Chapter 8

Resource Allocation

8.1 Introduction

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

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

8.2 Specification of the Mutual Exclusion Problem

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

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

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

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

8.3 Centralized Algorithm

There are many algorithms for mutual exclusion in a distributed system. However, the least expensive algorithm for the mutual exclusion is a centralized algorithm shown in Figure 8.1. If we are required to satisfy just the safety and liveness properties, then this simple queue-based algorithm works. One of the processes is designated as the leader (or the coordinator) for the critical section. The variable `haveToken` is true for the process that has access to the critical section. Any process that wants to enter the critical section sends a *request* message to the leader. The leader simply puts these requests in the `pendingQ` in the order it receives them. It also grants permission to the process that is at the head of the queue by sending an *okay* message. When a process has finished executing its critical section, it sends the *release* message to the leader. On receiving a *release* message, the leader sends the *okay* message to the next process in its `pendingQ` if the queue is nonempty. Otherwise, the leader sets `haveToken` to true.

The centralized algorithm does not satisfy the notion of fairness, which says that requests should be granted in the order they are made and not in the order they are received. Assume that the process P_i makes a request for the shared resource to the leader process P_k. After making the request, P_i sends a message to the process P_j. Now, P_j sends a request to P_k that reaches P_k earlier than the request made by the process P_i. This example shows that it is possible for the order in which requests are received by the leader process to be different from the order in which they are made. The modification of the algorithm to ensure fairness is left as an exercise (see Problem 8.1).

```
P0::
 var
     pendingQ: list of (pid, reqvector) initially null;
     havetoken: boolean initially true only for P0;

 Upon receive(request) from Pj:
     if havetoken then send okay to Pj ;
     else pendingQ.add(j);

 Upon receive(release) from Pj:
     if (!pendingQ.isEmpty())
          remove the first pid from pendingQ and send okay to that process;
     else havetoken := true;
```

Figure 8.1: A centralized algorithm for the coordinator process

8.4 Lamport's Algorithm

In Lamport's algorithm each process maintains a logical clock (used for timestamps) and a queue (used for storing requests for the critical section). The queue is ordered by the timestamps of the requests. The algorithm ensures that processes enter the critical section in the order of timestamps of their requests. It assumes FIFO ordering of messages. The rules of the algorithm shown in Fig. 8.2 are as follows:

- To request the critical section, a process sends a timestamped message to all other processes and adds a timestamped request to the queue.

- On receiving a request message, the request and its timestamp are stored in the queue and a timestamped acknowledgment is sent back.
- To release the critical section, a process sends a release message to all other processes.
- On receiving a release message, the corresponding request is deleted from the queue.
- A process determines that it can access the critical section if and only if its request is at the head of the queue and it has received an acknowledgement message from every other process.

```
P_i::
 var
      q: queue of (int, pid) initially null;
      numAcks: integer initially 0;

 request:
      send request with (logicalClock, i) to all other processes;
      numAcks := 0;

 On receive(request, (ts, j))) from P_j:
      insert (ts, j) in q;

 On receive(u, ack):
      numAcks := numAcks + 1;
      if (numAcks = N − 1) and P_i's request smallest in q then enter_critical_section;

 On receive(u, release) from P_j:
      delete the request by P_j from q
      if (numAcks = N − 1) and P_i's request smallest in q then enter_critical_section;

 release:
      send release to all processes;
```

Figure 8.2: Lamport's algorithm

We show that Lamport's algorithm satisfies mutual exclusion. Suppose, if possible, both P_i and P_j can enter the critical section concurrently. Without loss of generality, assume that P_i has lower timestamp for its request. Since P_j entered the critical section, it must have received an acknowledgement from P_i. If P_i sent this acknowledgement before its own request, then P_i's timestamp for its request must be greater that P_j's request contradicting our assumption that the request by P_i has smaller timestamp. If P_i sent this request before it sent the acknowledgement for P_j's request, then by FIFO property this request must be received by P_j before it received the acknowledgement. In this case, P_j has P_i's request in its queue when it received the acknowledgement and therefore P_j cannot enter the critical section.

The algorithm guarantee starvation freedom because processes enter the critical section in the order of the timestamps of their request.

Lamport's algorithm requires $3(N-1)$ messages per invocation of the critical section: $N-1$ request messages, $N-1$ acknowledgment messages, and $N-1$ release messages. There is a time delay of two serial messages to get permission for the critical section—a request message followed by an acknowledgment.

8.5 Ricart and Agrawala's Algorithm

Ricart and Agrawala's algorithm uses only $2(N-1)$ messages per invocation of the critical section. It does so by combining the functionality of acknowledgment and release messages. In this algorithm, a process does not always send back an acknowledgment on receiving a request. It may defer the reply for a later time. Another advantage of Ricart and Agrawala's algorithm is that it does not require FIFO ordering of messages.

The algorithm is stated by the following rules:

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

The algorithm is presented formally in Figure 8.3. There are two kinds of messages in the system—*request* messages and *okay* messages. Each process maintains the logical time of its request in the variable *myts*. In the method `requestCS`, a process simply broadcasts a *request* message with its timestamp. The variable *numOkay* counts the number of *okay* messages received since the request was made. On receiving any request with a timestamp lower than its own, it replies immediately with *okay*. Otherwise, it adds that process to *pendingQ*.

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

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

For an optimization of message complexity of Ricart and Agrawala's algorithm due to Carvalho and Roucairol, see the Exercise 8.4.

8.6 Dining Philosopher Algorithm

In the previous algorithm, every critical section invocation requires $2(N-1)$ messages. We now show an algorithm in which $2(N-1)$ messages are required only in the worst case. Consider a large distributed

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

 request:
      myts := logical_clock;
      send request with myts to all other processes;
      numOkay := 0;

 On receive(u, request) from P_j:
      if (u.myts, j) < (myts, i)) then
           send okay to process P_j;
      else append(pendingQ, j);

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

 release:
      myts := ∞;
      for j ∈ pendingQ do
           send okay to the process j;
      pendingQ := null;
```

Figure 8.3: Ricart and Agrawala's algorithm

system in which even though N is large, the number of processes that request the critical section, say, n, is small. In our next algorithm, processes that are not interested in the critical section will not be required to send messages eventually.

The next algorithm will also solve a more general problem, the dining philosopher problem, where a resource may not be shared by all the processes. The dining philosopher problem, as discussed in Chapter 3, consists of multiple philosophers who spend their time thinking and eating spaghetti. However, a philosopher requires shared resources, such as forks, to eat spaghetti. We are required to devise a protocol to coordinate access to the shared resources.

There are two requirements on the solution of the dining philosopher problem: (1) we require mutually exclusive use of shared resources, that is, a shared resource should not be used by more than one process at a time; and (2) we want freedom from starvation. Every philosopher (process) should be able to eat (perform its operation) infinitely often.

The crucial problem in resource allocation is that of resolving conflicts. If a set of processes require a resource and only one of them can use it at a time, then there is a conflict that must be resolved in favor of one of these processes. We have already studied one conflict resolution method via logical clocks in Lamport's and Ricart and Agrawala's mutual exclusion algorithms. The processes used logical clocks to resolve access to mutual exclusion. If two requests had the same logical clock value, then process identity was used to break ties. Now we study another mechanism that resolves conflicts based on location of auxiliary resources. The auxiliary resources are used only for conflict resolution and are not actual resources.

We model the problem as an undirected graph called a *conflict graph*, in which each node represents a process and an edge between process P_i and P_j denotes that one or more resources are shared between P_i and P_j. Figure 8.4(a) shows the conflict graph for five philosophers. If a process needs all the shared resources for performing its operation, then only one of any two adjacent nodes can perform its operation in any step. The conflict graph for a simple mutual exclusion algorithm is a complete graph.

Now consider the problem of five dining philosophers sitting around a table such that two adjacent philosophers share a fork. The conflict graph of this problem is a ring on five nodes.

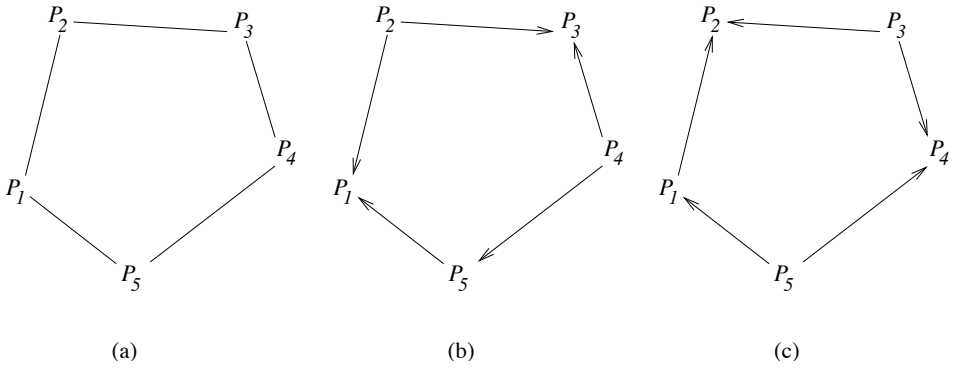

Figure 8.4: (a) Conflict graph; (b) an acyclic orientation with P_2 and P_4 as sources; (c) orientation after P_2 and P_4 finish eating

An *orientation* of an undirected graph consists of providing direction to all edges. The edge between P_i and P_j points from P_i to P_j if P_i has precedence over P_j. We say that an orientation is acyclic if the directed graph that results from the orientation is acyclic. Figure 8.4(b) shows an acyclic orientation of the conflict graph. In a directed graph, we call a node *source* if it does not have any incoming edge. Any finite-directed acyclic graph must have at least one source (see Problem 8.7). In Figure 8.4, processes P_2 and P_4 are sources.

To maintain orientation of an edge, we use the notion of an auxiliary resource, a fork, associated with each edge. Process P_i is considered to have the fork associated with the edge (i, j), if it has precedence over P_j in any conflict resolution.

The algorithm for dining philosophers obeys the following two rules:

- *Eating rule*: A process can eat only if it has all the forks for the edges incident to it.
- *Edge reversal*: On finishing the eating session, a process reverses orientations of all the outgoing edges to incoming edges.

Now let us look at the rules for transmitting forks. We do not require that once a philosopher has finished eating it sends all the forks to its neighbors. This is because its neighbors may be thinking and therefore not interested in eating. Thus we require that if a philosopher is hungry (interested in eating) and does not have the fork, then it should explicitly request the fork. To request the fork, we use a request token associated with each fork. Although a fork is not transmitted after eating, we still need to capture the fact that the other philosopher has priority over this fork to satisfy the edge reversal rule. Thus we need to distinguish the case when a philosopher has a fork but has not used it from the case when the philosopher has the fork and has used it for eating. This is done conveniently by associating a boolean variable *dirty* with each fork. Once a philosopher has eaten from a fork, it becomes dirty. Before a fork is sent to the neighbor, it is cleaned.

Our solution is based on keeping an acyclic conflict resolution graph as mentioned earlier. Philosopher u has priority over philosopher v if the edge between u and v points to v. The direction of the edge is from u to v if (1) u holds the fork and it is clean, (2) v holds the fork and it is dirty, or (3) the fork is in transit from v to u.

The forks are initially placed so that the conflict resolution graph is initially acyclic. The algorithm ensures that the graph stays acyclic. Observe that when a fork is cleaned before it is sent, the conflict graph does not change. The change in the conflict graph occurs only when a philosopher eats, thereby reversing all edges incident to it. The algorithm for the dining philosophers problem is given in Figure 8.5. In this algorithm, we have assumed that the conflict graph is a complete graph for simplicity.

It is easy to see that the conflict resolution graph is always acyclic. It is acyclic initially by our initialization. The only action that changes direction of any edge in the graph is eating (which dirties the fork). A philosopher can eat only when she has all the forks corresponding to the edges that she shares with other philosophers. By the act of eating, all those forks are dirtied and therefore all those edges point toward the philosopher after eating. This transformation cannot create a cycle.

Observe that when a fork is transmitted, it is cleaned before transmission and thus does not result in any change in the conflict resolution graph.

The conflict graph for the mutual exclusion on N processes is a complete graph on N nodes. For any philosopher to eat, she will need to request only those forks that she is missing. This can be at most $N-1$. This results in $2(N-1)$ messages in the worst case. Note that if a process never requests critical section after some time, it will eventually relinquish all its forks and will not be disturbed after that. Thus, the number of messages in the average case is proportional only to the number of processes who are active in accessing the resource.

8.7 Token-Based Algorithms

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

P_i::
var
$hungry, eating, thinking$: boolean;
$fork(f)$: boolean // P_i holds the fork f;
$request(f)$: boolean // P_i holds the request token for the fork f;
$dirty(f)$: boolean // fork f is dirty ;
initially
1. All forks are dirty.
2. Every fork and request token are held by different philosophers.
3. Conflict resolution graph is acyclic.

To request a fork:
if $hungry$ and $request(f)$ and $\neg fork(f)$ **then**
send request token for fork f;
$request(f) := false$;

Releasing a fork:
if $request(f)$ and $\neg eating$ and $dirty(f)$ **then**
$dirty(f) := false$;
$fork(f) := false$;
send fork f;

Upon receiving a request token for fork f:
$request(f) := true$;

Upon receiving a fork f:
$fork(f) := true$;

Figure 8.5: An algorithm for dining philosophers problem

In a token ring approach, all processes are organized in a ring. The token circulates around the ring. Any process that wants to enter the critical section waits for the token to arrive at that process. It then grabs the token and enters the critical section. The algorithm is initiated by the coordinator who sends the token to the next process in the ring. The local state of a process is simply the boolean variable `haveToken` which records whether the process has the token. By ensuring that a process enters the critical section only when it has the token, the algorithm guarantees the safety property trivially.

In a simple version of the algorithm, the token is sent to the next process in the ring after a fixed period of time. The reader is invited to design an algorithm in which the token moves only on receiving a request.

8.8 Quorum-Based Algorithms

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

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

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

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

It is important to recognize that the simple strategy of getting permission to enter the critical section from one of the quorums can result in a deadlock. In the majority voting system, if two requests gather $N/2$ votes each (for an even value of N), then neither of the requests will be granted. Quorum-based systems require additional messages to ensure that the system is deadlock-free. The details of ensuring deadlock freedom are left to the reader (see Problem 8.11).

8.9 Problems

8.1. How will you modify the centralized mutual exclusion algorithm to ensure fairness. (*Hint*: Use vector clocks modified appropriately.)

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

8.3. Assume that N is large. How will you optimize Lamport's algorithm to reduce the space and the computational overhead at each site?

8.4. (due to [CR83]) Show how you can reduce the average message complexity of Ricart and Agrawala's algorithm by requiring a process that wants to enter the critical section to request permission from only those processes to which it has sent `okay` message since the last time it entered the critical section. Give the modification of the algorithm and prove its correctness.

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

8.6. Build a multiuser *Chat* application in Java that ensures that a user can type its message only in its critical section. Ensure that your system handles a dynamic number of users, that is, allows users to join and leave a chat session.

8.7. Show that any finite directed acyclic graph has at least one source.

8.8. When can you combine the request token message with a fork message? With this optimization, show that a philosopher with d neighbors needs to send or receive at most $2d$ messages before making transition from hungry state to eating state.

8.9. Show that the solution to the dining problem does not deny the possibility of simultaneous eating from different forks by different philosophers (when there is no conflict in requirements of forks).

8.10. (due to Suzuki and Kasami [SK85]) Design a token-based algorithm in which any process that needs the token broadcast its request to all processes.

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

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

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

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

8.15. (due to Agrawal and El-Abbadi [AEA91]) In the tree-based quorum system, processes are organized in a rooted binary tree. A quorum in the system is defined recursively to be either the union of the root and a quorum in one of the two subtrees, or the union of quorums of subtrees. Analyze this coterie for availability and load.

8.10 Implementation in Java

8.10.1 Interfaces

We can abstract the mutual exclusion problem as implementation of a lock in a distributed environment. The interface `Lock` is as follows:

```
public interface Lock extends MsgHandler {
    public void requestCS(); //may block
    public void releaseCS();
}
```

Any lock implementation in a distributed environment will also have to handle messages that are used by the algorithm for locking. For this we use the interface `MsgHandler` shown below.

```
import java.io.*; import java.util.*;
public interface MsgHandler {
    public void handleMsg(Msg m, int srcId, String tag);
    public Msg receiveMsg(int fromId) throws IOException;
    public void mySignal();
    public void startListening();
}
```

Any implementation of the lock can be exercised by the program shown in Figure 8.6. It first creates a Linker that links all the processes in the system. After instantiating a lock implementation, it starts separate threads to listen for messages from all the other processes. The class `ListenerThread` is shown in Figure 8.7. A `ListenerThread` is passed a `MsgHandler` on its construction. It makes a blocking `receiveMsg` call at line 12, and on receiving a message gives it to the `MsgHandler` at line 13.

Most of our distributed programs in this book will extend the class `Process`. This will allow processes to have access to its identifier `myId`, the total number of processes `N`, and simple send and receive routines. The method `handleMsg` is empty, and any class that extends `Process` is expected to override this method.

```
1  import java.lang.reflect.*;
2  public class LockTester {
3      public static void main(String[] args) throws Exception {
4              Linker comm = new Linker(args);
5              Class classLoaded = Class.forName(args[0]);
6              Constructor mainCons = classLoaded.getConstructor(Linker.class);
7              Lock lock = (Lock) mainCons.newInstance(comm);
8              lock.startListening();
9              while (true) {
10                 System.out.println(comm.myId + " is not in CS");
11                 Util.mySleep(2000);
12                 lock.requestCS();
13                 Util.mySleep(2000);
14                 System.out.println(comm.myId + " is in CS *****");
15                 lock.releaseCS();
16             }
17     }
18 }
```

Figure 8.6: Testing a lock implementation

```
1  import java.io.*;
2  public class ListenerThread extends Thread {
3      int channel;
4      MsgHandler process;
5      public ListenerThread(int channel, MsgHandler process) {
6          this.channel = channel;
7          this.process = process;
8      }
9      public void run() {
10         while (true) {
11             try {
12                 Msg m = process.receiveMsg(channel);
13                 process.handleMsg(m, m.getSrcId(), m.getTag());
14                 process.mySignal(); // automatic notification after every message
15             } catch (IOException e) {
16                 System.err.println(e);
17             }
18         }
19     }
20 }
```

Figure 8.7: ListenerThread

```
import java.io.*; import java.lang.*;import java.util.*;
public class Process implements MsgHandler {
    int N, myId;
    Linker comm;
    public Process(Linker initComm) {
        comm = initComm;
        myId = comm.getMyId();
        N = comm.getNumProc();
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
    }
    public void sendMsg(int destId, Object ... objects ) {
        Util.println("Sending msg to " + destId );
        comm.sendMsg(destId, Util.getArrayList(objects));
    }
    public void broadcastMsg(String tag, int msg) {
        for (int i = 0; i < N; i++)
            if (i != myId) sendMsg(i, tag, msg);
    }
    public void sendToNeighbors(String tag, int msg) {
        for (int i : comm.neighbors)
             sendMsg(i, tag, msg);
    }
    public void relayToNeighbors(int src, String tag, int msg) {
        for (int i : comm.neighbors)
             if (i != src) sendMsg(i, tag, msg);
    }
    public void multicast(LinkedList<Integer> destIds, String tag, String msg) {
                for (int i : destIds)
                        sendMsg(i, tag, msg);
        }
    public boolean isNeighbor(int i) {
        return (comm.neighbors.contains(i));
    }

    public Msg receiveMsg(int fromId) {
                List<Object> recvdMssage = receiveMsgAsObjectList(fromId);
                return new Msg(fromId, myId, (String) recvdMssage.get(0),
                                (String) recvdMssage.get(1));
    }

    public List<Object> receiveMsgAsObjectList(int fromId){
        return comm.receiveMsg(fromId);
    }
    public synchronized void myWait() {
        try {
            wait();
        } catch (InterruptedException e) {System.err.println(e);
        }
    }
    public synchronized void mySignal() { notifyAll(); }
    public void startListening(){
        for (int i=0; i<N; i++)
            if (i != myId)
                (new ListenerThread(i, this)).start();
    }
    public static void println(String s){System.out.println(s);}

}
```

8.10.2 Centralized Algorithm

```
import java.util.*;
public class CentMutex extends Process implements Lock {
    // assumes that P_0 coordinates and does not request locks.
    boolean haveToken;
    final int leader = 0;
    LinkedList<Integer> pendingQ = new LinkedList<Integer>();
    public CentMutex(Linker initComm) {
        super(initComm);
        haveToken = (myId == leader);
    }
    public synchronized void requestCS() {
        sendMsg(leader, "request");
        while (!haveToken) myWait();
    }
    public synchronized void releaseCS() {
        sendMsg(leader, "release");
        haveToken = false;
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        if (tag.equals("request")) {
            if (haveToken){
                sendMsg(src, "okay");
                haveToken = false;
            }
            else
                pendingQ.add(src);
        } else if (tag.equals("release")) {
            if (!pendingQ.isEmpty()) {
                int pid = pendingQ.removeFirst();
                sendMsg(pid, "okay");
            } else
                haveToken = true;
        } else if (tag.equals("okay"))
            haveToken = true;
    }
}
```

8.10.3 Lamport's Algorithm for Mutual Exclusion

We now give an implementation of a slight variant of Lamport's algorithm for mutual exclusion in Java. In this version, every process maintains two vectors. These two vectors simulate the queue used in the informal description given earlier. These vectors are interpreted at process P_i as follows:

$q[j]$: the timestamp of the request by process P_j. The value `Symbols.infinity` signifies that P_i does not have any record of outstanding request by process P_j.

$v[j]$: the timestamp of the last message seen from P_j if $j \neq i$. The component $s.v[i]$ represents the value of the logical clock in state s. Thus the vector v is simply the direct-dependency clock.

To request the critical section (method `requestCS`), P_i simply records its clock in $q[i]$. Because all other processes also maintain this information, "request" messages are sent to all processes indicating the new value of $q[i]$. It then simply waits for the condition `okayCS` to become true.

To release the critical section (method `releaseCS`), P_i simply resets $q[i]$ to ∞ and sends "release" messages to all processes. Finally, we also require processes to acknowledge any request message as shown

in the method `handleMsg`. Note that every message is timestamped and when it is received, the vector v is updated according to the direct-dependency clock rules as discussed in Chapter 7.

Process P_i has permission to access the critical section when there is a request from P_i with its timestamp less than all other requests and P_i has received a message from every other process with a timestamp greater than the timestamp of its own request. Since two requests may have identical timestamps, we extend the set of timestamps to a total order using process identifiers as discussed in Chapter 7. Thus, if two requests have the same timestamp, then the request by the process with the smaller process number is considered smaller. Formally, P_i can enter the critical section if

$$\forall j : j \neq i : (q[i], i) < (v[j], j) \ \wedge \ (q[i], i) < (q[j], j)$$

This condition is checked in the method `okayCS`.

```
public class LamportMutex extends Process implements Lock {
    public DirectClock v;
    public int[] q; // request queue
    public LamportMutex(Linker initComm) {
        super(initComm);
        v = new DirectClock(N, myId);
        q = new int[N];
        for (int j = 0; j < N; j++)
            q[j] = Integer.MAX_VALUE; // infinity
    }
    public synchronized void requestCS() {
        v.tick();
        q[myId] = v.getValue(myId);
        broadcastMsg("request", q[myId]);
        while (!okayCS())
            myWait();
    }
    public synchronized void releaseCS() {
        q[myId] =  Integer.MAX_VALUE; // infinity
        broadcastMsg("release", v.getValue(myId));
    }
    public Boolean okayCS() {
        for (int j = 0; j < N; j++){
            if (isGreater(q[myId], myId, q[j], j))
                return false;
            if (isGreater(q[myId], myId, v.getValue(j), j))
                return false;
        }
        return true;
    }
    boolean isGreater(int entry1, int pid1, int entry2, int pid2) {
        return ((entry1 > entry2)
                || ((entry1 == entry2) && (pid1 > pid2)));
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        int timeStamp = m.getMessageInt();
        v.receiveAction(src, timeStamp);
        if (tag.equals("request")) {
            q[src] = timeStamp;
            sendMsg(src, "ack", v.getValue(myId));
        } else if (tag.equals("release"))
            q[src] = Integer.MAX_VALUE;
    }
}
```

8.10.4 Ricart and Agrawala's Algorithm

```
import java.util.*;
public class RAMutex extends Process implements Lock {
    public int myts;
    public LamportClock c = new LamportClock();
    LinkedList<Integer> pendingQ = new LinkedList<Integer>();
    public int numOkay = 0;
    public RAMutex(Linker initComm) {
        super(initComm);
        myts = Integer.MAX_VALUE;
    }
    public synchronized void requestCS() {
        c.tick();
        myts = c.getValue();
        broadcastMsg("request", myts);
        numOkay = 0;
        while (numOkay < N-1)
            myWait();
    }
    public synchronized void releaseCS() {
        myts = Integer.MAX_VALUE;
        while (!pendingQ.isEmpty())
            sendMsg(pendingQ.remove(), "okay", c.getValue());
    }
    public Boolean okayCS() {
        if(myts == Integer.MAX_VALUE || numOkay <N-1) return false;
        return true;
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        int timeStamp = m.getMessageInt();
        c.receiveAction(src, timeStamp);
        if (tag.equals("request")) {
            if ((timeStamp < myts) || ((timeStamp == myts) && (src < myId)))
                sendMsg(src, "okay", c.getValue());
            else
                pendingQ.add(src);
        } else if (tag.equals("okay"))
            numOkay++;
    }
}
```

8.10.5 Dining Philosopher Algorithm

```
public class DinMutex extends Process implements Lock {
    private static final int thinking = 0, hungry = 1, eating = 2;
    Boolean fork[] = null,  dirty[] = null, request[] = null;
    public int myState = thinking;
    public DinMutex(Linker initComm) {
        super(initComm);
        fork = new Boolean[N]; dirty = new Boolean[N];
        request = new Boolean[N];
        for (int i : comm.neighbors) {
            if (myId > i) {
                fork[i] = false; request[i] = true;
            } else { fork[i] = true; request[i] = false; }
            dirty[i] = true;
        }
    }
    public synchronized void requestCS() {
        myState = hungry;
        if (haveForks()) myState = eating;
        else
            for (int i: comm.neighbors)
                if (request[i] && !fork[i])
                    sendBool(i, "Request", request[i]);
        while (myState != eating) myWait();
    }
    public synchronized void releaseCS() {
        myState = thinking;
        for (int i: comm.neighbors){
            dirty[i] = true;
            if (request[i]) sendBool(i, "Fork", fork[i]);
        }
    }
    boolean haveForks() {
        for (int i: comm.neighbors)
            if (!fork[i]) return false;
        return true;
    }
    void sendBool(int dest, String tag, Boolean b) {
        sendMsg(dest, tag);
        b = false;
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        if (tag.equals("Request")) {
            request[src] = true;
            if ((myState != eating) && fork[src] && dirty[src]) {
                sendBool(src, "Fork", fork[src]);
                if (myState == hungry)
                    sendBool(src, "Request", request[src]);
            }
        } else if (tag.equals("Fork")) {
            fork[src] = true; dirty[src] = false;
            if (haveForks())
                myState = eating;
        }
    }
}
```

8.11 Bibliographic Remarks

Lamport's algorithm for mutual exclusion [Lam78] was initially presented as an application of logical clocks. The number of messages per invocation of the critical section in Lamport's algorithm can be reduced as shown by Ricart and Agrawala [RA81]. The token-based algorithm can be decentralized as shown by Suzuki and Kasami [SK85]. The tree-based algorithm in the problem set is due to Raymond [Ray89]. The use of majority voting systems for distributed control is due to Thomas [Tho79], and the use of weighted voting systems with R and W parameters is due to Gifford [Gif79]. Maekawa [Mae85] introduced grid-based quorums and quorums based on finite projective planes. The tree-based quorum in the problem set is due to Agrawal and El-Abbadi [AEA91]. The triangular quorum systems are due to Lovasz [Lov73]. The notion of crumbling walls is due to Peleg and Wool [PW95].