensure that all updates at all sites happen in the same order.

17.2. Let $M$ be the set of messages in a distributed computation. Given a message $x$, we use $x.s$ to denote the send event and $x.r$ to denote the receive event. We say that a computation is *causally* ordered if

$$\forall x, y \in M : (x.s \to y.s) \Rightarrow \neg(y.r \to x.r).$$

We say that a computation is *mysteriously* ordered if

$$\forall x, y \in M : (x.s \to y.r) \Rightarrow \neg(y.s \to x.r).$$

(a) Prove or disprove that every causally ordered computation is also mysteriously ordered.
(b) Prove or disprove that every mysteriously ordered computation is also causally ordered.

17.3. Show the relationship between conditions (C1), (C2), and (C3) on message delivery of a system.

$$(C1) \qquad s_1 \to s_2 \Rightarrow \neg(r_2 \to r_1)$$

$$(C2) \qquad s_1 \prec s_2 \Rightarrow \neg(r_2 \to r_1)$$

$$(C3) \qquad s_1 \rightarrow s_2 \Rightarrow \neg(r_2 \prec r_1)$$

where $s_1$ and $s_2$ are sends of any two messages and $r_1$ and $r_2$ are corresponding receives. Note that a computation satisfies a delivery condition if and only if the condition is true for all pairs of messages.

17.4. How will the algorithm for causal ordering change if messages can be multicast instead of point to point?

17.5. Assume that all messages are broadcast messages. How can you simplify the algorithm for guaranteeing causal ordering of messages under this condition?

17.6. Consider a system of $N+1$ processes $\{P_0, P_1, \ldots, P_N\}$ in which processes $P_1$ through $P_N$ can only send messages to $P_0$ or receive messages from $P_0$. Show that if all channels in the system are FIFO, then any computation on this system is causally ordered.

17.7. In this chapter, we have used the happened before model for modeling dependency of one message to the other. Thus all messages within a process are totally ordered. For some applications, messages sent from a process may be independent. Give an algorithm to ensure causal ordering of messages when the send events from a single process do not form a total order.

17.8. Suppose that the system is composed of nonoverlapping groups such that any communication outside the group is always through the group leader, that is, only group leaders are permitted to send or receive messages from outside the group. How will you exploit this structure to reduce the overhead in causal ordering of messages?

17.9. Design an algorithm for synchronous ordering for point-to-point messages that does not use a static priority scheme. (Hint: Impose an acyclic directed graph on processes. The edge from $P_i$ to $P_j$ means that $P_i$ is bigger than $P_j$ for the purpose of sending messages. Give a rule by which the direction of edges is reversed, such that acyclicity of the graph is maintained.)

17.10. Prove the correctness of Lamport's algorithm for providing causal total ordering of messages.

17.11. Prove the correctness of Skeen's algorithm for providing total ordering of messages.

# Bibliography

[AA91] D. Agrawal and A. E. Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, February 1991.

[AW98] H. Attiya and J. Welch. *Distributed Computing - Fundamentals, Simulations and Advanced Topics.* McGraw Hill, Berkshire, SL6 2QL, England, 1998.

[Bar96] V. Barbosa. *An Introduction to Distributed Algorithms.* The MIT Press, Cambridge, Massachusetts, 1996.

[CDK94] G. Couloris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design.* Addison-Wesley, 1994.

[CJ97] R. Chow and T. Johnson. *Distributed Operating Systems and Algorithms.* Addison-Wesley Longman, Readings, Massachusetts, 1997.

[CM89] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1989.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[Gar96] V. K. Garg. *Principles of Distributed Systems.* Kluwer Academic Publishers, Boston, MA, 1996.

[Gif79] D. K. Gifford. Weighted voting for replicated data. *Seventh SOSP,*, OSR 13(5):150–162, December, 1979.

[Gos91] A. Goscinski. *Distributed Operating Systems, The Logical Design.* Addison-Wesley, 1991. ISBN 0-201-41704-9.

[HA90] Phillip Hutto and Mustaque Ahamad. Slow memory : Weakening consistency to enhance concurreny in distributed shared memories. *Proceedings of Tenth International Conference on Distributed Computing Systems*, May 1990.

[HW90] M. P. Herlihy and J. M. Wing. Linerizability: A correctness condition for atomic objects. *TOPLAS*, 12(3):463–492, July 1990.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam79] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEETC: IEEE Transactions on Computers*, 46, 1979.

[Lam86] L. Lamport. On interprocess communication, part II: Algorithms. *Distributed Computing*, 1:86–101, 1986.

[Lov73] L. Lovasz. Coverings and colorings of hypergraphs. In *4th Southeastern Conference on Combinatorics, Graph Theory, and Computing*, pages 3–12, 1973.

[Lyn96] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1996. Prepared with $\LaTeX$.

[Mae85] M. Maekawa. A square root $N$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, 3(2):145–159, May 1985.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[Mul94] S. Mullender. *Distributed Systems, edited book.* Addison-Wesley, Reading, MA, 1994.

[PW95] D. Peleg and A. Wool. Crumbling walls: a class of practical and efficient quorum systems. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC '95)*, pages 120–129, New York, August 1995. ACM.

[RA81] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24, 1981.

[Ray88] M. Raynal. *Distributed Algorithms and Protocols.* John Wiley & Sons, 1988.

[Ray89] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, February 1989.

[SS94] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems.* McGraw Hill, 1994.

[SY85] R. E. Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[Tel94] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, Cambridge, England, 1994.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[TvS02] Andrew S. Tanenbaum and Maarten van Steen. Prentice Hall, 2002.

[YM94] Z. Yang and T. A. Marsland. Introduction. In Z. Yang and T. A. Marsland, editors, *Global State and Time in Distributed Systems.* IEEE Computer Society Press, 1994.