CHAPTER 18

DISTRIBUTED PROCESS MANAGEMENT

We cannot enter into alliance with neighboring princes until we are acquainted with their designs.

—*The Art of War,* Sun Tzu

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Give an explanation of process migration.
- Understand the concept of distributed global states.
- Analyze distributed mutual exclusion algorithms.
- Analyze distributed deadlock algorithms.

This chapter examines key mechanisms used in distributed operating systems. First we look at process migration, which is the movement of an active process from one machine to another. Next, we examine the question of how processes on different systems can coordinate their activities when each is governed by a local clock and when there is a delay in the exchange of information. Finally, we explore two key issues in distributed process management: mutual exclusion and deadlock.

18.1 PROCESS MIGRATION

Process migration is the transfer of a sufficient amount of the state of a process from one computer to another for the process to execute on the target machine. Interest in this concept grew out of research into methods of load balancing across multiple networked systems, although the application of the concept now extends beyond that one area.

In the past, only a few of the many papers on load distribution were based on true implementations of process migration, which includes the ability to preempt a process on one machine and reactivate it later on

another machine. Experience showed that preemptive process migration is possible, although with higher overhead and complexity than originally anticipated [ARTS89a]. This cost led some observers to conclude that process migration was not practical. Such assessments have proved too pessimistic. New implementations, including those in commercial products, have fueled a continuing interest and new developments in this area. This section provides an overview.

Motivation

Process migration is desirable in distributed systems for a number of reasons [SMIT88, JUL88], including:

- **Load sharing:** By moving processes from heavily loaded to lightly loaded systems, the load can be balanced to improve overall performance. Empirical data suggest that significant performance improvements are possible [LELA86, CABR86]. However, care must be taken in the design of load-balancing algorithms. [EAGE86] points out that the more communication necessary for the distributed system to perform the balancing, the worse the performance becomes. A discussion of this issue, with references to other studies, can be found in [ESKI90].
- **Communications performance:** Processes that interact intensively can be moved to the same node to reduce communications cost for the duration of their interaction. Also, when a process is performing data analysis on some file or set of files larger than the process's size, it may be advantageous to move the process to the data rather than vice versa.
- **Availability:** Long-running processes may need to move to survive in the face of faults for which advance notice is possible or in advance of

scheduled downtime. If the operating system provides such notification, a process that wants to continue can either migrate to another system or ensure that it can be restarted on the current system at some later time.

- **Utilizing special capabilities:** A process can move to take advantage of unique hardware or software capabilities on a particular node.

Process Migration Mechanisms

A number of issues need to be addressed in designing a process migration facility. Among these are the following:

- Who initiates the migration?
- What portion of the process is migrated?
- What happens to outstanding messages and signals?

Initiation of Migration

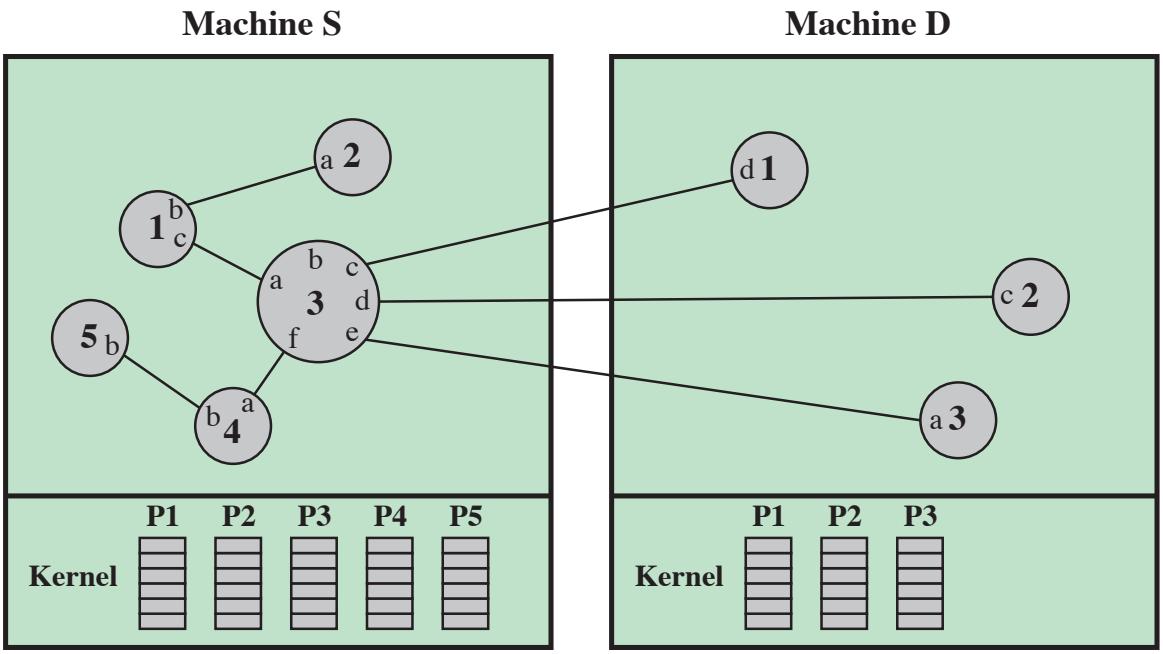
Who initiates migration will depend on the goal of the migration facility. If the goal is load balancing, then some module in the operating system that is monitoring system load will generally be responsible for deciding when migration should take place. The module will be responsible for preempting or signaling a process to be migrated. To determine where to migrate, the module will need to be in communication with peer modules in other systems so that the load patterns on other systems can be monitored. If the goal is to reach particular resources, then a process may migrate itself as the need arises. In this latter case, the process must be aware of the existence of a distributed system. In the former case, the entire migration function, and indeed the existence of multiple systems, may be transparent to the process.

What Is Migrated?

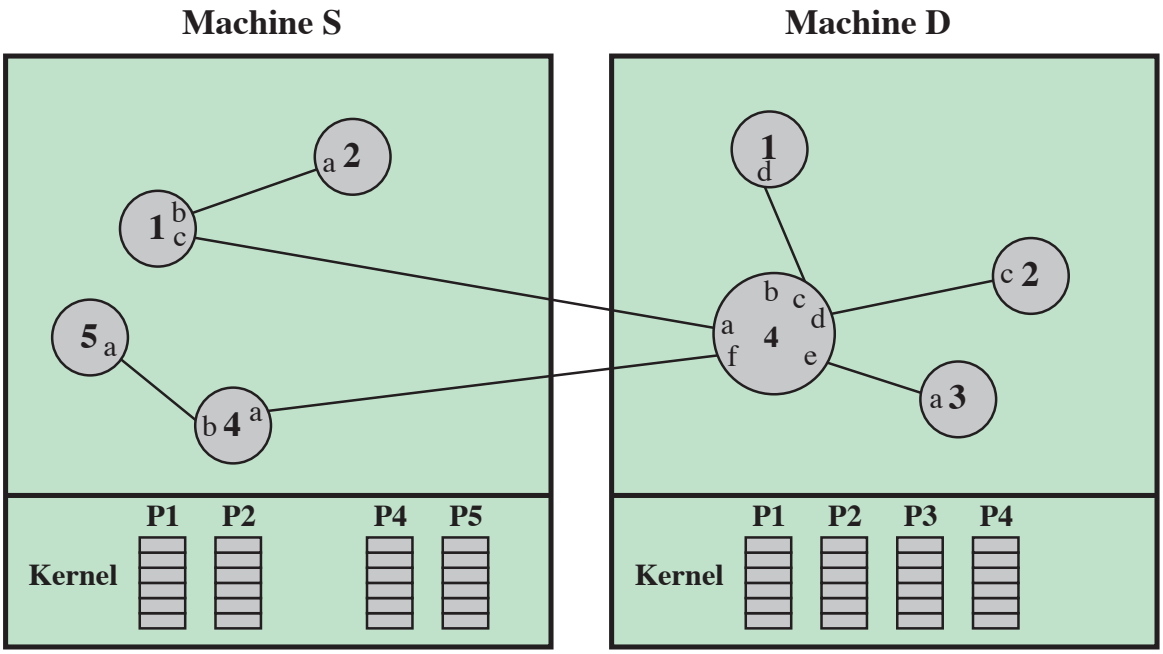
When a process is migrated, it is necessary to destroy the process on the source system and create it on the target system. This is a movement of a process, not a replication. Thus, the process image, consisting of at least the process control block, must be moved. In addition, any links between this process and other processes, such as for passing messages and signals, must be updated. Figure 18.1 illustrates these considerations. Process 3 has migrated out of machine S to become Process 4 in machine D. All link identifiers held by processes (denoted in lowercase letters) remain the same as before. It is the responsibility of the operating system to move the process control block and to update link mappings. The transfer of the process of one machine to another is invisible to both the migrated process and its communication partners.

The movement of the process control block is straightforward. The difficulty, from a performance point of view, concerns the process address space and any open files assigned to the process. Consider first the process address space and let us assume that a virtual memory scheme (paging or paging/segmentation) is being used. The following strategies have been considered [MILO00]:

- **Eager (all):** Transfer the entire address space at the time of migration. This is certainly the cleanest approach. No trace of the process need be left behind at the old system. However, if the address space is very large and if the process is likely not to need most of it, then this may be unnecessarily expensive. Initial costs of migration may be on the order of minutes. Implementations that provide a checkpoint/restart facility are likely to use this approach, because it is simpler to do the checkpointing and restarting if all of the address space is localized.



(a) Before migration



(b) After migration

Figure 18.1 Example of Process Migration

- **Precopy:** The process continues to execute on the source node while the address space is copied to the target node. Pages modified on the source during the precopy operation have to be copied a second time. This strategy reduces the time that a process is frozen and cannot execute during migration.
- **Eager (dirty):** Transfer only those pages of the address space that are in main memory and have been modified. Any additional blocks of the virtual address space will be transferred on demand only. This minimizes the amount of data that are transferred. It does require, however, that the source machine continue to be involved in the life of the process by maintaining page and/or segment table entries and it requires remote paging support.
- **Copy-on-reference:** This is a variation of eager (dirty) in which pages are only brought over when referenced. This has the lowest initial cost of process migration, ranging from a few tens to a few hundreds of microseconds.
- **Flushing:** The pages of the process are cleared from the main memory of the source by flushing dirty pages to disk. Then pages are accessed as needed from disk instead of from memory on the source node. This strategy relieves the source of the need to hold any pages of the migrated process in main memory, immediately freeing a block of memory to be used for other processes.

If it is likely that the process will not use much of its address space while on the target machine (e.g., the process is only temporarily going to another machine to work on a file and will soon return), then one of the last three strategies makes sense. On the other hand, if much of the address space will eventually be accessed while on the target machine, then the piecemeal transfer of blocks of the address space may be less efficient than

simply transferring all of the address space at the time of migration, using one of the first two strategies.

In many cases, it may not be possible to know in advance whether or not much of the nonresident address space will be needed. However, if processes are structured as threads, and if the basic unit of migration is the thread rather than the process, then a strategy based on remote paging would seem to be the best. Indeed, such a strategy is almost mandated, because the remaining threads of the process are left behind and also need access to the address space of the process. Thread migration is implemented in the Emerald operating system [JUL89].

Similar considerations apply to the movement of open files. If the file is initially on the same system as the process to be migrated and if the file is locked for exclusive access by that process, then it may make sense to transfer the file with the process. The danger here is that the process may only be gone temporarily and may not need the file until its return. Therefore, it may make sense to transfer the entire file only after an access request is made by the migrated process. If a file is shared by multiple distributed processes, then distributed access to the file should be maintained without moving the file.

If caching is permitted, as in the Sprite system (Figure 16.7), then an additional complexity is introduced. For example, if a process has a file open for writing and it forks and migrates a child, the file would then be open for writing on two different hosts; Sprite's cache consistency algorithm dictates that the file be made noncacheable on the machines on which the two processes are executing [DOUG89, DOUG91].

MESSAGES AND SIGNALS

The final issue listed previously, the fate of messages and signals, is addressed by providing a mechanism for temporarily storing outstanding

messages and signals during the migration activity and then directing them to the new destination. It may be necessary to maintain forwarding information at the initial site for some time to assure that all outstanding messages and signals get through.

A MIGRATION SCENARIO

As a representative example of self-migration, let us consider the facility available on IBM's AIX operating system [WALK89], which is a distributed UNIX operating system. A similar facility is available on the LOCUS operating system [POPE85], and in fact the AIX system is based on the LOCUS development. This facility has also been ported to the OSF/1 AD operating system, under the name TNC [ZAJC93].

The following sequence of events occurs:

1. When a process decides to migrate itself, it selects a target machine and sends a remote tasking message. The message carries a part of the process image and open file information.
2. At the receiving site, a kernel server process forks a child, giving it this information.
3. The new process pulls over data, environment, arguments, or stack information as needed to complete its operation. Program text is copied over if it is dirty or demand paged from the global file system if it is clean.
4. The originating process is signaled on the completion of the migration. This process sends a final done message to the new process and destroys itself.

A similar sequence would be followed when another process initiates the migration. The principal difference is that the process to be migrated must

be suspended so that it can be migrated in a nonrunning state. This procedure is followed in Sprite, for example [DOUG89].

In the foregoing scenario, migration is a dynamic activity involving a number of steps for moving the process image over. When migration is initiated by another process, rather than self-migration, another approach is to copy the process image and its entire address space into a file, destroy the process, copy the file to another machine using a file transfer facility, and then re-create the process from the file on the target machine. [SMIT89] describes such an approach.

Negotiation of Migration

Another aspect of process migration relates to the decision about migration. In some cases, the decision is made by a single entity. For example, if load balancing is the goal, a load-balancing module monitors the relative load on various machines and performs migration as necessary to maintain a load balance. If self-migration is used to allow a process access to special facilities or to large remote files, then the process itself may make the decision. However, some systems allow the designated target system to participate in the decision. One reason for this could be to preserve response time for users. A user at a workstation, for example, might suffer noticeable response time degradation if processes migrate to the user's system, even if such migration served to provide better overall balance.

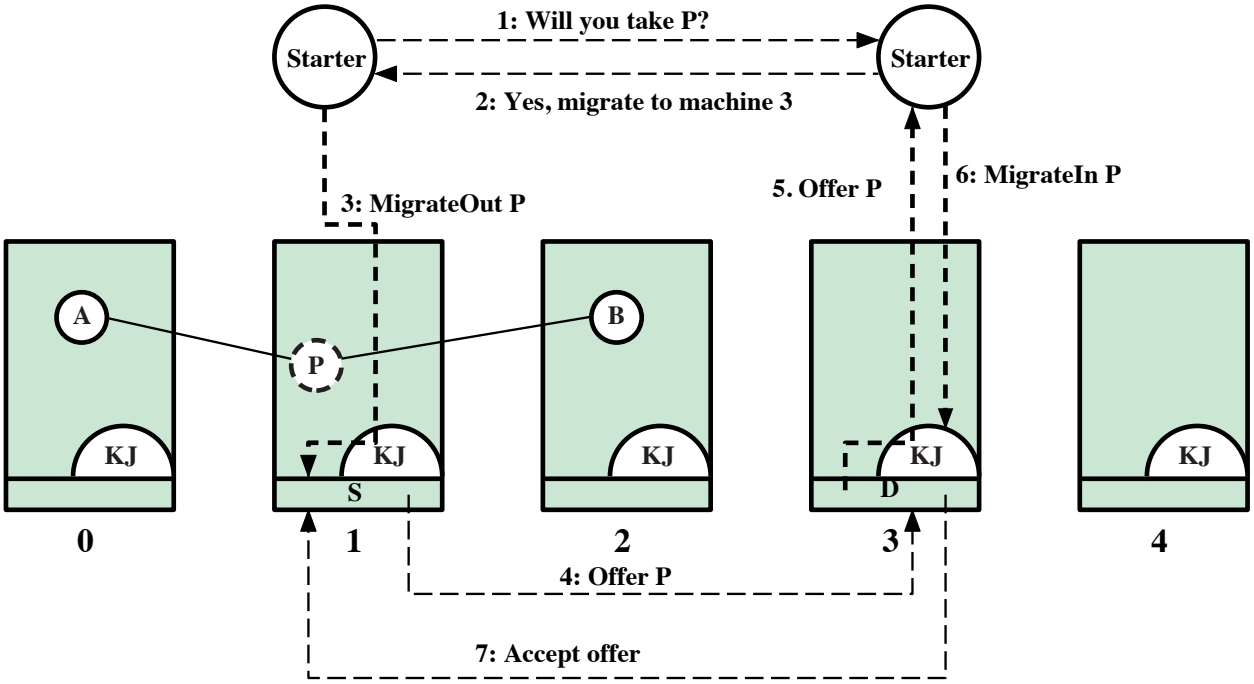


Figure 18.2 Negotiation of Process Migration

An example of a negotiation mechanism is that found in Charlotte [FINK89, ARTS89b]. Migration policy (when to migrate which process to what destination) is the responsibility of the Starter utility, which is a process that is also responsible for long-term scheduling and memory allocation. The Starter can therefore coordinate policy in these three areas. Each Starter process may control a cluster of machines. The Starter receives timely and fairly elaborate load statistics from the kernel of each of its machines.

The decision to migrate must be reached jointly by two Starter processes (one on the source node and one on the destination node), as illustrated in Figure 18.2. The following steps occur:

1. The Starter that controls the source system (S) decides that a process P should be migrated to a particular destination system (D). It sends a message to D's Starter, requesting the transfer.

2. If D's Starter is prepared to receive the process, it sends back a positive acknowledgment.
3. S's Starter communicates this decision to S's kernel via service call (if the starter runs on S) or a message to the KernJob (KJ) of machine S (if the starter runs on another machine). KJ is a process used to convert messages from remote processes into service calls.
4. The kernel on S then offers to send the process to D. The offer includes statistics about P, such as its age and processor and communication loads.
5. If D is short of resources, it may reject the offer. Otherwise, the kernel on D relays the offer to its controlling Starter. The relay includes the same information as the offer from S.
6. The Starter's policy decision is communicated to D by a MigrateIn call.
7. D reserves necessary resources to avoid deadlock and flow-control problems and then sends an acceptance to S.

Figure 18.2 also shows two other processes, A and B, that have links open to P. Following the foregoing steps, machine 1, where S resides, must send a link update message to both machines 0 and 2 to preserve the links from A and B to P. Link update messages tell the new address of each link held by P and are acknowledged by the notified kernels for synchronization purposes. After this point, a message sent to P on any of its links will be sent directly to D. These messages can be exchanged concurrently with the steps just described. Finally, after step 7 and after all links have been updated, S collects all of P's context into a single message and sends it to D.

Machine 4 is also running Charlotte but is not involved in this migration and therefore has no communication with the other systems in this episode.

Eviction

The negotiation mechanism allows a destination system to refuse to accept the migration of a process to itself. In addition, it might also be useful to allow a system to evict a process that has been migrated to it. For example, if a workstation is idle, one or more processes may be migrated to it. Once the user of that workstation becomes active, it may be necessary to evict the migrated processes to provide adequate response time.

An example of an eviction capability is that found in Sprite [DOUG89]. In Sprite, which is a workstation operating system, each process appears to run on a single host throughout its lifetime. This host is known as the home node of the process. If a process is migrated, it becomes a foreign process on the destination machine. At any time the destination machine may evict the foreign process, which is then forced to migrate back to its home node.

The elements of the Sprite eviction mechanism are as follows:

1. A monitor process at each node keeps track of current load to determine when to accept new foreign processes. If the monitor detects activity at the workstation's console, it initiates an eviction procedure on each foreign process.
2. If a process is evicted, it is migrated back to its home node. The process may be migrated again if another node is available.
3. Although it may take some time to evict all processes, all processes marked for eviction are immediately suspended. Permitting an evicted process to execute while it is waiting for eviction would reduce the time during which the process is frozen but also reduce the processing power available to the host while evictions are underway.
4. The entire address space of an evicted process is transferred to the home node. The time to evict a process and migrate it back to its home node may be reduced substantially by retrieving the memory image of an evicted process from its previous foreign host as

referenced. However, this compels the foreign host to dedicate resources and honor service requests from the evicted process for a longer period of time than necessary.

Preemptive versus Nonpreemptive Transfers

The discussion in this section has dealt with preemptive process migration, which involves transferring a partially executed process, or at least a process whose creation has been completed. A simpler function is nonpreemptive process transfer, which involves only processes that have not begun execution and hence do not require transferring the state of the process. In both types of transfer, information about the environment in which the process will execute must be transferred to the remote node. This may include the user's current working directory, the privileges inherited by the process, and inherited resources such as file descriptions.

Nonpreemptive process migration can be useful in load balancing (e.g., see [SHIV92]). It has the advantage that it avoids the overhead of full-blown process migration. The disadvantage is that such a scheme does not react well to sudden changes in load distribution.

18.2 DISTRIBUTED GLOBAL STATES

Global States and Distributed Snapshots

All of the concurrency issues that are faced in a tightly coupled system, such as mutual exclusion, deadlock, and starvation, are also faced in a distributed system. Design strategies in these areas are complicated by the fact that there is no global state to the system. That is, it is not possible for the operating system, or any process, to know the current state of all processes in the distributed system. A process can only know the current state of all the processes on the local system, by access to process control blocks in

memory. For remote processes, a process can only know state information that is received via messages, which represent the state of the remote process sometime in the past. This is analogous to the situation in astronomy: Our knowledge of a distant star or galaxy consists of light and other electromagnetic waves arriving from the distant object, and these waves provide a picture of the object sometime in the past. For example, our knowledge of an object at a distance of five light-years is five years old.

The time lags imposed by the nature of distributed systems complicate all issues relating to concurrency. To illustrate this, we present an example taken from [ANDR90]. We will use process/event graphs (Figures 18.3 and 18.4) to illustrate the problem. In these graphs, there is a horizontal line for each process representing the time axis. A point on the line corresponds to an event (e.g., internal process event, message send, message receive). A box surrounding a point represents a snapshot of the local process state taken at that point. An arrow represents a message between two processes.

In our example, an individual has a bank account distributed over two branches of a bank. To determine the total amount in the customer's account, the bank must determine the amount in each branch. Suppose that the determination is to be made at exactly 3:00 p.m. Figure 18.3a shows an instance in which a balance of $100.00 in the combined account is found. But the situation in Figure 18.3b is also possible. Here, the balance from branch A is in transit to branch B at the time of observation; the result is a false reading of $0.00. This particular problem can be solved by examining all messages in transit at the time of observation. Branch A will keep a record of all transfers out of the account, together with the identity of the destination of the transfer. Therefore, we will include in the "state" of a branch A account both the current balance and a record of transfers. When the two accounts are examined, the observer finds a transfer that has left branch A destined for the customer's account in branch B. Because the

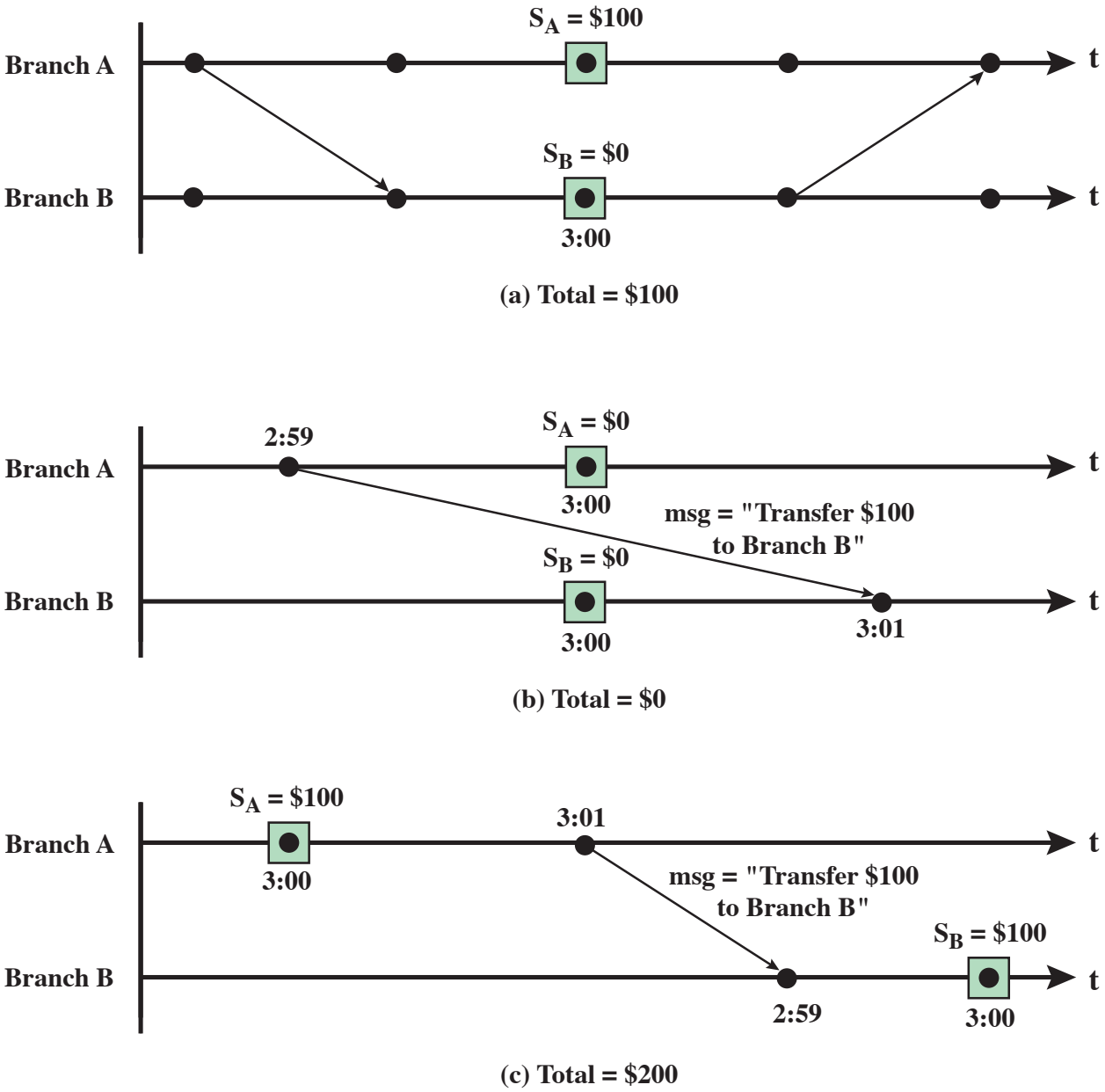


Figure 18.3 Example of Determining Global States

amount has not yet arrived at branch B, it is added into the total balance. Any amount that has been both transferred and received is counted only once, as part of the balance at the receiving account.

This strategy is not foolproof, as shown in Figure 18.3c. In this example, the clocks at the two branches are not perfectly synchronized. The state of

the customer account at branch A at 3:00 p.m. indicates a balance of $100.00. However, this amount is subsequently transferred to branch B at 3:01 according to the clock at A but arrives at B at 2:59 according to B's clock. Therefore, the amount is counted twice for a 3:00 observation.

To understand the difficulty we face and to formulate a solution, let us define the following terms:

- **Channel:** A channel exists between two processes if they exchange messages. We can think of the channel as the path or means by which the messages are transferred. For convenience, channels are viewed as unidirectional. Thus, if two processes exchange messages, two channels are required, one for each direction of message transfer.
- **State:** The state of a process is the sequence of messages that have been sent and received along channels incident with the process.
- **Snapshot:** A snapshot records the state of a process. Each snapshot includes a record of all messages sent and received on all channels since the last snapshot.
- **Global state:** The combined state of all processes.
- **Distributed snapshot:** A collection of snapshots, one for each process.

The problem is that a true global state cannot be determined because of the time lapse associated with message transfer. We can attempt to define a global state by collecting snapshots from all processes. For example, the global state of Figure 18.4a at the time of the taking of snapshots shows a message in transit on the < A,B > channel, one in transit on the < A,C > channel, and one in transit on the < C,A > channel. Messages 2 and 4 are represented appropriately, but message 3 is not. The distributed snapshot indicates that this message has been received but not yet sent.

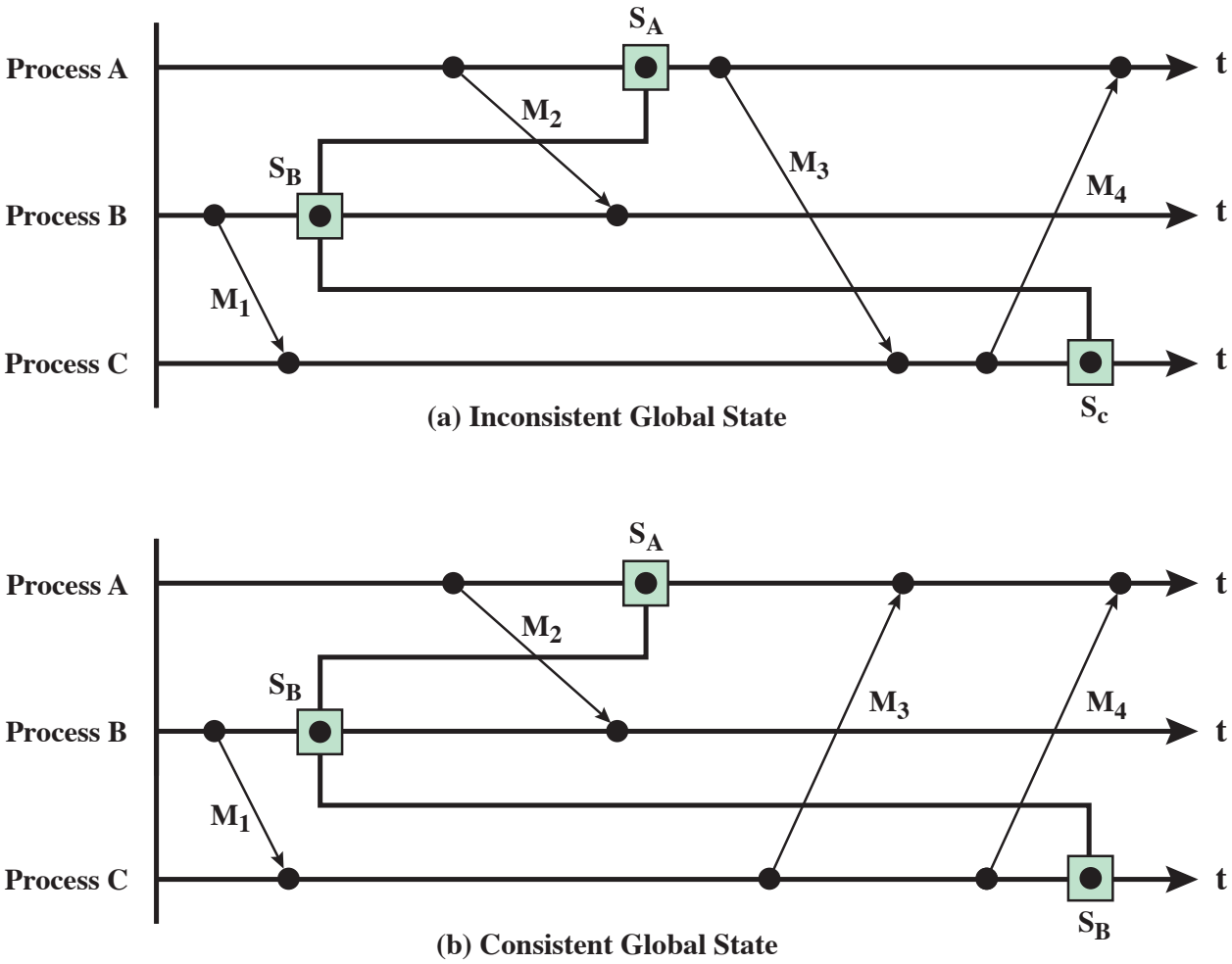


Figure 18.4 Inconsistent and Consistent Global States

We desire that the distributed snapshot record a consistent global state. A global state is consistent if for every process state that records the receipt of a message, the sending of that message is recorded in the process state of the process that sent the message. Figure 18.4b gives an example. An inconsistent global state arises if a process has recorded the receipt of a message but the corresponding sending process has not recorded that the message has been sent (Figure 18.4a).

The Distributed Snapshot Algorithm

A distributed snapshot algorithm that records a consistent global state has been described in [CHAN85]. The algorithm assumes that messages are delivered in the order that they are sent and that no messages are lost. A reliable transport protocol (e.g., TCP) satisfies these requirements. The algorithm makes use of a special control message, called a **marker**.

Some process initiates the algorithm by recording its state and sending a marker on all outgoing channels before any more messages are sent. Each process *p* then proceeds as follows. Upon the first receipt of the marker (say from process *q*), receiving process *p* performs the following:

1. *p* records its local state S_p.
2. *p* records the state of the incoming channel from *q* to *p* as empty.
3. *p* propagates the marker to all of its neighbors along all outgoing channels.

These steps must be performed atomically; that is, no messages can be sent or received by *p* until all 3 steps are performed.

At any time after recording its state, when *p* receives a marker from another incoming channel (say from process *r*), it performs the following:

1. *p* records the state of the channel from *r* to *p* as the sequence of messages *p* has received from *r* from the time *p* recorded its local state S_p to the time it received the marker from *r*.

The algorithm terminates at a process once the marker has been received along every incoming channel.

[ANDR90] makes the following observations about the algorithm:

1. Any process may start the algorithm by sending out a marker. In fact, several nodes could independently decide to record the state and the algorithm would still succeed.
2. The algorithm will terminate in finite time if every message (including marker messages) is delivered in finite time.
3. This is a distributed algorithm: Each process is responsible for recording its own state and the state of all incoming channels.
4. Once all of the states have been recorded (the algorithm has terminated at all processes), the consistent global state obtained by the algorithm can be assembled at every process by having every process send the state data that it has recorded along every outgoing channel and having every process forward the state data that it receives along every outgoing channel. Alternatively, the initiating process could poll all processes to acquire the global state.
5. The algorithm does not affect and is not affected by any other distributed algorithm that the processes are participating in.

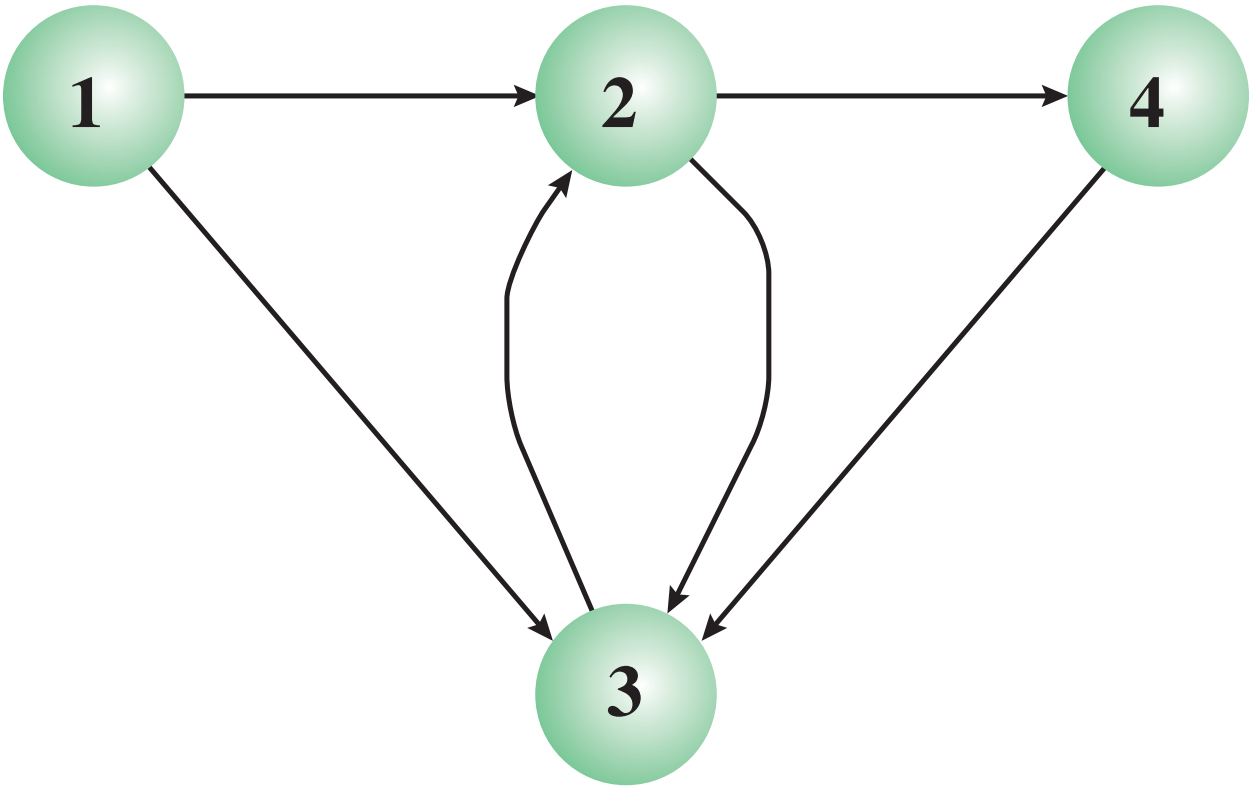


Figure 18.5 Process and Channel Graph

As an example of the use of the algorithm (taken from [BEN06]), consider the set of processes illustrated in Figure 18.5. Each process is represented by a node, and each unidirectional channel is represented by a line between two nodes, with the direction indicated by an arrowhead. Suppose that the snapshot algorithm is run, with nine messages being sent along each of its outgoing channels by each process. Process 1 decides to record the global state after sending six messages and process 4 independently decides to record the global state after sending three messages. Upon termination, the snapshots are collected from each process; the results are shown in Figure 18.6. Process 2 sent four messages on each of the two outgoing channels to processes 3 and 4 prior to the recording of the state. It received four messages from process 1 before recording its state, leaving messages 5 and 6 to be associated with the channel. The

Process 1	**Process 3**
Outgoing channels 2 sent 1,2,3,4,5,6 3 sent 1,2,3,4,5,6 Incoming channels	Outgoing channels 2 sent 1,2,3,4,5,6,7,8 Incoming channels 1 received 1,2,3 stored 4,5,6 2 received 1,2,3 stored 4 4 received 1,2,3
Process 2	**Process 4**
Outgoing channels 3 sent 1,2,3,4 4 sent 1,2,3,4 Incoming channels 1 received 1,2,3,4 stored 5,6 3 received 1,2,3,4,5,6,7,8	Outgoing channels 3 sent 1,2,3 Incoming channels 2 received 1,2 stored 3,4

Figure 18.6 An Example of a Snapshot

reader should check the snapshot for consistency: Each message sent was either received at the destination process or recorded as being in transit in the channel.

The distributed snapshot algorithm is a powerful and flexible tool. It can be used to adapt any centralized algorithm to a distributed environment, because the basis of any centralized algorithm is knowledge of the global state. Specific examples include detection of deadlock and detection of process termination (e.g., see [BEN06], [LYNC96]). It can also be used to provide a checkpoint of a distributed algorithm to allow rollback and recovery if a failure is detected.

18.3 DISTRIBUTED MUTUAL EXCLUSION

Recall that in Chapters 5 and 6 we addressed issues relating to the execution of concurrent processes. Two key problems that arose were those of mutual

exclusion and deadlock. Chapters 5 and 6 focused on solutions to this problem in the context of a single system, with one or more processors but with a common main memory. In dealing with a distributed operating system and a collection of processors that do not share common main memory or clock, new difficulties arise and new solutions are called for. Algorithms for mutual exclusion and deadlock must depend on the exchange of messages and cannot depend on access to common memory. In this section and the next, we examine mutual exclusion and deadlock in the context of a distributed operating system.

Distributed Mutual Exclusion Concepts

When two or more processes compete for the use of system resources, there is a need for a mechanism to enforce mutual exclusion. Suppose that two or more processes require access to a single nonsharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device, receiving status information, sending data, and/or receiving data. We will refer to such a resource as a critical resource, and the portion of the program that uses it as a critical section of the program. It is important that only one program at a time be allowed in its critical section. We cannot simply rely on the operating system to understand and enforce this restriction, because the detailed requirement may not be obvious. In the case of the printer, for example, we wish any individual process to have control of the printer while it prints an entire file. Otherwise, lines from competing processes will be interleaved.

The successful use of concurrency among processes requires the ability to define critical sections and enforce mutual exclusion. This is fundamental for any concurrent processing scheme. Any facility or capability that is to provide support for mutual exclusion should meet the following requirements:

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its noncritical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

Figure 18.7 shows a model that we can use for examining approaches to mutual exclusion in a distributed context. We assume some number of systems interconnected by some type of networking facility. Within each system, we assume that some function or process within the operating system is responsible for resource allocation. Each such process controls a number of resources and serves a number of user processes. The task is to devise an algorithm by which these processes may cooperate in enforcing mutual exclusion.

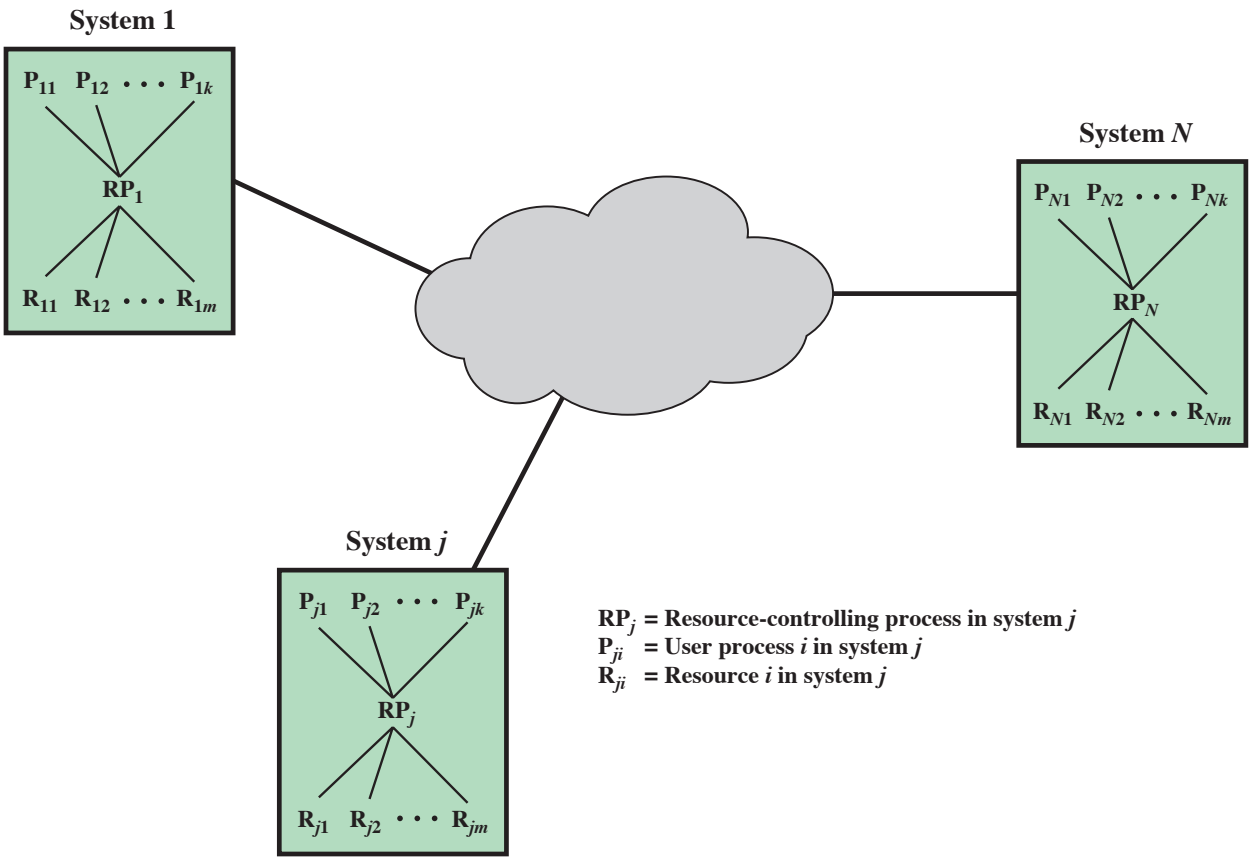


Figure 18.7 Model for Mutual Exclusion Problem in Distributed Process Management

Algorithms for mutual exclusion may be either centralized or distributed. In a fully **centralized algorithm**, one node is designated as the control node and controls access to all shared objects. When any process requires access to a critical resource, it issues a Request to its local resource-controlling process. This process, in turn, sends a Request message to the control node, which returns a Reply (permission) message when the shared object becomes available. When a process has finished with a resource, a Release message is sent to the control node. Such a centralized algorithm has two key properties:

1. Only the control node makes resource-allocation decisions.

2. All necessary information is concentrated in the control node, including the identity and location of all resources and the allocation status of each resource.

The centralized approach is straightforward, and it is easy to see how mutual exclusion is enforced: The control node will not satisfy a request for a resource until that resource has been released. However, such a scheme suffers several drawbacks. If the control node fails, then the mutual exclusion mechanism breaks down, at least temporarily. Furthermore, every resource allocation and deallocation requires an exchange of messages with the control node. Thus, the control node may become a bottleneck.

Because of the problems with centralized algorithms, there has been more interest in the development of distributed algorithms. A fully **distributed algorithm** is characterized by the following properties [MAEK87]:

1. All nodes have an equal amount of information, on average.
2. Each node has only a partial picture of the total system and must make decisions based on this information.
3. All nodes bear equal responsibility for the final decision.
4. All nodes expend equal effort, on average, in effecting a final decision.
5. Failure of a node, in general, does not result in a total system collapse.
6. There exists no systemwide common clock with which to regulate the timing of events.

Points 2 and 6 may require some elaboration. With respect to point 2, some distributed algorithms require that all information known to any node be communicated to all other nodes. Even in this case, at any given time, some of that information will be in transit and will not have arrived at all of

the other nodes. Thus, because of time delays in message communication, a node's information is usually not completely up to date and is in that sense only partial information.

With respect to point 6, because of the delay in communication among systems, it is impossible to maintain a systemwide clock that is instantly available to all systems. Furthermore, it is also technically impractical to maintain one central clock and to keep all local clocks synchronized precisely to that central clock; over a period of time, there will be some drift among the various local clocks that will cause a loss of synchronization.

It is the delay in communication, coupled with the lack of a common clock, that makes it much more difficult to develop mutual exclusion mechanisms in a distributed system compared to a centralized system. Before looking at some algorithms for distributed mutual exclusion, we examine a common approach to overcoming the clock inconsistency problem.

Ordering of Events in a Distributed System

Fundamental to the operation of most distributed algorithms for mutual exclusion and deadlock is the temporal ordering of events. The lack of a common clock or a means of synchronizing local clocks is thus a major constraint. The problem can be expressed in the following manner. We would like to be able to say that an event *a* at system *i* occurred before (or after) event *b* at system *j*, and we would like to be able to arrive consistently at this conclusion at all systems in the network. Unfortunately, this statement is not precise for two reasons. First, there may be a delay between the actual occurrence of an event and the time that it is observed on some other system. Second, the lack of synchronization leads to a variance in clock readings on different systems.

To overcome these difficulties, a method referred to as timestamping has been proposed by Lamport [LAMP78], which orders events in a distributed system without using physical clocks. This technique is so efficient and effective that it is used in the great majority of algorithms for distributed mutual exclusion and deadlock.

To begin, we need to decide on a definition of the term *event*. Ultimately, we are concerned with actions that occur at a local system, such as a process entering or leaving its critical section. However, in a distributed system, the way in which processes interact is by means of messages. Therefore, it makes sense to associate events with messages. A local event can be bound to a message very simply; for example, a process can send a message when it desires to enter its critical section or when it is leaving its critical section. To avoid ambiguity, we associate events with the sending of messages only, not with the receipt of messages. Thus, each time that a process transmits a message, an event is defined that corresponds to the time that the message leaves the process.

The timestamping scheme is intended to order events consisting of the transmission of messages. Each system i in the network maintains a local counter, C_i, which functions as a clock. Each time a system transmits a message, it first increments its clock by 1. The message is sent in the form

$$(m, T_i, i)$$

where

m = contents of the message
T_i = timestamp for this message, set to equal C_i
i = numerical identifier of this system in the distributed system

When a message is received, the receiving system j sets its clock to one more than the maximum of its current value and the incoming timestamp:

$$C_j \leftarrow 1 + \max[C_j , T_i]$$

At each site, the ordering of events is determined by the following rules. For a message x from site i and a message y from site j, x is said to precede y if one of the following conditions holds:

1. If $T_i < T_j$, or

2. If $T_i = T_j$ and $i < j$

The time associated with each message is the timestamp accompanying the message, and the ordering of these times is determined by the two foregoing rules. That is, two messages with the same timestamp are ordered by the numbers of their sites. Because the application of these rules is independent of site, this approach avoids any problems of drift among the various clocks of the communicating processes.

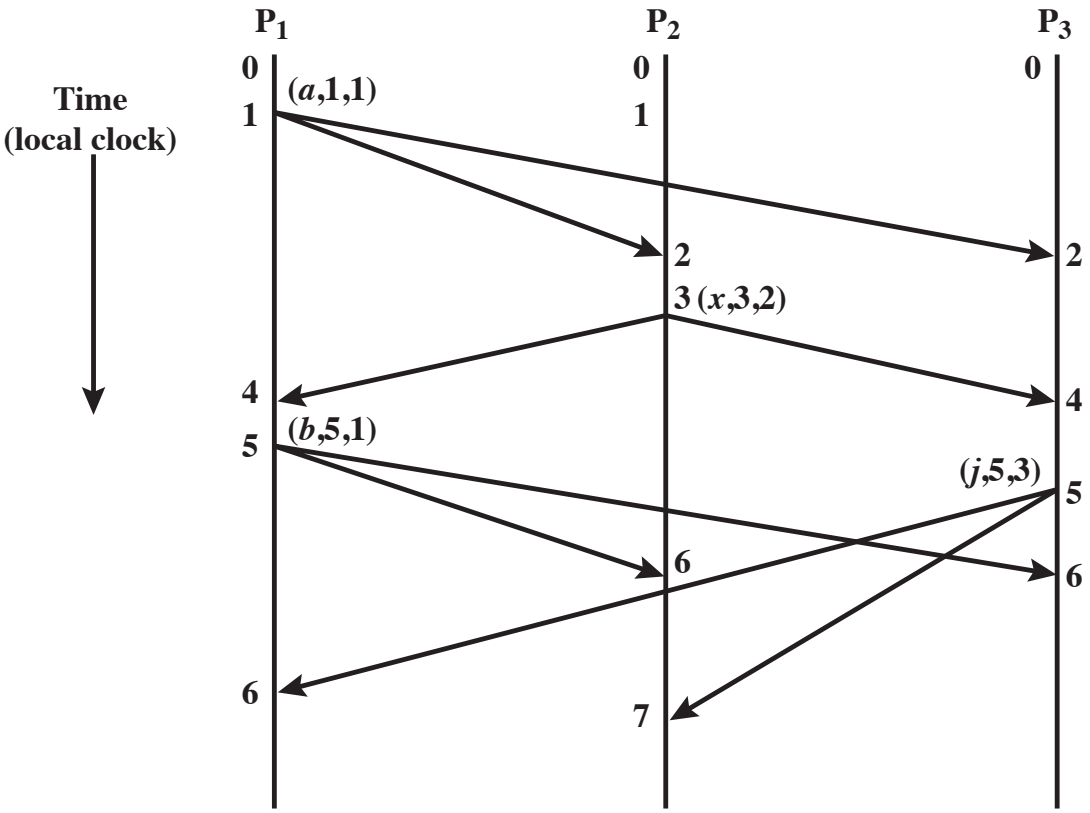


Figure 18.8 Example of Operation of Timestamping Algorithm

An example of the operation of this algorithm is shown in Figure 18.8. There are three sites, each of which is represented by a process that controls the timestamping algorithm. Process P_1 begins with a clock value of 0. To transmit message *a*, it increments its clock by 1 and transmits (*a*, 1, 1), where the first numerical value is the timestamp and the second is the identity of the site. This message is received by processes at sites 2 and 3. In both cases, the local clock has a value of zero and is set to a value of 2 = 1 + max[0, 1]. P_2 issues the next message, first incrementing its clock to 3. Upon receipt of this message, P_1 and P_3 increment their clocks to 4. Then P_1 issues message *b* and P_3 issues message *j* at about the same time and with the same timestamp. Because of the ordering principle outlined previously,

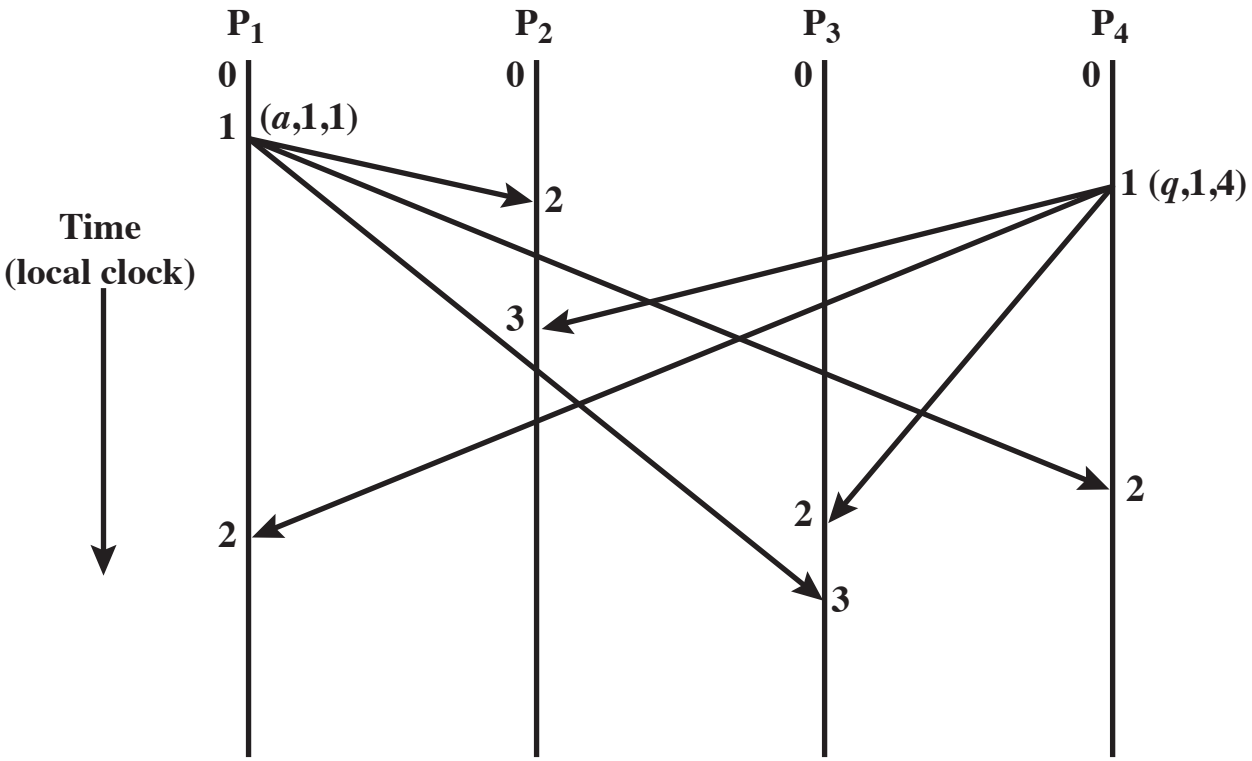


Figure 18.9 Another Example of Operation of Timestamping Algorithm

this causes no confusion. After all of these events have taken place, the ordering of messages is the same at all sites, namely {*a*, *x*, *b*, *j*}.

The algorithm works in spite of differences in transmission times between pairs of systems, as illustrated in Figure 18.9. Here, P_1 and P_4 issue messages with the same timestamp. The message from P_1 arrives earlier than that of P_4 at site 2 but later than that of P_4 at site 3. Nevertheless, after all messages have been received at all sites, the ordering of messages is the same at all sites: {*a*, *q*}.

Note that the ordering imposed by this scheme does not necessarily correspond to the actual time sequence. For the algorithms based on this timestamping scheme, it is not important which event actually happened

first. It is only important that all processes that implement the algorithm agree on the ordering that is imposed on the events.

In the two examples just discussed, each message is sent from one process to all other processes. If some messages are not sent this way, some sites do not receive all of the messages in the system and it is therefore impossible that all sites have the same ordering of messages. In such a case, a collection of partial orderings exist. However, we are primarily concerned with the use of timestamps in distributed algorithms for mutual exclusion and deadlock detection. In such algorithms, a process usually sends a message (with its timestamp) to every other process, and the timestamps are used to determine how the messages are processed.

Distributed Queue

First Version

One of the earliest proposed approaches to providing distributed mutual exclusion is based on the concept of a distributed queue [LAMP78]. The algorithm is based on the following assumptions:

1. A distributed system consists of *N* nodes, uniquely numbered from 1 to *N*. Each node contains one process that makes requests for mutually exclusive access to resources on behalf of other processes; this process also serves as an arbitrator to resolve incoming requests from other nodes that overlap in time.
2. Messages sent from one process to another are received in the same order in which they are sent.
3. Every message is correctly delivered to its destination in a finite amount of time.

4. The network is fully connected; this means that every process can send messages directly to every other process, without requiring an intermediate process to forward the message.

Assumptions 2 and 3 can be realized by the use of a reliable transport protocol, such as TCP (Chapter 13).

For simplicity, we describe the algorithm for the case in which each site only controls a single resource. The generalization to multiple resources is trivial.

The algorithm attempts to generalize an algorithm that would work in a straightforward manner in a centralized system. If a single central process managed the resource, it could queue incoming requests and grant requests in a first-in-first-out manner. To achieve this same algorithm in a distributed system, all of the sites must have a copy of the same queue. Timestamping can be used to assure that all sites agree on the order in which resource requests are to be granted. One complication arises: Because it takes some finite amount of time for messages to transit a network, there is a danger that two different sites will not agree on which process is at the head of the queue. Consider Figure 18.9. There is a point at which message *a* has arrived at P_2 and message *q* has arrived at P_3, but both messages are still in transit to other processes. Thus, there is a period of time in which P_1 and P_2 consider message *a* to be the head of the queue and in which P_3 and P_4 consider message *q* to be the head of the queue. This could lead to a violation of the mutual exclusion requirement. To avoid this, the following rule is imposed: For a process to make an allocation decision based on its own queue, it needs to have received a message from each of the other sites such that the process is guaranteed that no message earlier than its own head of queue is still in transit. This rule is explained in part 3b of the algorithm described subsequently.

At each site, a data structure is maintained that keeps a record of the most recent message received from each site (including the most recent message generated at this site). Lamport refers to this structure as a queue; actually it is an array with one entry for each site. At any instant, entry $q[j]$ in the local array contains a message from P_j. The array is initialized as follows:

$$q[j] = (\text{Release}, 0, j) \qquad j = 1, \ldots, N$$

Three types of messages are used in this algorithm:

- (Request, T_i, i): A request for access to a resource is made by P_i.
- (Reply, T_j, j): P_j grants access to a resource under its control.
- (Release, T_k, k): P_k releases a resource previously allocated to it.

The algorithm is as follows:

1. When P_i requires access to a resource, it issues a request (Request, T_i, i), timestamped with the current local clock value. It puts this message in its own array at $q[i]$ and sends the message to all other processes.
2. When P_j receives (Request, T_i, i), it puts this message in its own array at $q[i]$. If $q[j]$ does not contain a request message, then P_j transmits (Reply, T_j, j) to P_i. It is this action that implements the rule described previously, which assures that no earlier Request message is in transit at the time of a decision.
3. P_i can access a resource (enter its critical section) when both of these conditions hold:

a. P_i's own Request message in array q is the earliest Request message in the array; because messages are consistently ordered at all sites, this rule permits one and only one process to access the resource at any instant.

b. All other messages in the local array are later than the message in $q[i]$; this rule guarantees that P_i has learned about all requests that preceded its current request.

4. P_i releases a resource by issuing a release (Release, T_i, i), which it puts in its own array and transmits to all other processes.

5. When P_i receives (Release, T_j, j), it replaces the current contents of $q[j]$ with this message.

6. When P_i receives (Reply, T_j, j), it replaces the current contents of $q[j]$ with this message.

It is easily shown that this algorithm enforces mutual exclusion, is fair, avoids deadlock, and avoids starvation:

- **Mutual exclusion:** Requests for entry into the critical section are handled according to the ordering of messages imposed by the timestamping mechanism. Once P_i decides to enter its critical section, there can be no other Request message in the system that was transmitted before its own. This is true because P_i has by then necessarily received a message from all other sites and these messages from other sites date from later than its own Request message. We can be sure of this because of the Reply message mechanism; remember that messages between two sites cannot arrive out of order.
- **Fair:** Requests are granted strictly on the basis of timestamp ordering. Therefore, all processes have equal opportunity.

- **Deadlock free:** Because the timestamp ordering is consistently maintained at all sites, deadlock cannot occur.
- **Starvation free:** Once P_i has completed its critical section, it transmits the Release message. This has the effect of deleting P_i's Request message at all other sites, allowing some other process to enter its critical section.

As a measure of efficiency of this algorithm, note that to guarantee exclusion, $3 \times (N - 1)$ messages are required: $(N - 1)$ Request messages, $(N - 1)$ Reply messages, and $(N - 1)$ Release messages.

SECOND VERSION

A refinement of the Lamport algorithm was proposed in [RICA81]. It seeks to optimize the original algorithm by eliminating Release messages. The same assumptions as before are in force, except that it is not necessary that messages sent from one process to another are received in the same order in which they are sent.

As before, each site includes one process that controls resource allocation. This process maintains an array q and obeys the following rules:

1. When P_i requires access to a resource, it issues a request (Request, T_i, i), timestamped with the current local clock value. It puts this message in its own array at $q[i]$ and sends the message to all other processes.
2. When P_j receives (Request, T_i, i), it obeys the following rules:
   a. If P_j is currently in its critical section, it defers sending a Reply message (see Rule 4, which follows)
   b. If P_j is not waiting to enter its critical section (has not issued a Request that is still outstanding), it transmits (Reply, T_j, j) to P_i.

c. If P_j is waiting to enter its critical section and if the incoming message follows P_j's request, then it puts this message in its own array at $q[i]$ and defers sending a Reply message.

d. If P_j is waiting to enter its critical section and if the incoming message precedes P_j's request, then it puts this message in its own array at $q[i]$ and transmits (Reply, T_j, j) to P_i.

3. P_i can access a resource (enter its critical section) when it has received a Reply message from all other processes.

4. When P_i leaves its critical section, it releases the resource by sending a Reply message to each pending Request.

The state transition diagram for each process is shown in Figure 18.10.

To summarize, when a process wishes to enter its critical section, it sends a timestamped Request message to all other processes. When it receives a Reply from all other processes, it may enter its critical section. When a process receives a Request from another process, it must eventually send a matching Reply. If a process does not wish to enter its critical section, it sends a Reply at once. If it wants to enter its critical section, it compares the timestamp of its Request with that of the last Request received, and if the latter is more recent, it defers its Reply; otherwise Reply is sent at once.

With this method, $2 \times (N - 1)$ messages are required: $(N - 1)$ Request messages to indicate P_i's intention of entering its critical section, and $(N - 1)$ Reply messages to allow the access it has requested.

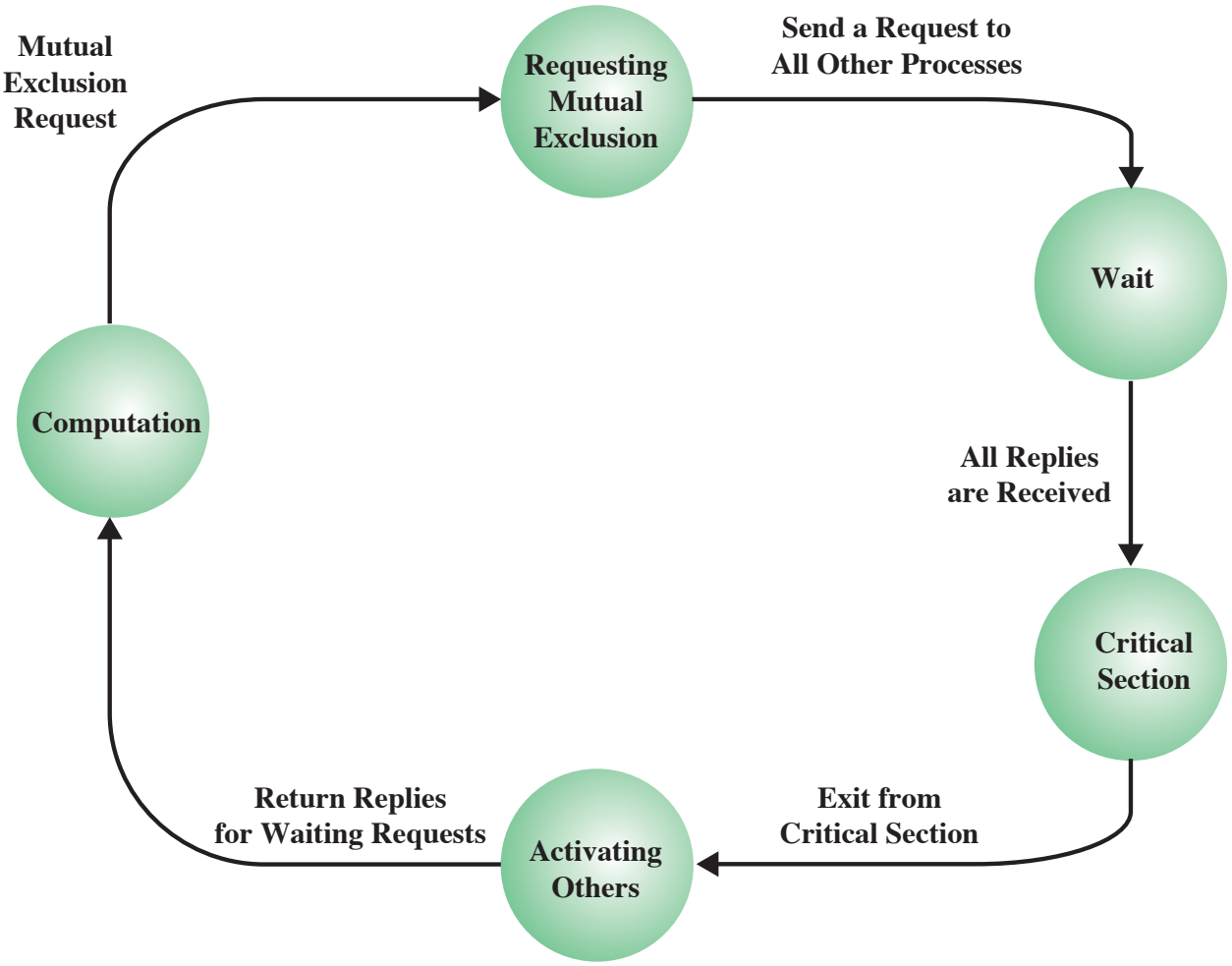


Figure 18.10 State Diagram for Algorithm in [RICA81]

The use of timestamping in this algorithm enforces mutual exclusion. It also avoids deadlock. To prove the latter, assume the opposite: It is possible that, when there are no more messages in transit, we have a situation in which each process has transmitted a Request and has not received the necessary Reply. This situation cannot arise, because a decision to defer a Reply is based on a relation that orders Requests. There is therefore one Request that has the earliest timestamp and that will receive all the necessary Replies. Deadlock is therefore impossible.

Starvation is also avoided because Requests are ordered. Because Requests are served in that order, every Request will at some stage become the oldest and will then be served.

A Token-Passing Approach

A number of investigators have proposed a quite different approach to mutual exclusion, which involves passing a token among the participating processes. The token is an entity that at any time is held by one process. The process holding the token may enter its critical section without asking permission. When a process leaves its critical section, it passes the token to another process.

In this subsection, we look at one of the most efficient of these schemes. It was first proposed in [SUZU82]; a logically equivalent proposal also appeared in [RICA83]. For this algorithm, two data structures are needed. The token, which is passed from process to process, is actually an array, `token`, whose *k*th element records timestamp of the last time that the token visited process P_k. In addition, each process maintains an array, request, whose *j*th element records the timestamp of the last Request received from P_j.

The procedure is as follows. Initially, the token is assigned arbitrarily to one of the processes. When a process wishes to use its critical section, it may do so if it currently possesses the token; otherwise it broadcasts a timestamped request message to all other processes and waits until it receives the token. When process P_j leaves its critical section, it must transmit the token to some other process. It chooses the next process to receive the token by searching the request array in the order $j + 1, j + 2, \ldots, 1, 2, \ldots, j - 1$ for the first entry request [*k*] such that the timestamp for P_k's last request for the token is greater than the value recorded in the token for P_k's last holding of the token, that is, request [*k*] > token [*k*].

```
if (!token_present) {
     clock++;                                        /* Prelude */
     broadcast (Request, clock, i);
     wait (access, token);
     token_present = true;
}

token_held = true;
<critical section>;

token[i] = clock;                                   /* Postlude */
token_held = false;
for (int j = i + 1; j < n; j++) {
     if (request(j) > token[j] && token_present) {
          token_present = false;
          send (access, token[j]);
     }
}
```

(a) First Part

```
if (received (Request, k, j)) {
     request (j) = max(request(j), k);
     if (token_present && !token_held)
       <text of postlude>;
}
```

(b) Second Part

Notation

send (j, access, token)	end message of type access, with token, by process j
broadcast (request, clock, i)	send message from process i of type request, with time-stamp clock, to all other processes
received (request, t, j)	receive message from process j of type request, with time-stamp t

Figure 18.11 Token-Passing Algorithm (for process P_i)

Figure 18.11 depicts the algorithm, which is in two parts. The first part deals with the use of the critical section and consists of a prelude, followed by the critical section, followed by a postlude. The second part concerns the action to be taken upon receipt of a request. The variable clock is the local

counter used for the timestamp function. The operation wait (access, token) causes the process to wait until a message of the type "access" is received, which is then put into the variable array token.

The algorithm requires either of the following:

- *N* messages (*N* – 1 to broadcast the request and 1 to transfer the token) when the requesting process does not hold the token
- No messages, if the process already holds the token

18.4 DISTRIBUTED DEADLOCK

In Chapter 6, we defined deadlock as the permanent blocking of a set of processes that either compete for system resources or communicate with one another. This definition is valid for a single system as well as for a distributed system. As with mutual exclusion, deadlock presents more complex problems in a distributed system, compared with a shared memory system. Deadlock handling is complicated in a distributed system because no node has accurate knowledge of the current state of the overall system and because every message transfer between processes involves an unpredictable delay.

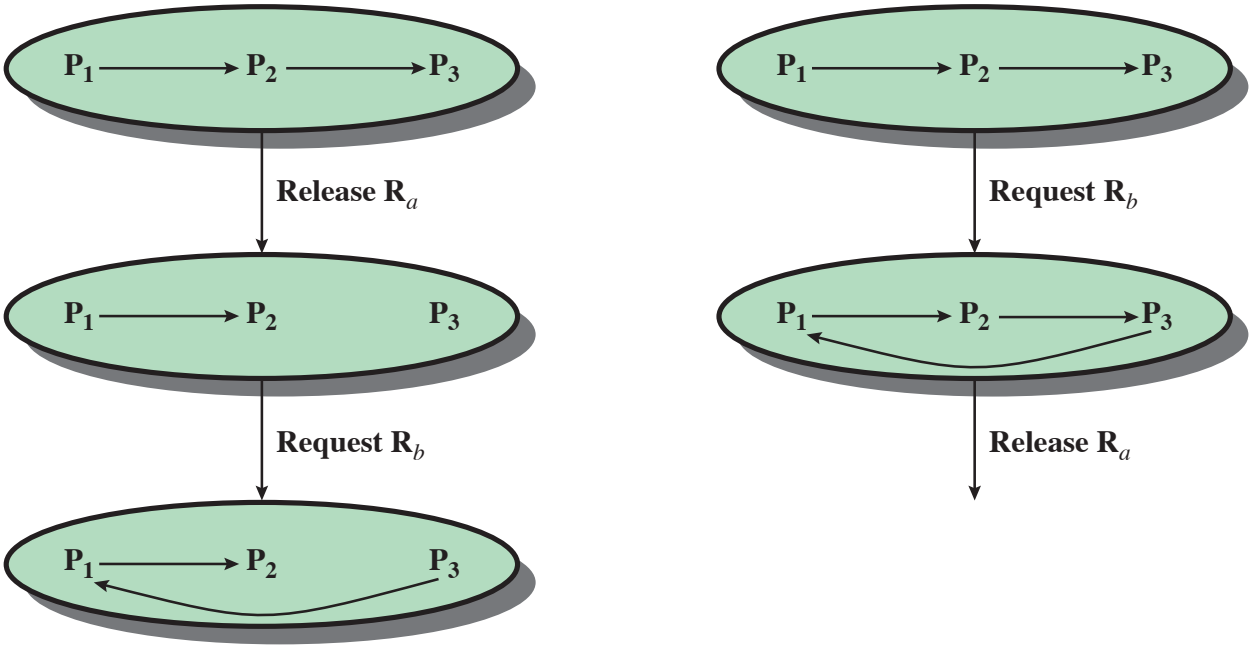
Two types of distributed deadlock have received attention in the literature: those that arise in the allocation of resources, and those that arise with the communication of messages. In resource deadlocks, processes attempt to access resources, such as data objects in a database or I/O resources on a server; deadlock occurs if each process in a set of processes requests a resource held by another process in the set. In communications deadlocks, messages are the resources for which processes wait; deadlock occurs if each process in a set is waiting for a message from another process in the set and no process in the set ever sends a message.

Deadlock in Resource Allocation

Recall from Chapter 6 that a deadlock in resource allocation exists only if all of the following conditions are met:

- **Mutual exclusion:** Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.
- **Hold and wait:** A process may hold allocated resources while awaiting assignment of others.
- **No preemption:** No resource can be forcibly removed from a process holding it.
- **Circular wait:** A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

The aim of an algorithm that deals with deadlock is either to prevent the formation of a circular wait or to detect its actual or potential occurrence. In a distributed system, the resources are distributed over various sites and access to them is regulated by control processes that do not have complete, up-to-date knowledge of the global state of the system and must therefore make their decisions on the basis of local information. Thus, new deadlock algorithms are required.



(a) Release arrives before request **(b) Request arrives before release**

Figure 18.12 Phantom Deadlock

One example of the difficulty faced in distributed deadlock management is the phenomenon of phantom deadlock. An example of phantom deadlock is illustrated in Figure 18.12. The notation $P_1 \rightarrow P_2 \rightarrow P_3$ means that P_1 is halted waiting for a resource held by P_2, and P_2 is halted waiting for a resource held by P_3. Let us say that at the beginning of the example, P_3 owns resource R_a and P_1 owns resource R_b. Suppose now that P_3 issues first a message releasing R_a and then a message requesting R_b. If the first message reaches a cycle-detecting process before the second, the sequence of Figure 18.12a results, which properly reflects resource requirements. If, however, the second message arrives before the first message, a deadlock is registered (Figure 18.12b). This is a false detection, not a real deadlock, due to the lack of a global state, such as would exist in a centralized system.

DEADLOCK PREVENTION

Two of the deadlock prevention techniques discussed in Chapter 6 can be used in a distributed environment.

1. The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering. A major disadvantage of this method is that resources may not be requested in the order in which they are used; thus resources may be held longer than necessary.
2. The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time, and blocking the process until all requests can be granted simultaneously. This approach is inefficient in two ways. First, a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources. Second, resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes.

Both of these methods require that a process determine its resource requirements in advance. This is not always the case; an example is a database application in which new items can be added dynamically. As an example of an approach that does not require this foreknowledge, we consider two algorithms proposed in [ROSE78]. These were developed in the context of database work, so we shall speak of transactions rather than processes.

```
if (e(T2) < e(T1))
   halt_T2 ('wait');
else
   kill_T2 ('die');
```

(a) Wait-die method

```
if (e(T2) < e(T1))
   kill_T1 ('wound');
else
   halt_T2 ('wait');
```

(b) Wound-wait method

Figure 18.13 Deadlock Prevention Methods

The proposed methods make use of timestamps. Each transaction carries throughout its lifetime the timestamp of its creation. This establishes a strict ordering of the transactions. If a resource R already being used by transaction T1 is requested by another transaction T2, the conflict is resolved by comparing their timestamps. This comparison is used to prevent the formation of a circular-wait condition. Two variations of this basic method are proposed by the authors, referred to as the "wait-die" method and the "wound-wait" method.

Let us suppose that T1 currently holds R and that T2 issues a request. For the **wait-die method**, Figure 18.13a shows the algorithm used by the resource allocator at the site of R. The timestamps of the two transactions are denoted as e(T1) and e(T2). If T2 is older, it is blocked until T1 releases R, either by actively issuing a release or by being "killed" when requesting another resource. If T2 is younger, then T2 is restarted but with the same timestamp as before.

Thus, in a conflict, the older transaction takes priority. Because a killed transaction is revived with its original timestamp, it grows older and therefore gains increased priority. No site needs to know the state of allocation of all resources. All that are required are the timestamps of the transactions that request its resources.

The **wound-wait method** immediately grants the request of an older transaction by killing a younger transaction that is using the required resource. This is shown in Figure 18.13b. In contrast to the wait-die method, a transaction never has to wait for a resource being used by a younger transaction.

Deadlock Avoidance

Deadlock avoidance is a technique in which a decision is made dynamically whether a given resource allocation request could, if granted, lead to a deadlock. [SING94] points out that distributed deadlock avoidance is impractical for the following reasons:

1. Every node must keep track of the global state of the system; this requires substantial storage and communications overhead.
2. The process of checking for a safe global state must be mutually exclusive. Otherwise, two nodes could each be considering the resource request of a different process and concurrently reach the conclusion that it is safe to honor the request, when in fact if both requests are honored, deadlock will result.
3. Checking for safe states involves considerable processing overhead for a distributed system with a large number of processes and resources.

Deadlock Detection

With deadlock detection, processes are allowed to obtain free resources as they wish, and the existence of a deadlock is determined after the fact. If a deadlock is detected, one of the *constituent* processes is selected and required to release the resources necessary to break the deadlock.

Table 18.1 Distributed Deadlock Detection Strategies

Centralized Algorithms		Hierarchical Algorithms		Distributed Algorithms	
Strengths	**Weaknesses**	**Strengths**	**Weaknesses**	**Strengths**	**Weaknesses**
•Algorithms are conceptually simple and easy to implement •Central site has complete information and can optimally resolve deadlocks	•Considerable communications overhead; every node must send state information to central node •Vulnerable to failure of central node	•Not vulnerable to single point of failure •Deadlock resolution activity is limited if most potential deadlocks are relatively localized	•May be difficult to configure system so that most potential deadlocks are localized; otherwise there may actually be more overhead than in a distributed approach	•Not vulnerable to single point of failure •No node is swamped with deadlock detection activity	•Deadlock resolution is cumbersome because several sites may detect the same deadlock and may not be aware of other nodes involved in the deadlock •Algorithms are difficult to design because of timing considerations

The difficulty with distributed deadlock detection is that each site only knows about its own resources, whereas a deadlock may involve distributed resources. Several approaches are possible, depending on whether the system control is centralized, hierarchical, or distributed (Table 18.1).

With **centralized control**, one site is responsible for deadlock detection. All request and release messages are sent to the central process as well as to the process that controls the particular resource. Because the central process has a complete picture, it is in a position to detect a deadlock. This approach requires a lot of messages and is vulnerable to a failure of the central site. In addition, phantom deadlocks may be detected.

With **hierarchical control**, the sites are organized in a tree structure, with one site serving as the root of the tree. At each node, other than leaf nodes, information about the resource allocation of all dependent nodes is collected. This permits deadlock detection to be done at lower levels than the root node. Specifically, a deadlock that involves a set of resources will be

detected by the node that is the common ancestor of all sites whose resources are among the objects in conflict.

With **distributed control**, all processes cooperate in the deadlock detection function. In general, this means that considerable information must be exchanged, with timestamps; thus the overhead is significant. [RAYN88] cites a number of approaches based on distributed control, and [DATT90] provides a detailed examination of one approach.

We now give an example of a distributed deadlock detection algorithm ([DATT92], [JOHN91]). The algorithm deals with a distributed database system in which each site maintains a portion of the database and transactions may be initiated from each site. A transaction can have at most one outstanding resource request. If a transaction needs more than one data object, the second data object can be requested only after the first data object has been granted.

Associated with each data object *i* at a site are two parameters: a unique identifier D_i and the variable Locked_by(D_i). This latter variable has the value nil if the data object is not locked by any transaction; otherwise its value is the identifier of the locking transaction.

Associated with each transaction *j* at a site are four parameters:

- A unique identifier T_j
- The variable Held_by(T_j), which is set to nil if transaction T_j is executing or in a Ready state. Otherwise, its value is the transaction that is holding the data object required by transaction T_j.
- The variable Wait_for(T_j), which has the value nil if transaction T_i is not waiting for any other transaction. Otherwise, its value is the identifier of the transaction that is at the head of an ordered list of transactions that are blocked.

- A queue Request_Q(T_j), which contains all outstanding requests for data objects being held by T_j. Each element in the queue is of the form (T_k, D_k), where T_k is the requesting transaction and D_k is the data object held by T_j.

For example, suppose that transaction T_2 is waiting for a data object held by T_1, which is, in turn, waiting for a data object held by T_0. Then the relevant parameters have the following values:

Transaction	Wait_for	Held_by	Request_Q
T_0	nil	nil	T_1
T_1	T_0	T_0	T_2
T_2	T_0	T_1	nil

This example highlights the difference between Wait_for(T_i) and Held_by(T_i). Neither process can proceed until T_0 releases the data object needed by T_1, which can then execute and release the data object needed by T_2.

Figure 18.14 shows the algorithm used for deadlock detection. When a transaction makes a lock request for a data object, a server process associated with that data object either grants or denies the request. If the request is not granted, the server process returns the identity of the transaction holding the data object.

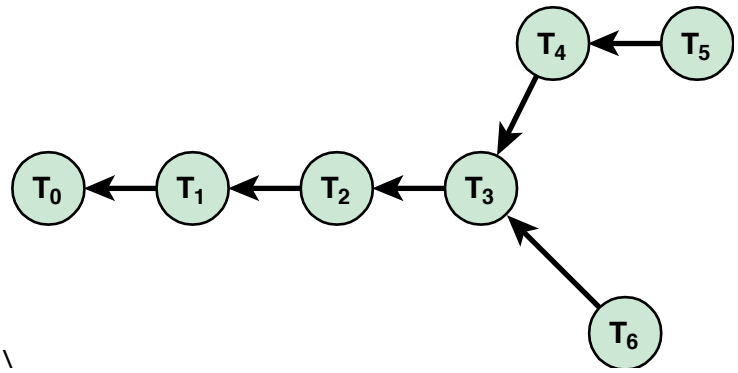
```
/* Data object Dj receiving a lock_request(Ti) */
   if (Locked_by(Dj) == null)
     send(granted);
   else  {
     send not granted to Ti;
     send Locked_by(Dj) to Ti
   }

/* Transaction Ti makes a lock request for data object Dj */
   send lock_request(Ti) to Dj;
   wait for granted/not granted;
   if (granted) {
     Locked_by(Dj) = Ti;
     Held_by(Ti) = f;
   }
   else {   /* suppose Dj is being used by transaction Tj */
     Held_by(Ti) = Tj;
     Enqueue(Ti, Request_Q(Tj));
     if (Wait_for(Tj) == null)
       Wait_for(Ti) = Tj ;
     else
       Wait_for(Ti) = Wait_for(Tj);
     update(Wait_for(Ti), Request_Q(Ti));
                                          }

/* Transaction Tj receiving an update message */

   if (Wait_for(Tj) != Wait_for(Ti))
     Wait_for(Tj) = Wait_for(Ti);
   if (intersect(Wait_for(Tj), Request_Q(Tj)) = null)
     update(Wait_for(Ti), Request_Q(Tj);
   else {
      DECLARE DEADLOCK;
      /* initiate deadlock resolution as follows */
      /* Tj is chosen as the transaction to be aborted */
      /* Tj releases all the data objects it holds */
      send_clear(Tj, Held_by(Tj));
      allocate each data object Di held by Tj to the first
            requester Tk in Request_Q(Tj);
      for (every transaction Tn in Request_Q(Tj) requesting
            data object Di held by Tj)
      {
            Enqueue(Tn, Request_Q(Tk));
      }
   }

/* Transaction Tk receiving a clear(Tj, Tk) message */
           purge the tuple having Tj as the requesting transaction from
                                 Request_Q(Tk);
```

Figure 18.14 A Distributed Deadlock Detection Algorithm

When the requesting transaction receives a granted response, it locks the data object. Otherwise, the requesting transaction updates its Held_by variable to the identity of the transaction holding the data object. It adds its identity to the Request_Q of the holding transaction. It updates is Wait_for variable either to the identity of the holding transaction (if that transaction is not waiting) or to the identity of the Wait_for variable of the holding transaction. In this way, the Wait_for variable is set to the value of the transaction that ultimately is blocking execution. Finally, the requesting transaction issues an update message to all of the transactions in its own Request_Q to modify all the Wait_for variables that are affected by this change.

When a transaction receives an update message, it updates its Wait_for variable to reflect the fact that the transaction on which it had been ultimately waiting is now blocked by yet another transaction. Then it does the actual work of deadlock detection by checking to see if it is now waiting for one of the processes that is waiting for it. If not, it forwards the update message. If so, the transaction sends a clear message to the transaction holding its requested data object and allocates every data object that it holds to the first requester in its Request_Q and enqueues remaining requesters to the new transaction.

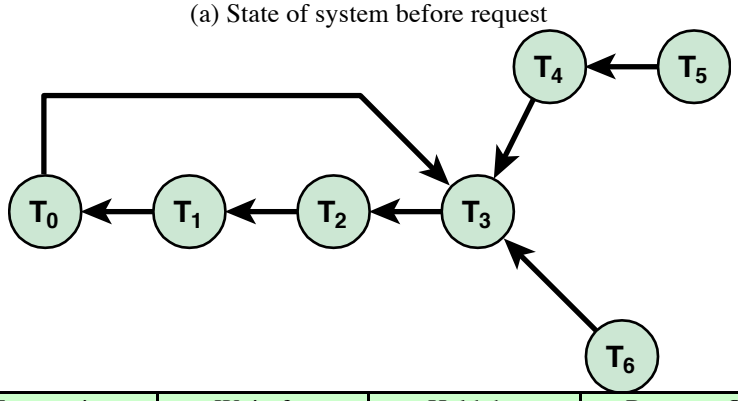
An example of the operation of the algorithm is shown in Figure 18.15. When T_0 makes a request for a data object held by T_3, a cycle is created. T_0 issues an update message that propagates from T_1 to T_2 to T_3. At this point, T_3 discovers that the intersection of its Wait_for and Request_Q variables is not empty. T_3 sends a clear message to T_2 so that T_3 is purged from Request_Q(T_2), and it releases the data objects it held, activating T_4 and T_6.



\

Transaction	Wait_for	Held_by	Request_Q
T_0	nil	nil	T_1
T_1	T_0	T_0	T_2
T_2	T_0	T_1	T_3
T_3	T_0	T_2	T_4, T_6
T_4	T_0	T_3	T_5
T_5	T_0	T_4	Nil
T_6	T_0	T_3	Nil

(a) State of system before request



Transaction	Wait_for	Held_by	Request_Q
T_0	T_0	T_3	T_1
T_1	T_0	T_0	T_2
T_2	T_0	T_1	T_3
T_3	T_0	T_2	T_4, T_6, T_0
T_4	T_0	T_3	T_5
T_5	T_0	T_4	Nil
T_6	T_0	T_3	Nil

(b) State of system after T_0 makes a request to T_3

Figure 18.15 Example of Distributed Deadlock Detection Algorithm of Figure 18.14

Deadlock in Message Communication

Mutual Waiting

Deadlock occurs in message communication when each of a group of processes is waiting for a message from another member of the group and there are no messages in transit.

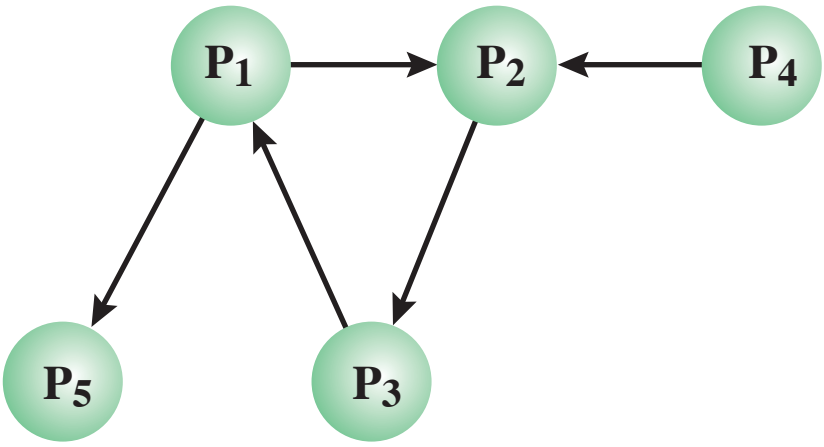
To analyze this situation in more detail, we define the dependence set (DS) of a process. For a process P_i that is halted, waiting for a message, $DS(P_i)$ consists of all processes from which P_i is expecting a message. Typically, P_i can proceed if any of the expected messages arrives. An alternative formulation is that P_i can proceed only after all of the expected messages arrive. The former situation is the more common one and is considered here.

With the preceding definition, a deadlock in a set S of processes can be defined as follows:

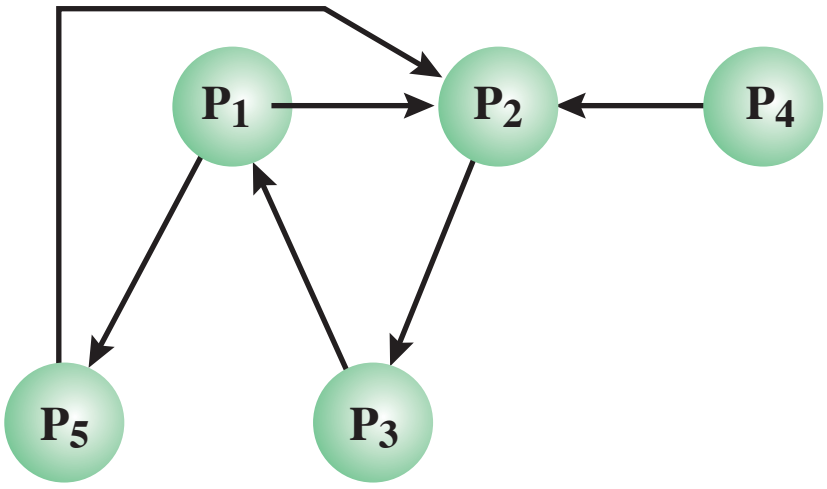
1. All the processes in S are halted, waiting for messages.
2. S contains the dependence set of all processes in S.
3. No messages are in transit between members of S.

Any process in S is deadlocked because it can never receive a message that will release it.

In graphical terms, there is a difference between message deadlock and resource deadlock. With resource deadlock, a deadlock exists if there is a closed loop, or cycle, in the graph that depicts process dependencies. In the resource case, one process is dependent on another if the latter holds a resource that the former requires. With message deadlock, the condition for deadlock is that all successors of any member of S are themselves in S.



(a) No deadlock



(b) Deadlock

Figure 18.16 Deadlock in Message Communication

Figure 18.16 illustrates the point. In Figure 18.16a, P_1 is waiting for a message from either P_2 or P_5; P_5 is not waiting for any message and so can send a message to P_1, which is therefore released. As a result, the links (P_1, P_5) and (P_1, P_2) are deleted. Figure 18.16b adds a dependency: P_5 is waiting for a message from P_2, which is waiting for a message from P_3, which is

waiting for a message from P_1, which is waiting for a message from P_2. Thus, deadlock exists.

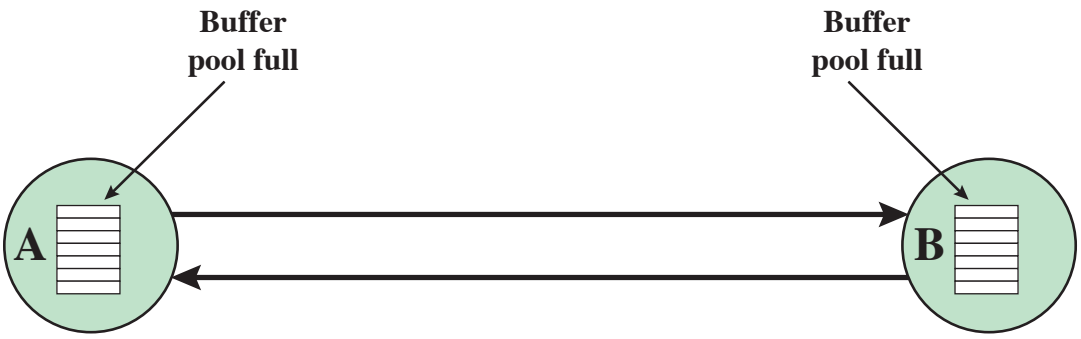
As with resource deadlock, message deadlock can be attacked by either prevention or detection. [RAYN88] gives some examples.

UNAVAILABILITY OF MESSAGE BUFFERS

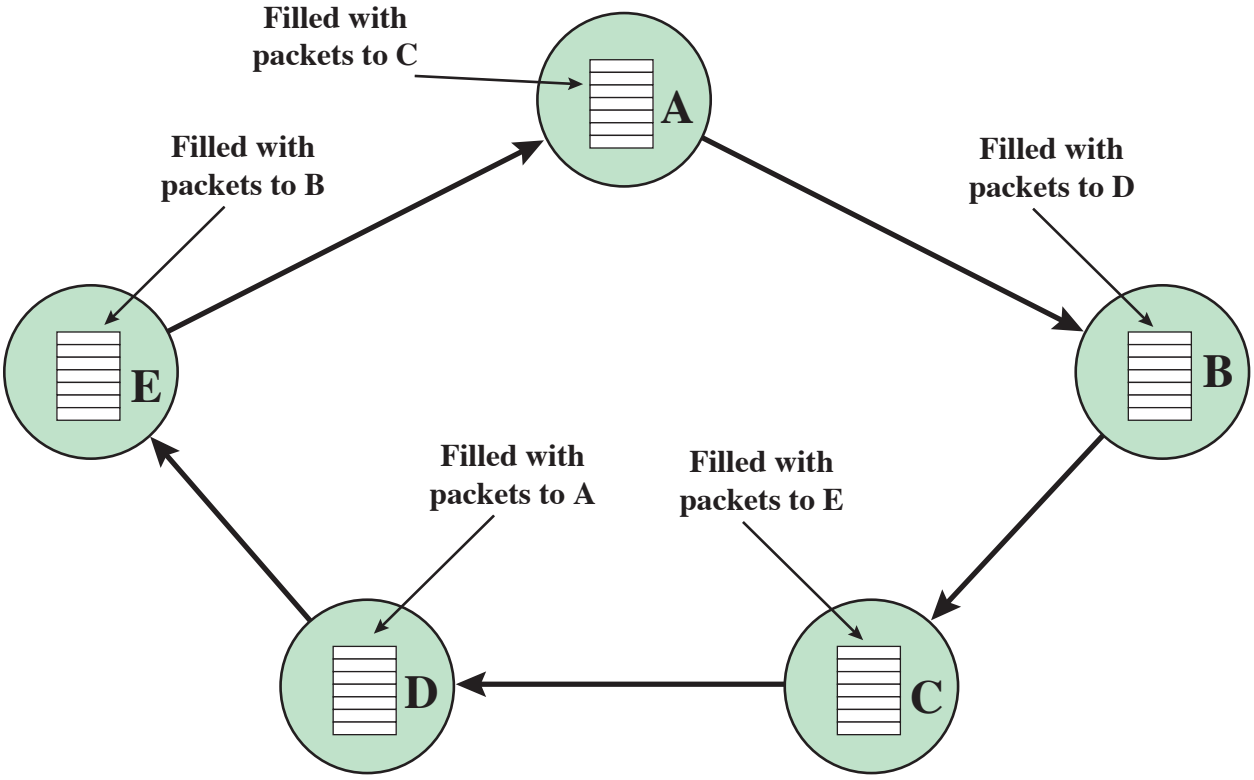
Another way in which deadlock can occur in a message-passing system has to do with the allocation of buffers for the storage of messages in transit. This kind of deadlock is well known in packet-switching data networks. We first examine this problem in the context of a data network and then view it from the point of view of a distributed operating system.

The simplest form of deadlock in a data network is direct store-and-forward deadlock and can occur if a packet-switching node uses a common buffer pool from which buffers are assigned to packets on demand. Figure 18.17a shows a situation in which all of the buffer space in node A is occupied with packets destined for B. The reverse is true at B. Neither node can accept any more packets because their buffers are full. Thus neither node can transmit or receive on any link.

Direct store-and-forward deadlock can be prevented by not allowing all buffers to end up dedicated to a single link. Using separate fixed-size buffers, one for each link, will achieve this prevention. Even if a common buffer pool is used, deadlock is avoided if no single link is allowed to acquire all of the buffer space.



(a) Direct store-and-forward deadlock



(b) Indirect store-and-forward deadlock

Figure 18.17 Store-and-Forward Deadlock

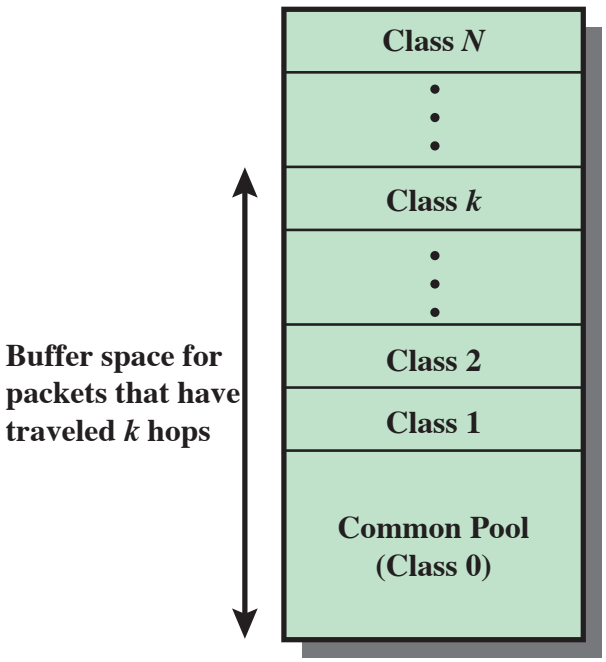


Figure 18.18 Structured Buffer Pool for Deadlock Prevention

A more subtle form of deadlock, indirect store-and-forward deadlock, is illustrated in Figure 18.17b. For each node, the queue to the adjacent node in one direction is full with packets destined for the next node beyond. One simple way to prevent this type of deadlock is to employ a structured buffer pool (Figure 18.18). The buffers are organized in a hierarchical fashion. The pool of memory at level 0 is unrestricted; any incoming packet can be stored there. From level 1 to level *N* (where *N* is the maximum number of hops on any network path), buffers are reserved in the following way: Buffers at level *k* are reserved for packets that have traveled at least *k* hops so far. Thus, in heavy load conditions, buffers fill up progressively from level 0 to level *N*. If all buffers up through level *k* are filled, arriving packets that have covered *k* or less hops are discarded. It can be shown [GOPA85] that this strategy eliminates both direct and indirect store-and-forward deadlocks.

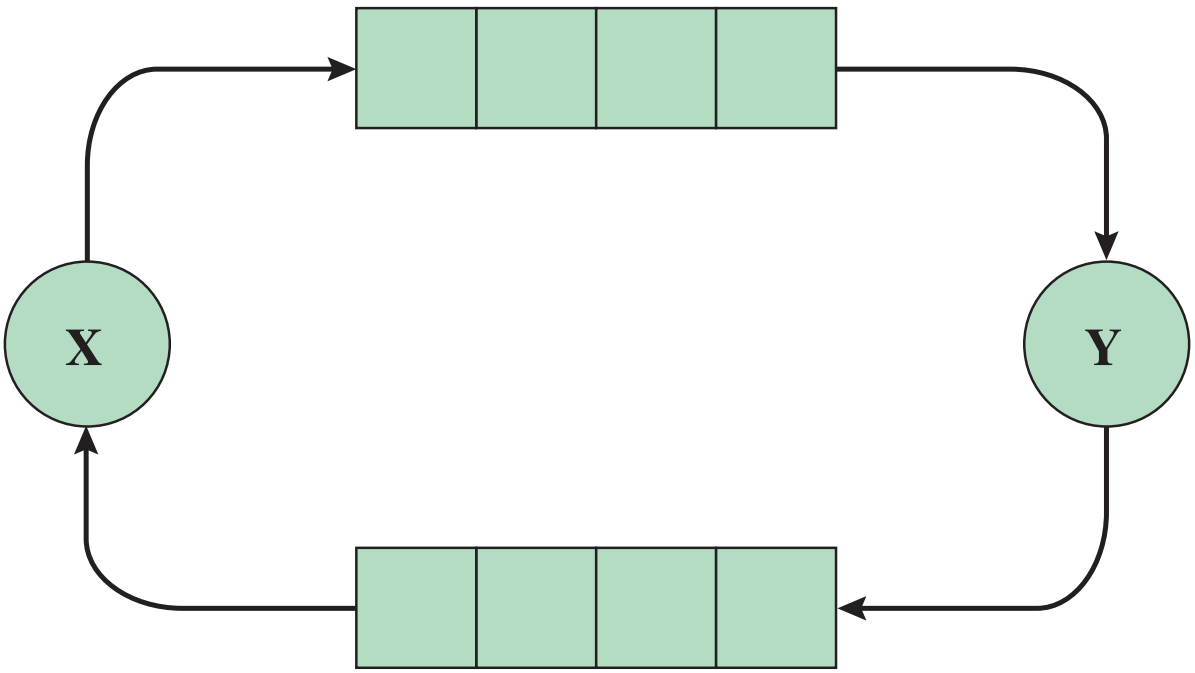


Figure 18.19 Communication Deadlock in a Distributed System

The deadlock problem just described would be dealt with in the context of communications architecture, typically at the network layer. The same sort of problem can arise in a distributed operating system that uses message passing for interprocess communication. Specifically, if the send operation is nonblocking, then a buffer is required to hold outgoing messages. We can think of the buffer used to hold messages to be sent from process X to process Y to be a communications channel between X and Y. If this channel has finite capacity (finite buffer size), then it is possible for the send operation to result in process suspension. That is, if the buffer is of size *n* and there are currently *n* messages in transit (not yet received by the destination process), then the execution of an additional send will block the sending process until a receive has opened up space in the buffer.

Figure 18.19 illustrates how the use of finite channels can lead to deadlock. The figure shows two channels, each with a capacity of four

messages, one from process X to process Y and one from Y to X. If exactly four messages are in transit in each of the channels and both X and Y attempt a further transmission before executing a receive, then both are suspended and a deadlock arises.

If it is possible to establish upper bounds on the number of messages that will ever be in transit between each pair of processes in the system, then the obvious prevention strategy would be to allocate as many buffer slots as needed for all these channels. This might be extremely wasteful and of course requires this foreknowledge. If requirements cannot be known ahead of time, or if allocating based on upper bounds is deemed too wasteful, then some estimation technique is needed to optimize the allocation. It can be shown that this problem is unsolvable in the general case; some heuristic strategies for coping with this situation are suggested in [BARB90].

18.5 SUMMARY

A distributed operating system may support process migration. This is the transfer of a sufficient amount of the state of a process from one machine to another for the process to execute on the target machine. Process migration may be used for load balancing, to improve performance by minimizing communication activity, to increase availability, or to allow processes access to specialized remote facilities.

With a distributed system, it is often important to establish global state information, to resolve contention for resources, and to coordinate processes. Because of the variable and unpredictable time delay in message transmission, care must be taken to assure that different processes agree on the order in which events have occurred.

Process management in a distributed system includes facilities for enforcing mutual exclusion and for taking action to deal with deadlock. In both cases, the problems are more complex than those in a single system.

18.6 RECOMMENDED READING

[GALL00] and [TEL01] cover all of the topics in this chapter.

A broad and detailed survey of process migration mechanisms and implementations is [MILO00]. [ESKI90] and [SMIT88] are other useful surveys. [NUTT94] describes a number of OS implementations of process migration. [FIDG96] surveys a number of approaches to ordering events in distributed systems and concludes that the general approach outlined in this chapter is preferred.

Algorithms for distributed process management (mutual exclusion, deadlock) can be found in [SINH97] and [RAYN88]. More formal treatment are contained in [RAYN90], [GARG02], and [LYNC96].

ESKI90 Eskicioglu, M. "Design Issues of Process Migration Facilities in Distributed Systems." *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems and Application Environments*, Summer 1990.

FIDG96 Fidge, C. "Fundamentals of Distributed System Observation." *IEEE Software*, November 1996.

GALL00 Galli, D. *Distributed Operating Systems: Concepts and Practice.* Upper Saddle River, NJ: Prentice Hall, 2000.

GARG02 Garg, V. *Elements of Distributed Computing.* New York: Wiley, 2002.

LYNC96 Lynch, N. *Distributed Algorithms.* San Francisco, CA: Morgan Kaufmann, 1996.

MILO00 Milojicic, D.; Douglis, F.; Paindaveine, Y.; Wheeler, R.; and Zhou, S. "Process Migration." *ACM Computing Surveys*, September 2000.

NUTT94 Nuttal, M. "A Brief Survey of Systems Providing Process or Object Migration Facilities." *Operating Systems Review*, October 1994.

RAYN88 Raynal, M. *Distributed Algorithms and Protocols.* New York: Wiley, 1988.

RAYN90 Raynal, M., and Helary, J. *Synchronization and Control of Distributed Systems and Programs.* New York: Wiley, 1990.

SINH97 Sinha, P. *Distributed Operating Systems.* Piscataway, NJ: IEEE Press, 1997.

SMIT88 Smith, J. "A Survey of Process Migration Mechanisms." *Operating Systems Review*, July 1988.

TEL01 Tel, G. *Introduction to Distributed Algorithms.* Cambridge: Cambridge University Press, 2001.

18.7 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

channel	eviction	preemptive transfer
distributed deadlock	global state	process migration
distributed mutual exclusion	nonpreemptive transfer	snapshot

Review Questions

18.1 Discuss some of the reasons for implementing process migration.

18.2 How is the process address space handled during process migration?

18.3 What are the motivations for preemptive and nonpreemptive process migration?

18.4 Why is it impossible to determine a true global state?

18.5 What is the difference between distributed mutual exclusion enforced by a centralized algorithm and enforced by a distributed algorithm?

18.6 Define the two types of distributed deadlock.

Problems

18.1 The flushing policy is described in the subsection on process migration strategies in Section 18.1.

- **a.** From the perspective of the source, which other strategy does flushing resemble?
- **b.** From the perspective of the target, which other strategy does flushing resemble?

18.2 For Figure 18.9, it is claimed that all four processes assign an ordering of $\{a, q\}$ to the two messages, even though q arrives before a at P_3. Work through the algorithm to demonstrate the truth of the claim.

18.3 For Lamport's algorithm, are there any circumstances under which P_i can save itself the transmission of a Reply message?

18.4 For the mutual exclusion algorithm of [RICA81],

- **a.** Prove that mutual exclusion is enforced.
- **b.** If messages do not arrive in the order that they are sent, the algorithm does not guarantee that critical sections are executed in the order of their requests. Is starvation possible?

18.5 In the token-passing mutual exclusion algorithm, is the timestamping used to reset clocks and correct drifts, as in the distributed queue algorithms? If not, what is the function of the timestamping?

18.6 For the token-passing mutual exclusion algorithm, prove that it

- **a.** guarantees mutual exclusion
- **b.** avoids deadlock
- **c.** is fair

18.7 In Figure 18.11b, explain why the second line cannot simply read "request (j) = t".