

**DISTRIBUTED  
MUTUAL  
EXCLUSION**

## 6.1 INTRODUCTION

In the problem of mutual exclusion, concurrent access to a shared resource by several uncoordinated user-requests is serialized to secure the integrity of the shared resource. It requires that the actions performed by a user on a shared resource must be *atomic*. That is, if several users concurrently access a shared resource then the actions performed by a user, as far as the other users are concerned, must be instantaneous and indivisible such that the net effect on the shared resource is the same as if user actions were executed serially, as opposed to in an interleaved manner.

The problem of mutual exclusion frequently arises in distributed systems whenever concurrent access to shared resources by several sites is involved. For correctness, it is necessary that the shared resource be accessed by a single site (or process) at a time. A typical example is directory management, where an update to a directory must be done atomically because if updates and reads to a directory proceed concurrently, reads may obtain inconsistent information. If an entry contains several fields, a read operation may read some fields before the update and some after the update. Mutual exclusion is a fundamental issue in the design of distributed systems and an efficient and robust technique for mutual exclusion is essential to the viable design of distributed systems.

## Mutual exclusion in single-computer systems vs. distributed systems

The problem of mutual exclusion in a single-computer system, where shared memory exists, was studied in Chap. 2. In single-computer systems, the status of a shared resource and the status of users is readily available in the shared memory, and solutions to the mutual exclusion problem can be easily implemented using shared variables (e.g., semaphores). However, in distributed systems, both the shared resources and the users may be distributed and shared memory does not exist. Consequently, approaches based on shared variables are not applicable to distributed systems and approaches based on message passing must be used.

The problem of mutual exclusion becomes much more complex in distributed systems (as compared to single-computer systems) because of the lack of both shared memory and a common physical clock and because of unpredictable message delays. Owing to these factors, it is virtually impossible for a site in a distributed system to have current and complete knowledge of the state of the system.

## 6.2 THE CLASSIFICATION OF MUTUAL EXCLUSION ALGORITHMS

Over the last decade, the problem of mutual exclusion has received considerable attention and several algorithms to achieve mutual exclusion in distributed systems have been proposed. They tend to differ in their communication topology (e.g., tree, ring, and any arbitrary graph) and in the amount of information maintained by each site about other sites. These algorithms can be grouped into two classes. The algorithms in the first class are nontoken-based, e.g., [4, 9, 10, 16, 19]. These algorithms require two or more successive rounds of message exchanges among the sites. These algorithms are assertion based because a site can enter its critical section (CS) when an assertion defined on its local variables becomes true. Mutual exclusion is enforced because the assertion becomes true only at one site at any given time.

The algorithms in the second class are token-based, e.g., [11, 14, 20, 21, 22]. In these algorithms, a unique token (also known as the PRIVILEGE message) is shared among the sites. A site is allowed to enter its CS if it possesses the token and it continues to hold the token until the execution of the CS is over. These algorithms essentially differ in the way a site carries out the search for the token.

In this chapter, we describe several distributed mutual exclusion algorithms and compare their features and performance. We discuss relationship among various mutual exclusion algorithms and examine trade offs among them.

## 6.3 PRELIMINARIES

We now describe the underlying system model and requirements that mutual exclusion algorithms should meet. We also introduce terminology that is used in describing the performance of mutual exclusion algorithms.

**SYSTEM MODEL.** At any instant, a site may have several requests for CS. A site queues up these requests and serves them one at a time. A site can be in one of the following three states: *requesting* CS, *executing* CS, or neither requesting nor executing CS (i.e., *idle*). In the requesting CS state, the site is blocked and cannot make further requests for CS. In the idle state, the site is executing outside its CS. In the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS. Such a state is referred to as an *idle token* state.

### 6.3.1 Requirements of Mutual Exclusion Algorithms

The primary objective of a mutual exclusion algorithm is to maintain mutual exclusion; that is, to guarantee that only one request accesses the CS at a time. In addition, the following characteristics are considered important in a mutual exclusion algorithm:

**Freedom from Deadlocks.** Two or more sites should not endlessly wait for messages that will never arrive.

**Freedom from Starvation.** A site should not be forced to wait indefinitely to execute CS while other sites are repeatedly executing CS. That is, every requesting site should get an opportunity to execute CS in a finite time.

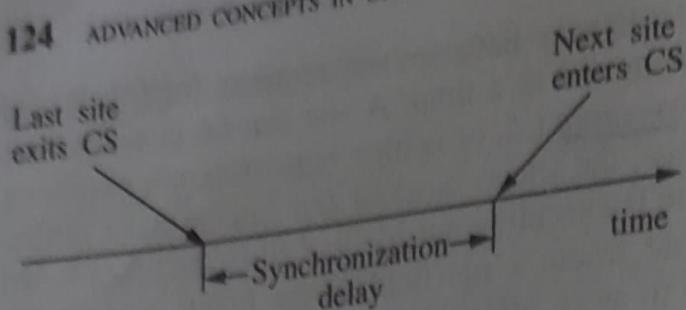
**Fairness.** Fairness dictates that requests must be executed in the order they are made (or the order in which they arrive in the system). Since a physical global clock does not exist, time is determined by logical clocks. Note that fairness implies freedom from starvation, but not vice-versa.

**Fault Tolerance.** A mutual exclusion algorithm is fault-tolerant if in the wake of a failure, it can reorganize itself so that it continues to function without any (prolonged) disruptions.

### 6.3.2 How to Measure the Performance

The performance of mutual exclusion algorithms is generally measured by the following four metrics: First, the *number of messages* necessary per CS invocation. Second, the *synchronization delay*, which is the time required after a site leaves the CS and before the next site enters the CS (see Fig. 6.1). Note that normally one or more sequential message exchanges are required after a site exits the CS and before the next site enters the CS. Third, the *response time*, which is the time interval a request waits for its CS execution to be over after its request messages have been sent out (see Fig. 6.2). Thus, response time does not include the time a request waits at a site before its request messages have been sent out. Fourth, the *system throughput*, which is the rate at which the system executes requests for the CS. If  $sd$  is the synchronization delay and  $E$  is the average critical section execution time, then the throughput is given by the following equation:

$$\text{system throughput} = 1/(sd + E)$$

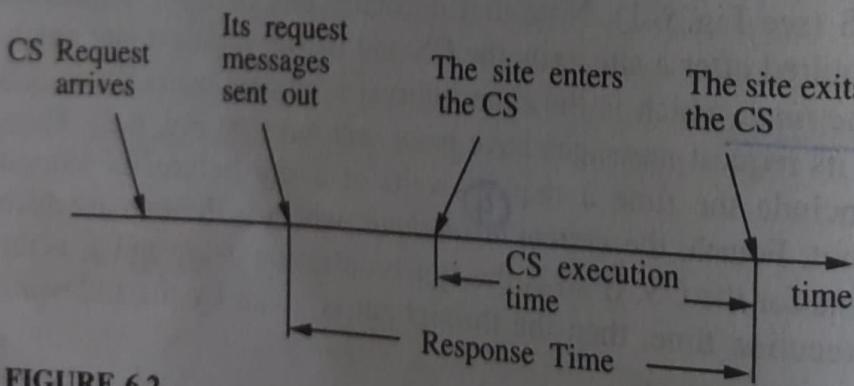


**FIGURE 6.1**  
Synchronization delay.

**LOW AND HIGH LOAD PERFORMANCE.** Performance of a mutual exclusion algorithm depends upon the loading conditions of the system and is often studied under two special loading conditions, viz., *low load* and *high load*. Under low load conditions, there is seldom more than one request for mutual exclusion simultaneously in the system. Under high load conditions, there is always a pending request for mutual exclusion at a site. Thus, after having executed a request, a site immediately initiates activities to let the next site execute its CS. A site is seldom in an idle state under high load conditions. For many mutual exclusion algorithms, the performance metrics can be easily determined under low and high loads through simple reasoning.

**BEST AND WORST CASE PERFORMANCE.** Generally, mutual exclusion algorithms have best and worst cases for the performance metrics. In the best case, prevailing conditions are such that a performance metric attains the best possible value. For example, in most algorithms the best value of the response time is a round-trip message delay plus CS execution time,  $2T + E$  (where  $T$  is the average message delay and  $E$  is the average critical section execution time).

Often for mutual exclusion algorithms, the best and worst cases coincide with low and high loads, respectively. For example, the best and worst values of the response time are achieved when the load is, respectively, low and high. The best and the worse message traffic is generated in Maekawa's algorithm [10] at low and high load conditions, respectively. When the value of a performance metric fluctuates statistically, we generally talk about the average value of that metric.



**FIGURE 6.2**  
Response time.

## 6.4 A SIMPLE SOLUTION TO DISTRIBUTED MUTUAL EXCLUSION

In a simple solution to distributed mutual exclusion, a site, called the control site, is assigned the task of granting permission for the CS execution. To request the CS, a site sends a REQUEST message to the control site. The control site queues up the requests for the CS and grants them permission, one by one. This method to achieve mutual exclusion in distributed systems requires only three messages per CS execution.

*Control site*

This naive, centralized solution has several drawbacks. First, there is a single point of failure, the control site. Second, the control site is likely to be swamped with extra work. Also, the communication links near the control site are likely to be congested and become a bottleneck. Third, the synchronization delay of this algorithm is  $2T$  because a site should first release permission to the control site and then the control site should grant permission to the next site to execute the CS. This has serious implications for the system throughput, which is equal to  $1/(2T + E)$  in this algorithm. Note that if the synchronization delay is reduced to  $T$ , the system throughput is almost doubled to  $1/(T + E)$ . We later discuss several mutual exclusion algorithms that reduce the synchronization delay to  $T$  at the cost of higher message traffic.

## 6.5 NON-TOKEN-BASED ALGORITHMS

In non-token-based mutual exclusion algorithms, a site communicates with a set of other sites to arbitrate who should execute the CS next. For a site  $S_i$ , request set  $R_i$  contains ids of all those sites from which site  $S_i$  must acquire permission before entering the CS. Next, we discuss some non-token-based mutual exclusion algorithms which are good representatives of this class.

Non-token-based mutual exclusion algorithms use timestamps to order requests for the CS and to resolve conflicts between simultaneous requests for the CS. In all these algorithms, logical clocks are maintained and updated according to Lamport's scheme [9]. Each request for the CS gets a timestamp, and smaller timestamp requests have priority over larger timestamp requests.

## 6.6 LAMPORT'S ALGORITHM

Lamport was the first to give a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme [9]. In Lamport's algorithm,  $\forall i : 1 \leq i \leq N :: R_i = \{S_1, S_2, \dots, S_N\}$ . Every site  $S_i$  keeps a queue,  $request\_queue_i$ , which contains mutual exclusion requests ordered by their timestamps. This algorithm requires messages to be delivered in the FIFO order between every pair of sites.

### The Algorithm

#### Requesting the critical section.

- When a site  $S_i$  wants to enter the CS, it sends a  $REQUEST(ts_i, i)$  message to all the sites in its request set  $R_i$  and places the request on  $request\_queue_i$ . ( $(ts_i, i)$  is the timestamp of the request.)

2. When a site  $S_j$  receives the REQUEST( $ts_i, i$ ) message from site  $S_i$ , it returns a timestamped REPLY message to  $S_i$  and places site  $S_i$ 's request on  $request\_queue_j$ .

**Executing the critical section.** Site  $S_i$  enters the CS when the two following conditions hold:

- [L1:]  $S_i$  has received a message with timestamp larger than  $(ts_i, i)$  from all other sites.
- [L2:]  $S_i$ 's request is at the top of  $request\_queue_i$ .

### Releasing the critical section.

3. Site  $S_i$ , upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.
4. When a site  $S_j$  receives a RELEASE message from site  $S_i$ , it removes  $S_i$ 's request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. The algorithm executes CS requests in the increasing order of timestamps.

### Correctness

**Theorem 6.1.** Lamport's algorithm achieves mutual exclusion.

**Proof:** The proof is by contradiction. Suppose two sites  $S_i$  and  $S_j$  are executing the CS concurrently. For this to happen, conditions L1 and L2 must hold at both the sites concurrently. This implies that at some instant in time, say  $t$ , both  $S_i$  and  $S_j$  have their own requests at the top of their  $request\_queues$  and condition L1 holds at them. Without a loss of generality, assume that  $S_i$ 's request has a smaller timestamp than the request of  $S_j$ . Due to condition L1 and the FIFO property of the communication channels, it is clear that at instant  $t$ , the request of  $S_i$  must be present in  $request\_queue_j$ , when  $S_j$  was executing its CS. This implies that  $S_j$ 's own request is at the top of its own  $request\_queue$  when a smaller timestamp request,  $S_i$ 's request, is present in the  $request\_queue_j$ —a contradiction! Hence, Lamport's algorithm achieves mutual exclusion.  $\square$

**Example 6.1.** In Fig. 6.3 through Fig. 6.6, we illustrate the operation of Lamport's algorithm. In Fig. 6.3, sites  $S_1$  and  $S_2$  are making requests for the CS and send out REQUEST messages to other sites. The timestamps of the requests are (2, 1) and (1, 2), respectively. In Fig. 6.4,  $S_2$  has received REPLY messages from all the other sites and its request is at the top of its  $request\_queue$ . Consequently, it enters the CS. In Fig. 6.5,  $S_2$  exits and sends RELEASE messages to all other sites. In Fig. 6.6, site  $S_1$  has received REPLY messages from all other sites and its request is at the top of its  $request\_queue$ . Consequently, it enters the CS next.

**PERFORMANCE.** Lamport's algorithm requires  $3(N-1)$  messages per CS invocation:  $(N-1)$  REQUEST,  $(N-1)$  REPLY, and  $(N-1)$  RELEASE messages. Synchronization delay in the algorithm is  $T$ .

**AN OPTIMIZATION.** Lamport's algorithm can be optimized to require between  $3(N-1)$  and  $2(N-1)$  messages per CS execution by suppressing REPLY messages

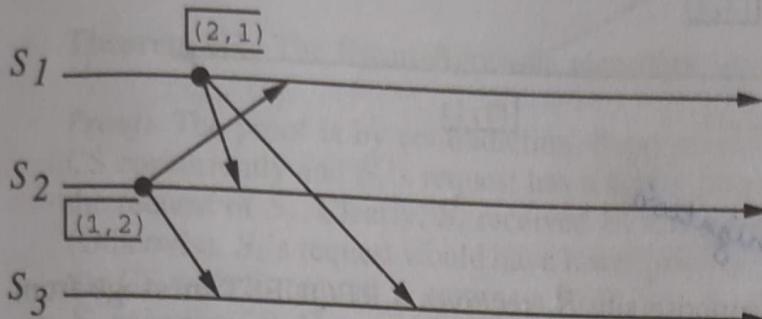


FIGURE 6.3

Sites \$S\_1\$ and \$S\_2\$ are making requests for the CS.

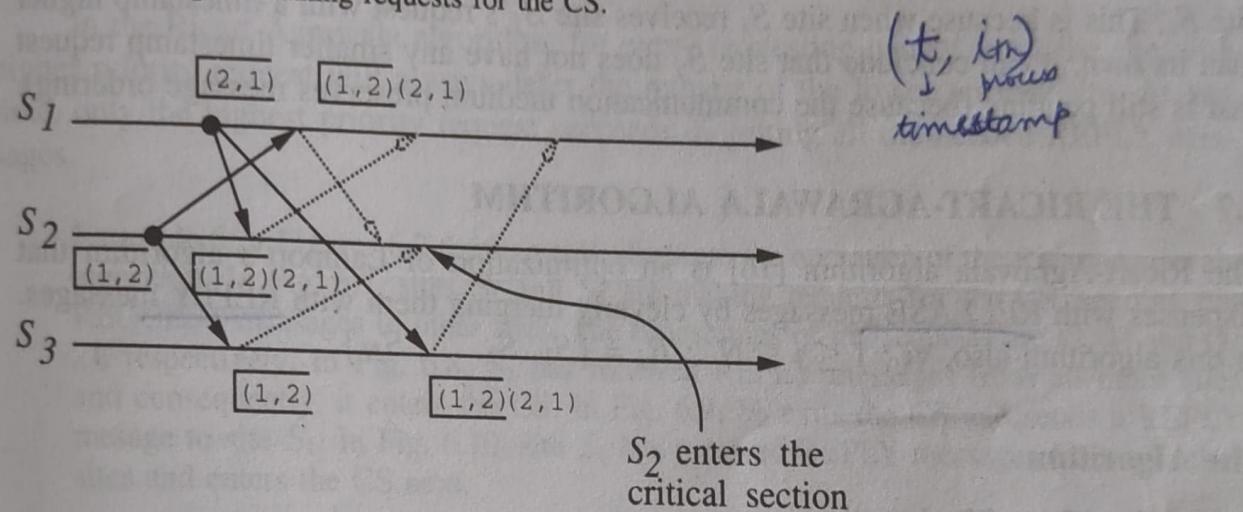


FIGURE 6.4

Site \$S\_2\$ enters the CS.

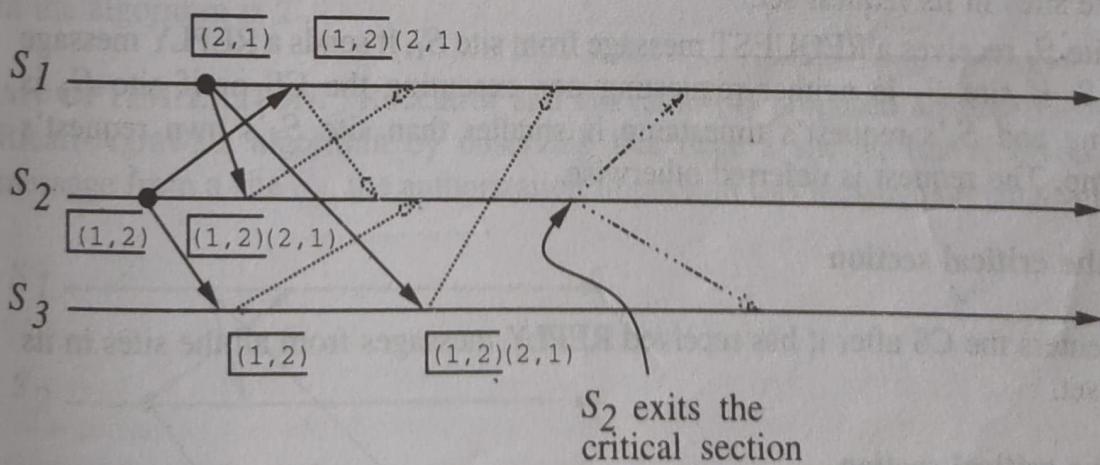
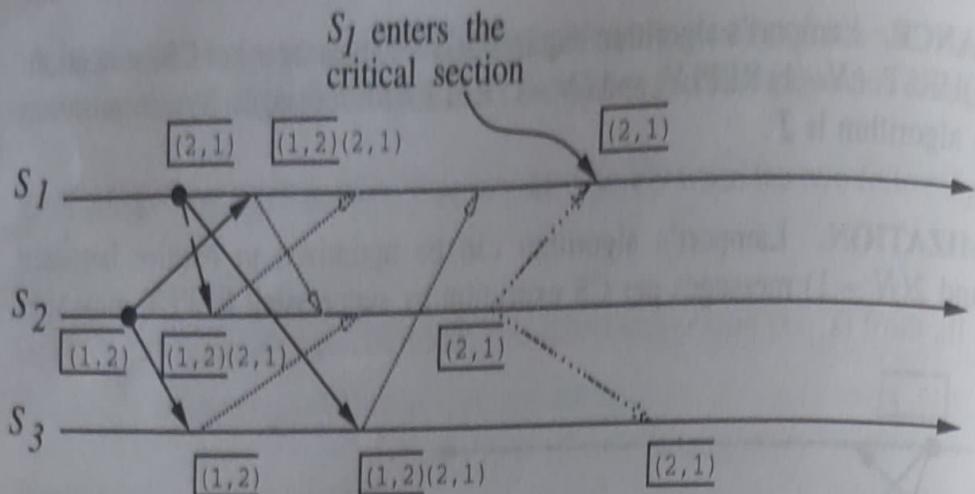


FIGURE 6.5

Site \$S\_2\$ exits the CS and sends RELEASE messages.

**FIGURE 6.6**

Site \$S\_1\$ enters the CS.

*optimization*

in certain situations. For example, suppose site \$S\_j\$ receives a REQUEST message from site \$S\_i\$ after it has sent its own REQUEST message with timestamp higher than the timestamp of site \$S\_i\$'s request. In this case, site \$S\_j\$ need not send a REPLY message to site \$S\_i\$. This is because when site \$S\_i\$ receives site \$S\_j\$'s request with a timestamp higher than its own, it can conclude that site \$S\_j\$ does not have any smaller timestamp request that is still pending (because the communication medium preserves message ordering).

## 6.7 THE RICART-AGRAWALA ALGORITHM

The Ricart-Agrawala algorithm [16] is an optimization of Lamport's algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages. In this algorithm also, \$\forall i : 1 \leq i \leq N :: R\_i = \{S\_1, S\_2, \dots, S\_N\}\$.

### The Algorithm

#### Requesting the critical section.

- When a site \$S\_i\$ wants to enter the CS, it sends a timestamped REQUEST message to all the sites in its request set.
- When site \$S\_j\$ receives a REQUEST message from site \$S\_i\$, it sends a REPLY message to site \$S\_i\$ if site \$S\_j\$ is neither requesting nor executing the CS or if site \$S\_j\$ is requesting and \$S\_i\$'s request's timestamp is smaller than site \$S\_j\$'s own request's timestamp. The request is deferred otherwise.

#### Executing the critical section

- Site \$S\_i\$ enters the CS after it has received REPLY messages from all the sites in its request set.

#### Releasing the critical section

- When site \$S\_i\$ exits the CS, it sends REPLY messages to all the deferred requests.

A site's REPLY messages are blocked only by sites that are requesting the CS with higher priority (i.e., a smaller timestamp). Thus, when a site sends out REPLY messages to all the deferred requests, the site with the next highest priority request receives the last needed REPLY message and enters the CS. The execution of CS requests in this algorithm is always in the order of their timestamps.

## CORRECTNESS

**Theorem 6.2.** The Ricart-Agrawala algorithm achieves mutual exclusion.

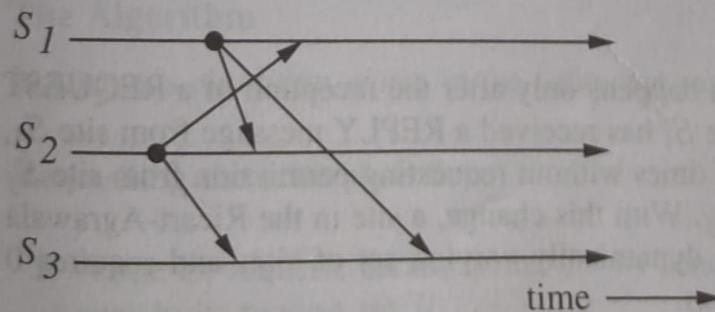
**Proof:** The proof is by contradiction. Suppose two sites  $S_i$  and  $S_j$  are executing the CS concurrently and  $S_i$ 's request has a higher priority (i.e., a smaller timestamp) than the request of  $S_j$ . Clearly,  $S_i$  received  $S_j$ 's request after it had made its own request. (Otherwise,  $S_i$ 's request would have lower priority.) Thus,  $S_j$  can concurrently execute the CS with  $S_i$  only if  $S_i$  returns a REPLY to  $S_j$  (in response to  $S_j$ 's request) before  $S_i$  exits the CS. However, this is impossible because  $S_j$ 's request has lower priority. Therefore, the Ricart-Agrawala algorithm achieves mutual exclusion.  $\square$

In the Ricart-Agrawala algorithm, for every requesting pair of sites, the site with higher priority request will always defer the request of the lower priority site. At any time, only the highest priority request succeeds in getting all the needed REPLY messages.

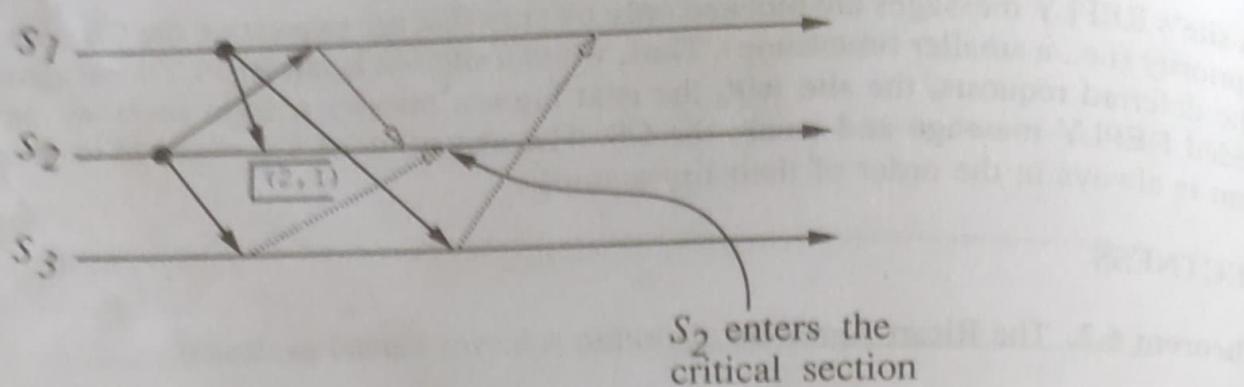
**Example 6.2.** Figures 6.7 through 6.10 illustrate the operation of the Ricart-Agrawala algorithm. In Fig. 6.7, sites  $S_1$  and  $S_2$  are making requests for the CS, sending out REQUEST messages to other sites. The timestamps of the requests are (2, 1) and (1, 2), respectively. In Fig. 6.8,  $S_2$  has received REPLY messages from all other sites and consequently, it enters the CS. In Fig. 6.9,  $S_2$  exits the CS and sends a REPLY message to site  $S_1$ . In Fig. 6.10, site  $S_1$  has received REPLY messages from all other sites and enters the CS next.

**PERFORMANCE.** The Ricart-Agrawala algorithm requires  $2(N - 1)$  messages per CS execution:  $(N - 1)$  REQUEST and  $(N - 1)$  REPLY messages. Synchronization delay in the algorithm is  $T$ .

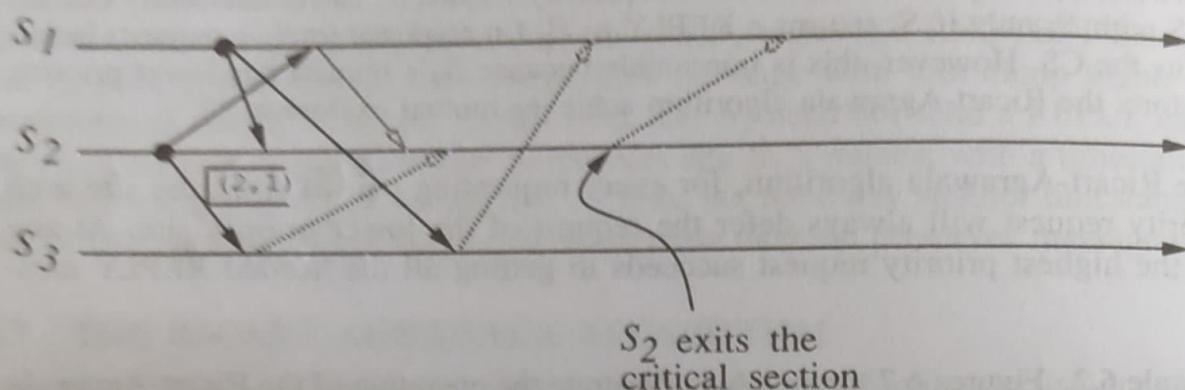
**AN OPTIMIZATION.** Roucair and Carvalho [4] proposed an improvement to the Ricart-Agrawala algorithm by observing that once a site  $S_i$  has received a REPLY message from a site  $S_j$ , the authorization implicit in this message remains valid until  $S_i$



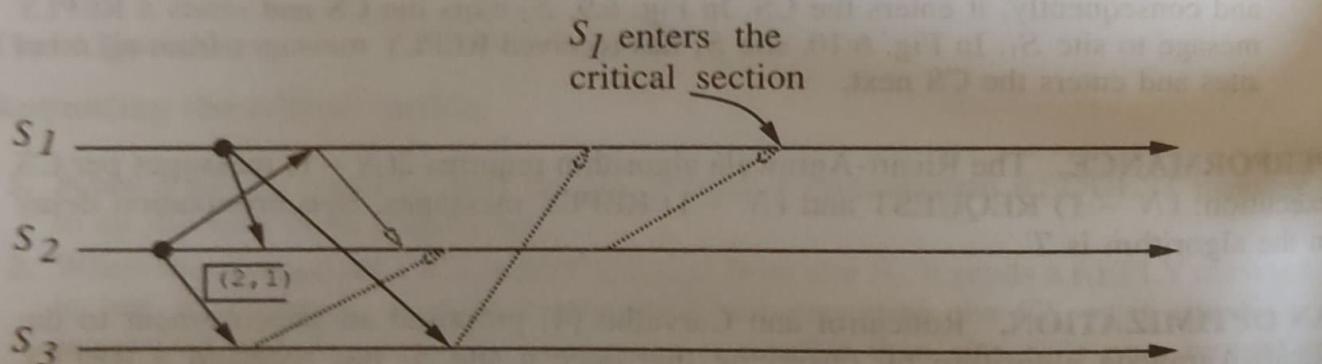
**FIGURE 6.7**  
Sites  $S_1$  and  $S_2$  are making requests for the CS.



**FIGURE 6.8**  
Site \$S\_2\$ enters the CS.



**FIGURE 6.9**  
Site \$S\_2\$ exits the CS and sends a REPLY message to \$S\_1\$.



**FIGURE 6.10**  
Site \$S\_1\$ enters the CS.

sends a REPLY message to \$S\_j\$ (which happens only after the reception of a REQUEST message from \$S\_j\$). Therefore, after site \$S\_i\$ has received a REPLY message from site \$S\_j\$, site \$S\_i\$ can enter its CS any number of times without requesting permission from site \$S\_j\$ until \$S\_i\$ sends a REPLY message to \$S\_j\$. With this change, a site in the Ricart-Agrawala algorithm requests permission from a dynamically varying set of sites and requires 0 to  $2(N - 1)$  messages per CS execution.

is the reason that most mutual exclusion algorithms for distributed systems rely on information structures.

## 6.10 TOKEN-BASED ALGORITHMS

In token-based algorithms, a unique token is shared among all sites. A site is allowed to enter its CS if it possesses the token. Depending upon the way a site carries out its search for the token, there are numerous token-based algorithms. Next, we discuss some representative token-based mutual exclusion algorithms.

Before we start with the discussion of token-based algorithms, two comments are in order: First, token-based algorithms use sequence numbers instead of timestamps. Every request for the token contains a sequence number and the sequence numbers of sites advance independently. A site increments its sequence number counter every time it makes a request for the token. A primary function of the sequence numbers is to distinguish between old and current requests. Second, a correctness proof of token-based algorithms to ensure that mutual exclusion is enforced is trivial because an algorithm guarantees mutual exclusion so long as a site holds the token during the execution of the CS. Rather, the issues of freedom from starvation and freedom from deadlock are prominent.

## 6.11 SUZUKI-KASAMI'S BROADCAST ALGORITHM

In the Suzuki-Kasami's algorithm [21], if a site attempting to enter the CS does not have the token, it broadcasts a REQUEST message for the token to all the other sites. A site that possesses the token sends it to the requesting site upon receiving its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has exited the CS. A site holding the token can enter its CS repeatedly until it sends the token to some other site.

The main design issues in this algorithm are: (1) distinguishing outdated REQUEST messages from current REQUEST messages and (2) determining which site has an outstanding request for the CS.

Outdated REQUEST messages are distinguished from current REQUEST messages in the following manner: A REQUEST message of site  $S_j$  has the form REQUEST( $j, n$ ) where  $n$  ( $n = 1, 2, \dots$ ) is a sequence number that indicates that site  $S_j$  is requesting its  $n^{\text{th}}$  CS execution. A site  $S_i$  keeps an array of integers  $RN_i[1..N]$  where

$RN_i[j]$  is the largest sequence number received so far in a REQUEST message from site  $S_j$ . A REQUEST( $j, n$ ) message received by site  $S_i$  is outdated if  $RN_i[j] > n$ . When site  $S_i$  receives a REQUEST( $j, n$ ) message, it sets  $RN_i[j] := \max(RN_i[j], n)$ .

Sites with outstanding requests for the CS are determined in the following manner: The token consists of a queue of requesting sites,  $Q$ , and an array of integers  $LN[1..N]$  where  $LN[j]$  is the sequence number of the request that site  $S_j$  executed most recently. After executing its CS, a site  $S_i$  updates  $LN[i] := RN_i[i]$  to indicate that its request corresponding to sequence number  $RN_i[i]$  has been executed. The token array  $LN[1..N]$  permits a site to determine if some other site has an outstanding request for the CS. Note that at site  $S_i$ , if  $RN_i[j] = LN[j]+1$ , then site  $S_j$  is currently requesting the token. After having executed the CS, a site checks this condition for all the  $j$ 's to determine all the sites that are requesting the token and places their ids in queue  $Q$  if not already present in this queue  $Q$ . Then the site sends the token to the site at the head of the queue  $Q$ .

## The Algorithm

### Requesting the critical section

1. If the requesting site  $S_i$  does not have the token, then it increments its sequence number,  $RN_i[i]$ , and sends a REQUEST( $i, sn$ ) message to all other sites. ( $sn$  is the updated value of  $RN_i[i]$ .)
2. When a site  $S_j$  receives this message, it sets  $RN_j[i]$  to  $\max(RN_j[i], sn)$ . If  $S_j$  has the idle token, then it sends the token to  $S_i$  if  $RN_j[i] = LN[i]+1$ .

### Executing the critical section.

3. Site  $S_i$  executes the CS when it has received the token.

**Releasing the critical section.** Having finished the execution of the CS, site  $S_i$  takes the following actions:

4. It sets  $LN[i]$  element of the token array equal to  $RN_i[i]$ .
5. For every site  $S_j$ , whose ID is not in the token queue, it appends its ID to the token queue if  $RN_i[j] = LN[j]+1$ .
6. If token queue is nonempty after the above update, then it deletes the top site ID from the queue and sends the token to the site indicated by the ID.

Thus, after having executed its CS, a site gives priority to other sites with outstanding requests for the CS (over its pending requests for the CS). The Suzuki-Kasami algorithm is not symmetric because a site retains the token even if it does not have a request for the CS, which is contrary to the spirit of Ricart and Agrawala's definition of a symmetric algorithm: "*no site possesses the right to access its CS when it has not been requested.*"

## CORRECTNESS

**Theorem 6.5.** A requesting site enters the CS in finite time.

*Proof:* Token request messages of a site  $S_i$  reach other sites in finite time. Since one of these sites will have the token in finite time, site  $S_i$ 's request will be placed in the token queue in finite time. Since there can be at most  $N - 1$  requests in front of this request in the token queue, site  $S_i$  will execute the CS in finite time.  $\square$

**PERFORMANCE.** The beauty of the Suzuki-Kasami algorithm lies in its simplicity and efficiency. The algorithm requires 0 or  $N$  messages per CS invocation. Synchronization delay in this algorithm is 0 or  $T$ . No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request.

Note that not only should the misuse of secret information be prevented, but the destruction of such information should be prevented as well. For example, the destruction of information about customer account balances and bank transactions can have serious socioeconomic ramifications.

In this chapter, we study models of protection and techniques to enforce security and protection in computer systems.

## 14.2 PRELIMINARIES

### 14.2.1 Potential Security Violations

Anderson [2] has classified the potential security violations into three categories:

**Unauthorized information release.** This occurs when an unauthorized person is able to read and take advantage of the information stored in a computer system. This also includes the unauthorized use of a computer program.

**Unauthorized information modification.** This occurs when an unauthorized person is able to alter the information stored in a computer. Examples include changing student grades in a university database and changing account balances in a bank database. Note that an unauthorized person need not read the information before changing it. Blind writes can be performed.

**Unauthorized denial of service.** An unauthorized person should not succeed in preventing an authorized user from accessing the information stored in a computer. Note that services can be denied to authorized users by some internal actions (like crashing the system by some means, overloading the system, changing the scheduling algorithm) and by external actions (such as setting fire or disrupting electrical supply).

### 14.2.2 External vs. Internal Security

Computer systems security can be divided into two parts: external security and internal security. External security, also called physical security, deals with regulating access to the premises of computer systems, which include the physical machine (hardware, disks, tapes, power supply, air conditioning), terminals, computer console, etc. External security can be enforced by placing a guard at the door, by giving a key or secret code to authorized persons, etc.

Internal security deals with the access and use of computer hardware and software information stored in the computer system. Aside from external and internal securities, there is an issue of *authentication* by which a user “logs into” the computer system to access the hardware and the software resources.

Clearly, issues involved in external security are simple and administrative in nature. In this chapter, we will mainly be concerned with the internal security in computer systems, which is more challenging and subtle.

### 14.2.3 Policies and Mechanisms

Recall from Chap. 1 that policies refer to what should be done and mechanisms refer to how it should be done. A protection mechanism provides a set of tools that can be

used to design or specify a wide array of protection policies, whereas a policy gives assignment of the access rights of users to various resources. The separation of policies and mechanisms enhances design flexibility.

Protection in an operating system refers to mechanisms that control user access to system resources, whereas policies decide which user can have access to what resources. Policies can change with time and applications. Thus, a protection scheme must be amenable to a wide variety of policies to enforce security in computer systems. In this chapter, we will mainly be concerned with the design of protection mechanisms in operating systems.

**PROTECTION VS. SECURITY.** Hydra [39] designers make a distinction between protection and security. According to them, *protection is a mechanism and security is a policy*. Protection deals with mechanisms to build secure systems and security deals with policy issues that use protection mechanisms to build secure systems.

#### 14.2.4 Protection Domain

The protection domain of a process specifies the resources that it can access and the types of operations that the process can perform on the resources. In a typical computation, the control moves through a series of processes. To enforce security in the system, it is good policy to allow a process to access only those resources that it requires to complete its task. This eliminates the possibility of a process breaching security maliciously or unintentionally (such as by a software bug) and increases accountability.

The concept of protection domain of a process enables us to achieve the policy of limiting a process's access to only needed resources. Every process executes in its protection domain and protection domain is switched appropriately whenever control jumps from a process to another process.

#### 14.2.5 Design Principles for Secure Systems

Saltzer and Schroeder [34] gave the following principles for designing a secure computer system:

**Economy.** A protection mechanism should be economical to develop and use. Its inclusion in a system should not result in substantial cost or overhead to the system. One easy way to achieve economy is to keep the design as simple and small as possible [34].

**Complete Mediation.** The design of a completely secure system requires that every request to access an object be checked for the authority to do so.

**Open Design.** A protection mechanism should not stake its integrity on the ignorance of potential attackers concerning the protection mechanism itself (i.e., the underlying principle used to achieve the security). A protection mechanism should work even if its underlying principles are known to an attacker.

**Separation of Privileges.** A protection mechanism that requires two keys to unlock a lock (or gain access to a protected object) is more robust and flexible than one

that allows only a single key to unlock a lock. In computer systems, the presence of two keys may mean satisfying two independent conditions before an access is allowed.

**Least Privilege.** A subject should be given the bare minimum access rights that are sufficient for it to complete its task. If the requirement of a subject changes, the subject should acquire it by switching the domain. (Recall that a domain defines access rights of a subject to various objects.)

**Least Common Mechanism.** According to this principle, the portion of a mechanism that is common to more than one user should be minimized, as any coupling among users (through shared mechanisms and variables) represents a potential information path between users and is thus a potential threat to their security.

**Acceptability.** A protection mechanism must be simple to use. A complex and obscure protection mechanism will deter users from using it.

**Fail-Safe Defaults.** Default case should mean lack of access (because it is safer this way). If a design or implementation mistake is responsible for denial of an access, it will eventually be discovered and be fixed. However, the opposite is not true.

### 14.3 THE ACCESS MATRIX MODEL

A model of protection abstracts the essential features of a protection system so that various properties of it can be proven. A protection system consists of mechanisms to control user access to system resources or to control information flow in the system. In this section, we study the most fundamental model of protection—the access matrix model—in computer systems. Advanced models of protection are covered in Sec. 14.6. A survey of models for protection in computer systems can be found in a paper by Landwehr [23].

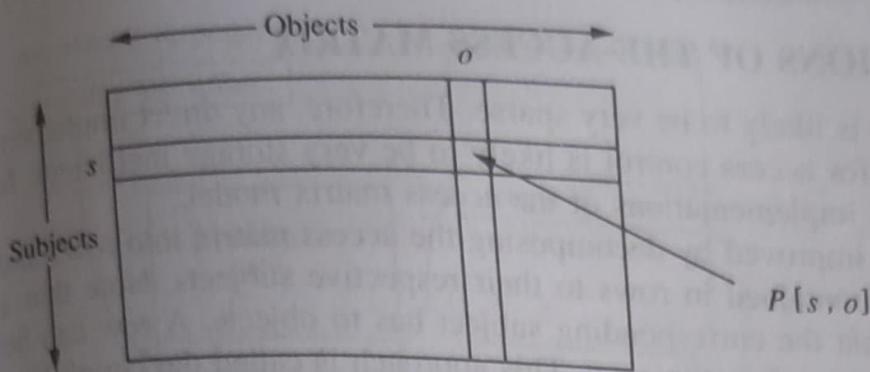
The access matrix model was first proposed by Lampson [21]. It was further enhanced and refined by Graham and Denning [18] and Harrison et al. [19]. The description of the access matrix model in this section is based on the work of Harrison et al. [19]. This model consists of the following three components:

**Current Objects.** Current objects are a finite set of entities to which access is to be controlled. The set is denoted by ' $O$ '. A typical example of an object is a file.

**Current Subjects.** Current subjects are a finite set of entities that access current objects. The set is denoted by ' $S$ '. A typical example of a subject is a process. Note that  $S \subseteq O$ . That is, subjects can be treated as objects and can be accessed like an object by other subjects.

**Generic Rights.** A finite set of generic rights,  $R = \{r_1, r_2, r_3, \dots, r_m\}$ , gives various access rights that subjects can have to objects. Typical examples of such rights are read, write, execute, own, delete, etc.

**THE PROTECTION STATE OF A SYSTEM.** The *protection state* of a system is represented by a triplet  $(S, O, P)$ , where  $S$  is the set of current subjects,  $O$  is the set of current objects, and  $P$  is a matrix, called the *access matrix*, with a row for every current subject and a column for every current object. A schematic diagram of an access matrix



**FIGURE 14.1**  
A schematic of an access matrix.

is shown in Fig. 14.1. Note that the access matrix  $P$  itself is a protected object. Let variables  $s$  and  $o$  denote a subject and an object, respectively. Entry  $P[s, o]$  is a subset of  $R$ , the generic rights, and denotes the access rights which subject  $s$  has to object  $o$ .

**ENFORCING A SECURITY POLICY.** A security policy is enforced by validating every user access for appropriate access rights. Every object has a monitor that validates all accesses to that object in the following manner.

1. A subject  $s$  requests an access  $\alpha$  to object  $o$ .
2. The protection system presents triplet  $(s, \alpha, o)$  to the monitor of  $o$ .
3. The monitor looks into the access rights of  $s$  to  $o$ . If  $\alpha \in P[s, o]$ , then the access is permitted; Else it is denied.

**Example 14.1.** Figure 14.2 illustrates an access matrix that represents the protection state of a system with three subjects,  $s_1, s_2, s_3$ , and five objects,  $o_1, o_2, s_1, s_2, s_3$ . In this protection state, subject  $s_1$  can read and write object  $o_1$ , delete  $o_2$ , send mail to  $s_2$ , and receive mail from  $s_3$ . Subject  $s_3$  owns  $o_1$  and can read and write  $o_2$ .

The access matrix model of a protection system is very popular because of its simplicity, elegant structure, and amenability to various implementations. We next discuss implementations of the access matrix model.

	$o_1$	$o_2$	$s_1$	$s_2$	$s_3$
$s_1$	read, write	own, delete	own	sendmail	recmail
$s_2$	execute	copy	recmail	own	block, wakeup
$s_3$	own	read, write	sendmail	block, wakeup	own

**FIGURE 14.2**  
An access matrix representing a protection state.

## 14.4 IMPLEMENTATIONS OF THE ACCESS MATRIX

Note that the access matrix is likely to be very sparse. Therefore, any direct implementation of the access matrix for access control is likely to be very storage inefficient. In this section, we study three implementations of the access matrix model.

The efficiency can be improved by decomposing the access matrix into rows and assigning the access rights contained in rows to their respective subjects. Note that a row denotes access rights that the corresponding subject has to objects. A row can be collapsed by deleting null entries for efficiency. This approach is called the *capability-based* method. An orthogonal approach is to decompose the access control matrix by columns and assign the columns to their respective objects. Note that a column denotes access rights of various subjects to the object. A column can be collapsed by deleting null entries for higher efficiency. This technique is called the *access control list* method. The third approach, called the *lock-key* method, is a combination of the first two approaches.

### 14.4.1 Capabilities

The capability based method corresponds to the row-wise decomposition of the access matrix. Each subject  $s$  is assigned a list of tuples  $(o, P[s, o])$  for all objects  $o$  that it is allowed to access. The tuples are referred to as *capabilities*. If subject  $s$  possesses a capability  $(o, P[s, o])$ , then it is authorized to access object  $o$  in manners specified in  $P[s, o]$ . Possession of a capability by a user is treated as *prima facie* evidence that the user has authority to access the object in the ways specified in the capability. The list of capabilities assigned to subject  $s$  corresponds to access rights contained in the row for subject  $s$  in the access matrix. At any time, a subject is authorized to access only those objects for which it has capabilities. Clearly, one must not be able to forge capabilities.

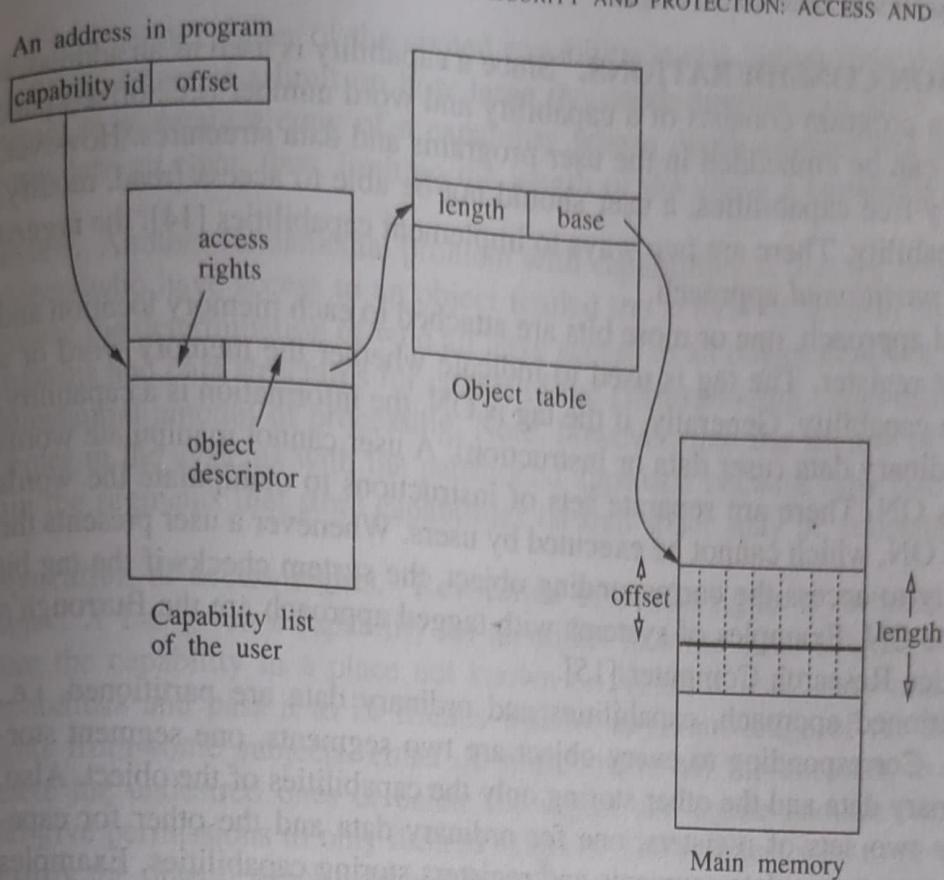
A schematic view of a capability is shown in Fig. 14.3. A capability has two fields. First, an object descriptor, which is an identifier for the object and second, access rights, which indicate the allowed access rights to the object. The object descriptor can very well be the address of the corresponding objects and therefore, aside from providing protection, capabilities can also be used as an addressing mechanism by the system. The main advantage of using a capability as an addressing mechanism is that it provides an address that is context independent. That is, it provides an absolute address [14]. However, when a capability is used as an addressing mechanism, the system must allow the embedding of capabilities in user programs and data structures, as a capability will be a part of the address.

**CAPABILITY-BASED ADDRESSING.** Capability-based addressing is illustrated in Fig. 14.4. A user program issues a request to access a word within an object. The

Object descriptor	Access rights
	read, write, execute, etc.

FIGURE 14.3

A schematic view of a capability.

**FIGURE 14.4**

An illustration of capability-based addressing.

address of the request contains the capability ID of the object (which tells what object in the main memory is to be accessed) and an offset within the object (which gives the relative location of the word in the object to be accessed). The system uses the capability ID to search the capability list of the user to locate the capability that contains the allowed access rights and an object descriptor. The system checks if the requested access is permitted by checking the access rights in the capability. The object descriptor is used to search the object table to locate the entry for the object. The entry consists of the base address of the object in main memory and the length of the object. The system adds the base address to the offset in the request to determine the exact memory location of the accessed word.

Capability-based addressing has two salient features, relocatability and sharing. An object can be relocated anywhere in the main memory without making any change to the capabilities that refer to it. (For every relocation, only the base field of the object needs be changed in the object table.) Sharing is made easy as several programs can share the same object (program or data) with different names (object descriptors) for the object. Note that this type of sharing and relocatability is achieved by introducing a level of indirection (via the object table) in addressing the objects—the object descriptor in a capability contains the address of the object.

If there is a separate object table for each process or subject, then the resolution of an object descriptor is done in the context of a process. If there is a global object table, then the resolution of an object descriptor is done in a single global context.