CIS 520, *Operating Systems Concepts*

Lecture 10

Inter-process Communications and Synchronization in Distributed Systems

Outline

- Interprocess Communications (message passing)

- Problems with a faulty channel

- Equivalence of semaphores and message passing

- A few notes on mutual exclusion mechanisms in distributed systems

Remember Semaphores?

- They serve two functions:
  - Provision of mutual exclusion
  - Synchronization
- They assume that the memory is shared among CPUs

There is a mechanism, which is much better conceptually (although technically equivalent)

Message Passing

◆ A source process issues
  `send(destination_process,`
          `&message)`

◆ A destination processes issues
  `receive(source_process,`
              `&message)`

◆ Observations:
  ▪ Both look pretty much like an I/O request
  ▪ *receive* is obviously blocking
  ▪ *send* may (but does not have to) be blocking (why?)

Major Issues

- *Acknowledgement:* If processes run on the same machine, we take it for granted that a message that was sent will be received—but this is not true if the processes are on different machines (see next slide). And what to do if the acknowledgement is not received? Retransmit? Forget it?

- *Authentication*: How can the destination process know for sure that the source process is what the message says it is. (That could be questionable even on a single machine—although it is easier to get something done about that in that case; it is a major problem for the machines connected through Internet.)

- *Naming:* A process ID is significant within a particular operating system running on a particular machine—what happens when more than one machine is involved? How to create uniform naming?

Acknowledgments:
The War of Blue and Red Armies

$\frac{1}{2}$

$\frac{1}{2}$

Attack tomorrow at 4 AM!

1
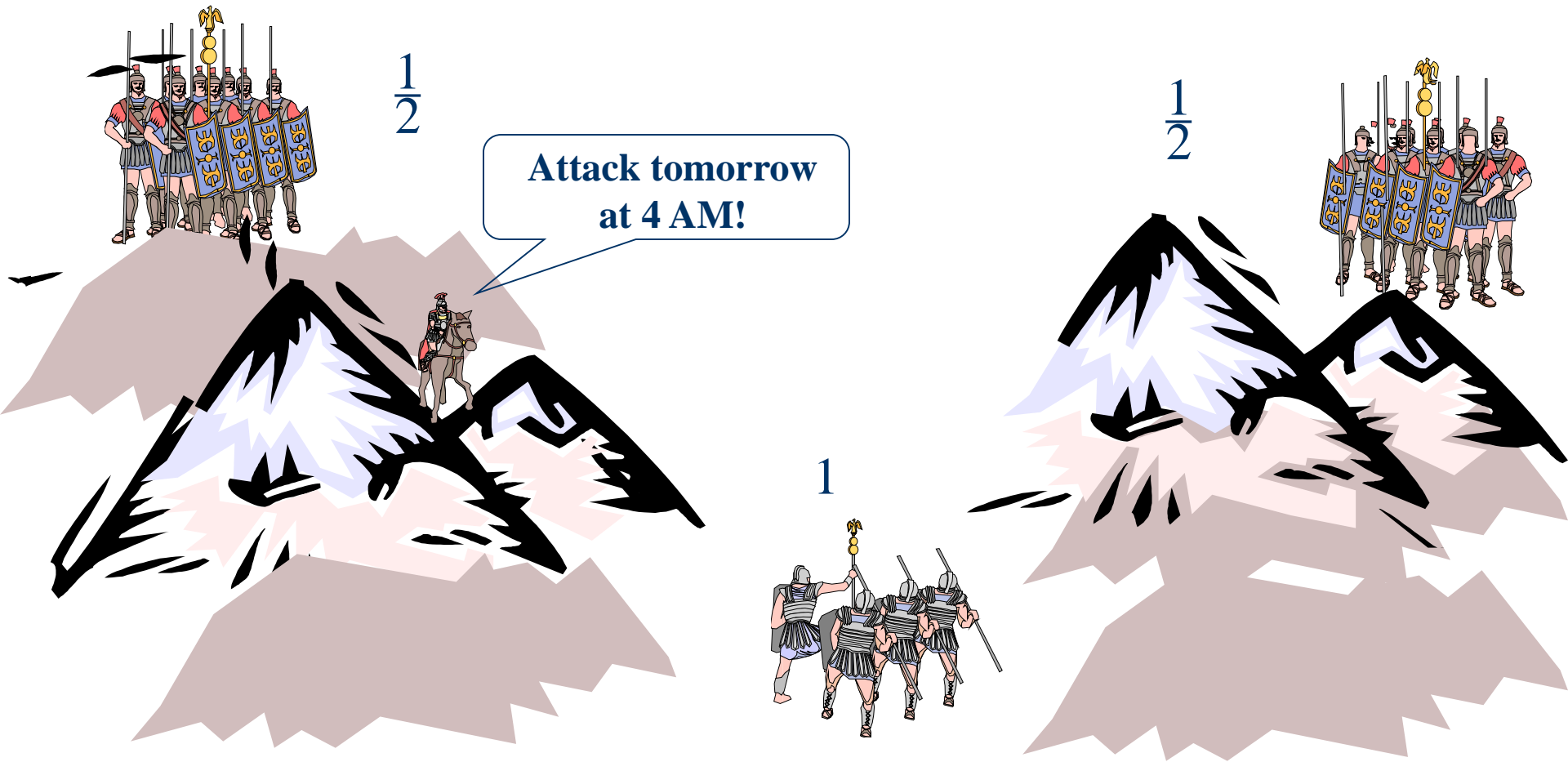
A Bit More General Result

In presence of a faulty communication channel between them, it is *impossible* for two processes *A* and *B* to compute something together

Proof: Suppose they can, and there is a minimal set of messages *M* that need to be exchanged in support of the computation. Let *m* (sent, say, from *A* to B) be the last message in this sequence. Since *A* does not know whether *B* will receive it (but it computed anyway!), *m* is redundant. But then *M* is not a *minimal* sequence—a contradiction.

Stepping Back

◆ Let us, for now, assume that the channel is not faulty (we can simply assume that the processes are on the same machine)

◆ Let us show then that a properly implemented package {*send, receive*} is equivalent to a semaphore package

◆ "Properly" means that

1. The messages are queued (so that none is lost)

2. When the receiving queue reaches its maximum size, the sending process is suspended

• (A simple variation on the latter scheme is to have each process issuing *send* blocked until the corresponding *receive* is issued. This scheme is much easier to implement; it is called *rendez-vous*. It is not as flexible as the queuing scheme)

Working with Mailboxes

- A mechanism similar to that used by the post system
- To *create* a mailbox in which messages to *A* are collected, we define a function that returns the mailbox address

```
box_A = create_mailbox(max_size, A_id)
```

- The OS also provides a function to be called by a process that needs to communicate with *A:*

```
box_A = get_mailbox_address(A_id);
```

- The mailbox has a queue (of the size specified at creation) and a name. Then the primitives work like this:

```
send(box_A,&message)  /* by any process */
receive(box_A,&message)  /* by A */
```

- The operation fails (or, alternatively, the sending process is blocked)when *max_size* is reached
- *A* is blocked on *receive* when *box_A* is empty

Semaphores (a reminder)

```
void wait (semaphore S)

    {
      S.value--;
      if (S.value < 0)
        queue_and_block(S.queue);
    }


  void signal (semaphore S)

    {
      S.value++;
      if (S.value < 0)
        advance_queue(S.queue);
    }
```

Using Message Passing to Implement Semaphores

◆ All mailboxes allow only one message before blocking

◆ First, we develop the synchronization process, *S,* which creates semaphores and keeps their counters and process's mailbox name queues

◆ To create a semaphore, a process sends the *create* message to *S,* and issues the *receive* to get the *acknowledgement* of this action

◆ When a process calls *signal*, first the corresponding message is sent to the synchronization process, then a *receive* is issued; the synchronization process should respond with an empty message acknowledgement as soon as possible

◆ When a process calls *wait,* first it sends the *wait* command to the synchronization process, and then it again issues the *receive*. But the synchronization process will respond *only* when the semaphore value is non-negative

How it works

send "create" or "signal" or "wait" and wait for "Ack"

send "ACK"
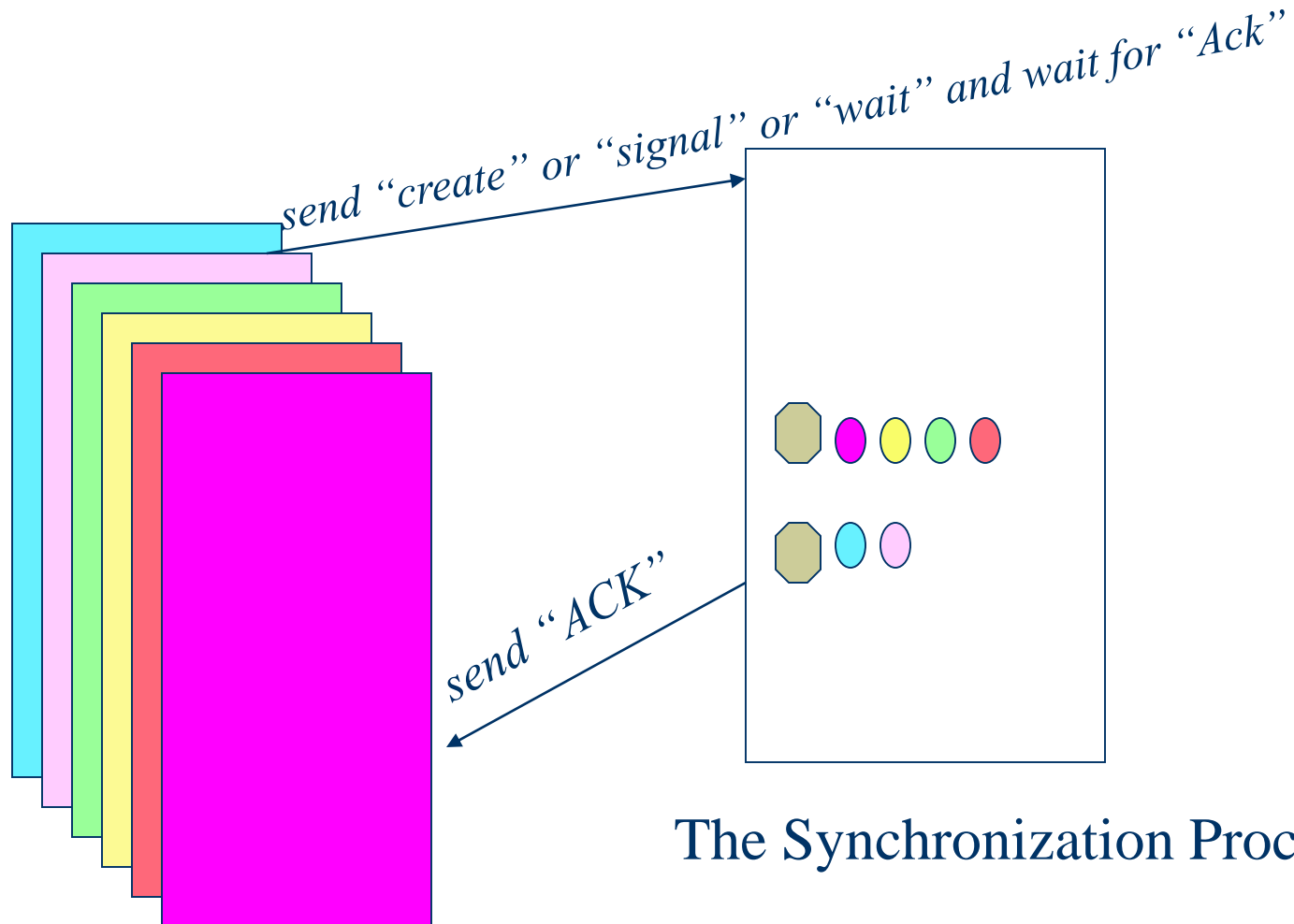
The Synchronization Process

The Library

```
void wait (semaphore X, mbox S)

    {
      send   (S, "wait");
      receive(my_box, &ack);
    }


  void signal (semaphore X, mbox S)

    {
      send   (S, "signal");
      receive(my_box, &ack);
    }
```
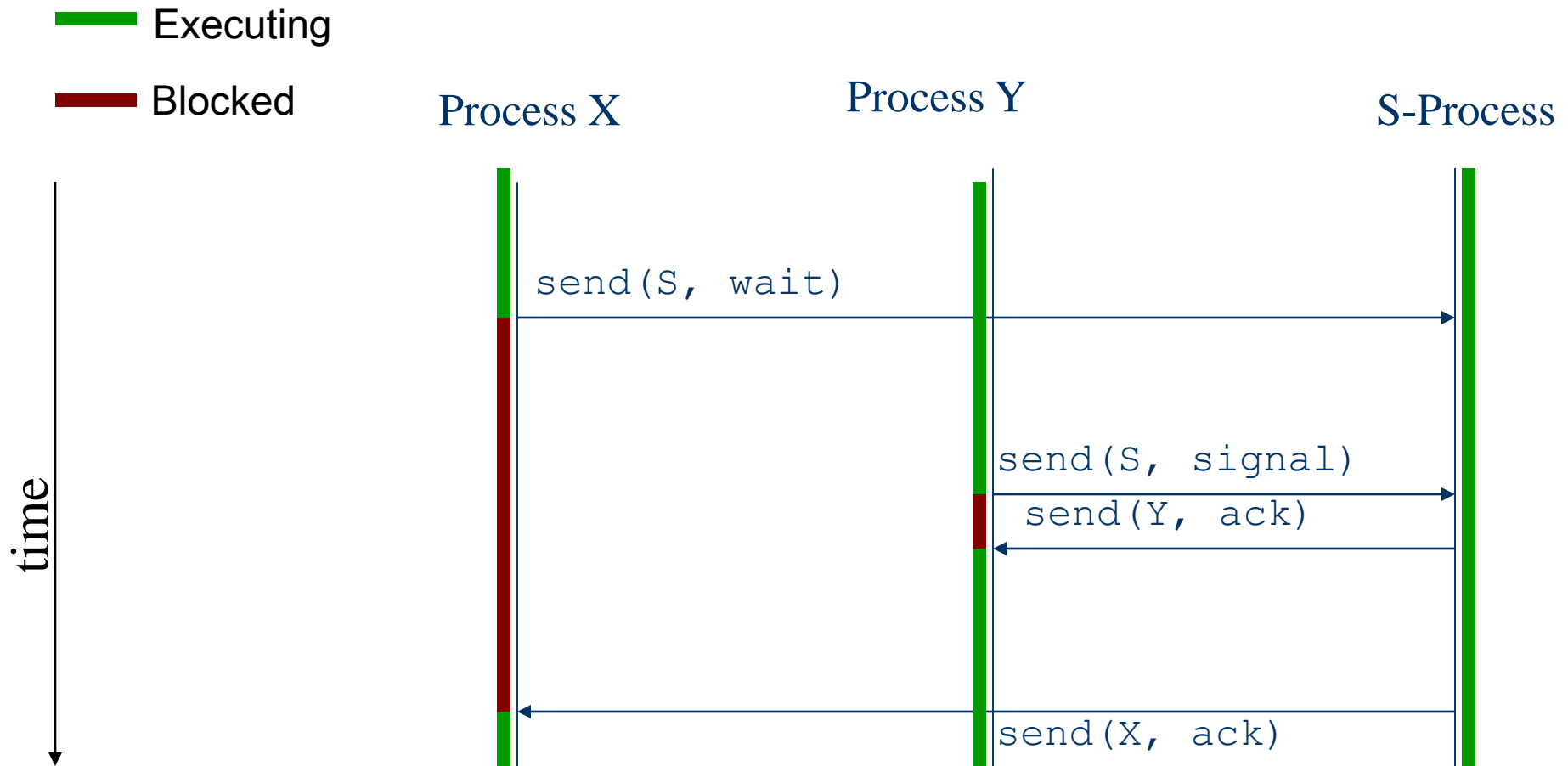
The Synchronization Process

```
main ()

  { do {
    receive (S_box, &command)

     command = instruction|X;

     switch (instruction)
        {
        case create:
          create a semaphore;
          send (sender_box, ack);
        break;

        case wait:
          X.value--;
          if (X.value < 0)
            enter(X, X.queue)
          else
            send (X, ack);
          break;

        case signal:
          send(X, ack);
          X.value++;
          if non-empty (X.queue)
              {
                send( first(X.queue), ack)
                advance_queue (X.queue);
              }

          break
        }

  }    while TRUE;
```

An Illustration

Executing
Blocked

Process X   Process Y   S-Process

send(S, wait)

send(S, signal)
send(Y, ack)

time

send(X, ack)

One Big Problem...

- This implementation involves a central entity—a synchronization process
- Thus, the solution is not fully distributed (even though it can be easily implemented in a distributed environment)
- But our goal—at the moment—is not to find a distributed implementation, but to prove the equivalence of semaphores and message passing

So far, we have proven that semaphores can be implemented with message passing

Converse: Using Semaphores to Implement Message Passing

◆ First, each process has a semaphore, on which it blocks when a *send* or *receive* must wait for completion (initial value is *0*)

◆ Mailboxes (linked lists of message slots) are maintained in the shared buffer area protected by a binary semaphore *mutex*

◆ Then two queues are maintained

1. A queue of processes that cannot send to the mailbox (because it is full) and

2. A queue of processes that cannot receive (because it is empty)

Distributed Systems

Interconnected Networks
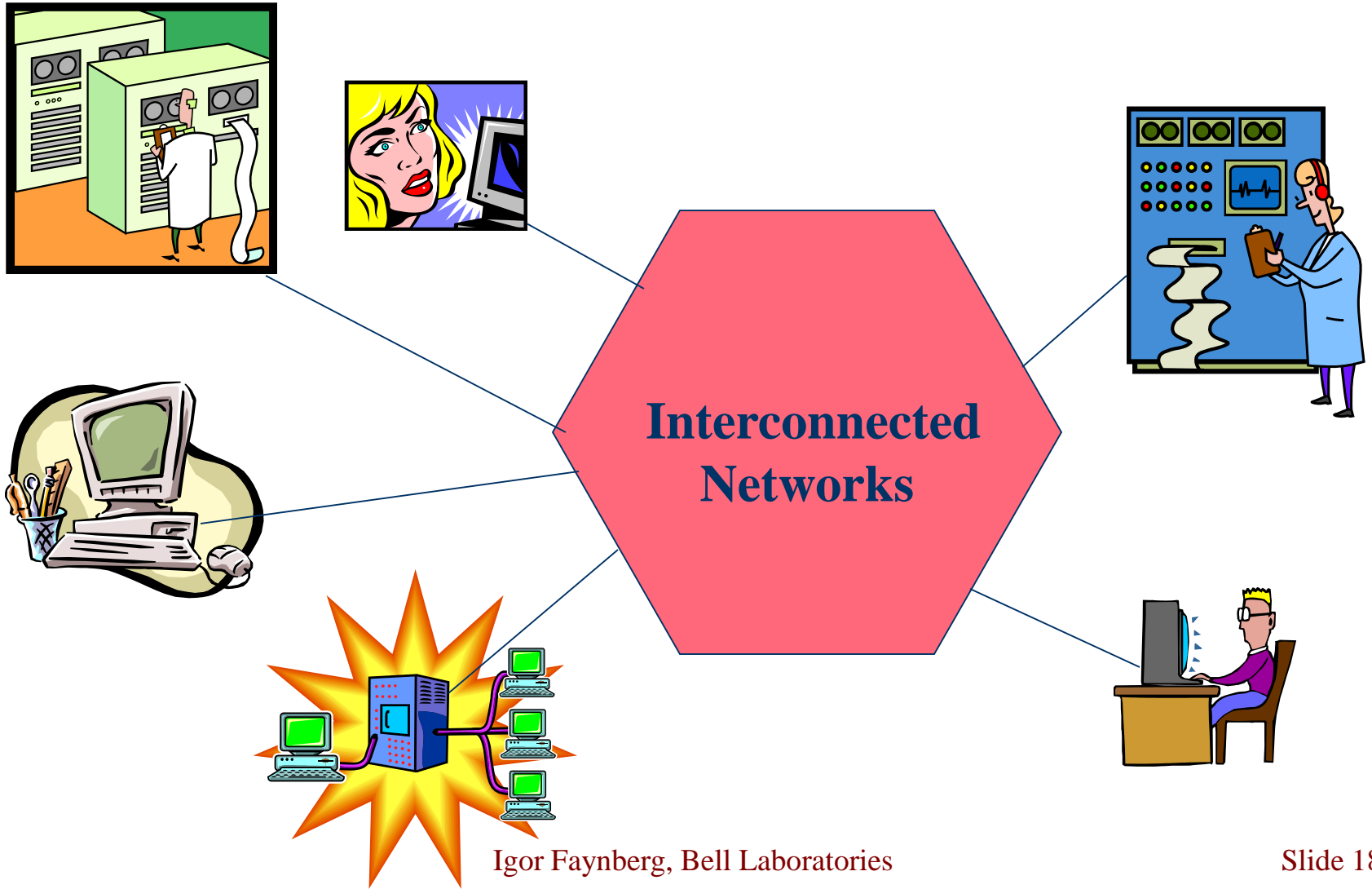
Synchronization in Distributed Systems

◆ The information is scattered among multiple machines, but processes make decisions based *only* on locally available information

◆ A single point of failure is unacceptable

◆ No common clock exists

◆ The *current time* is subject to *agreement* among the processes—*logical clocks* rather than physical clocks are used

Synchronization

A *happens-before* relation is defined as follows:

- If the events a and b took place in the same process, and a took place before b, then $a \rightarrow b$
- If a is the event that took place when a process send a message to another process, and b is the event that took place when that message was received then $a \rightarrow b$

This relation is transitive: $(a \rightarrow b, b \rightarrow c) \Rightarrow a \rightarrow c$

This relation is not defined for *all* event pairs; the events for which it remains undefined are called *concurrent* (not that it says much...)

Lamport's Algorithm for Assigning Time to Events: the Total Order

◆Each sending process attaches the local time value to the message carrying the event

◆The receiver changes the clock to the received value plus one tick (*only* if the local time when the message was received is *less than* what is indicated in the time stamp)

(If a message was sent at *30*, and received at *23* [local time], the receiver adjusts its clock to *31*)

Adjusting Time

- Lamport's algorithm allows the ordering of network events rather than actually assign them correct time
- Assigning the correct time can be done using a passive *time server* by employing an iterative algorithm (*Christian's Algorithm*)
- It can also be done using averaging by the active time server (*Berkeley Algorithm*)
- Or it can be done using a decentralized (truly distributed) algorithm with statistical analysis and averaging of multiple responses to a broadcast time request
- Recently, GPS system has been used for clock synchronization (three satellites for geometric position; the fourth, for time synchronization).

Mutual Exclusion

◆ Mutual exclusion can be implemented with a synchronization process running on a dedicated machine—but this is *not* a distributed solution since the machine on which it is running is a single point of failure

Synchronization Process
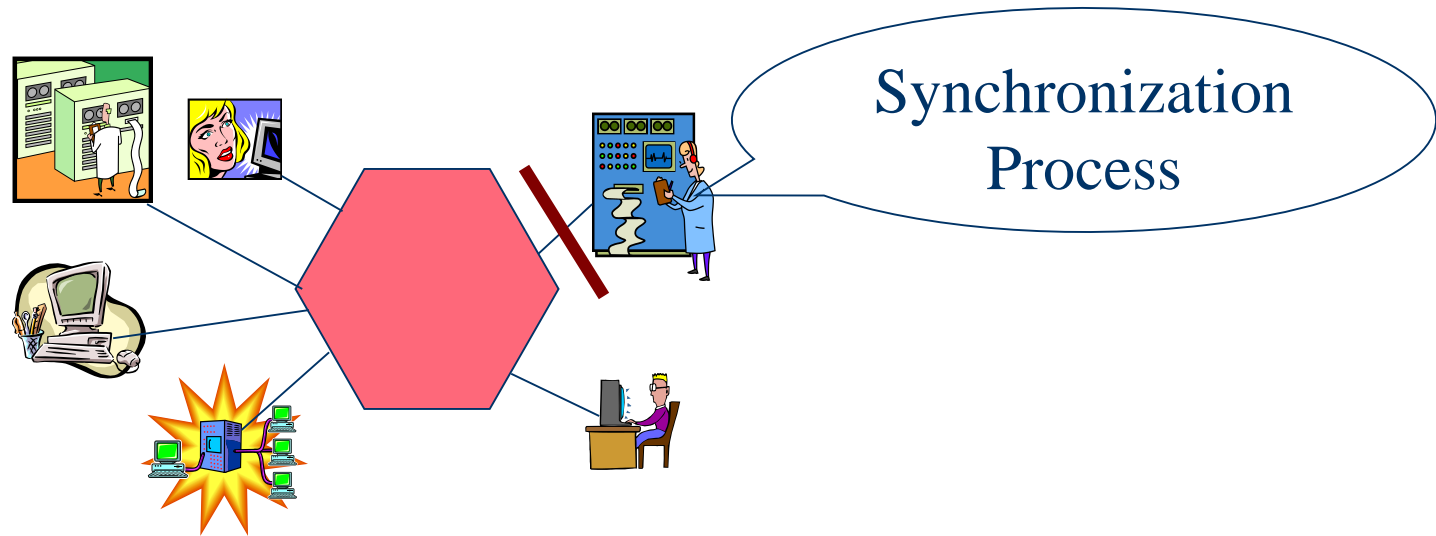
Mutual Exclusion— Ricart and Agrawala's Algorithm

◆ <u>Assumptions:</u>
  - Total ordering of the events
  - Non-faulty communications channels
  - Knowledge of *all* processes in the system
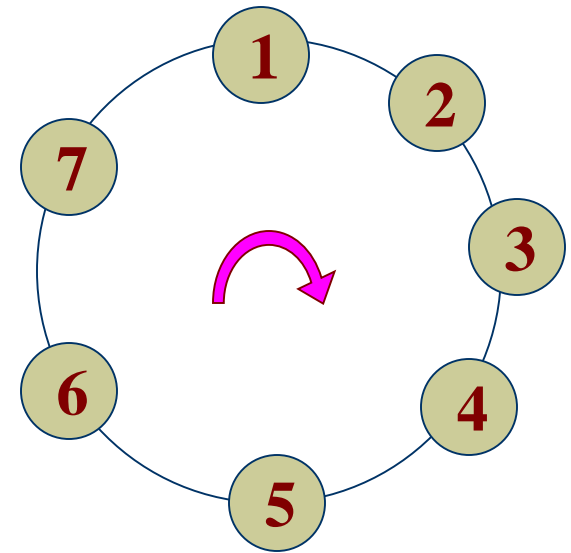
◆ <u>Algorithm:</u>
  - When a process wants to enter a critical section, it broadcasts its intent
    • If the receiver is not in a critical section and is not planning on entering it at the moment, it sends back a "go ahead" message
    • If the receiver wants to enter the critical section, but has not done so yet, it compares the timestamp of the message it sent with the one it just received—the earliest wins because the latest sends the "go ahead" message
    • If the receiver *is* in a critical section, it does *not* respond; it just queues the request
  - When the process exits the critical section, it broadcasts "go ahead" to all wannabes in its queue

A Problem: Multiple-point failure (what if a process crashes and never responds?)

A Token Ring Algorithm

- The processes are organized in a ring (each process knows who is *next*)
- When the ring is initialized, the first process is given a token
- If a process has a token, it either enters a critical section or it must pass the token to the next process
- When a process leaves the critical section, it passes the token to the next process

Problems: Multiple-point failure (a process crashes, a link is down, a token is lost)

A Comparison

Algorithm	Number of messages per entry	Delay before entry	Problems
Centralized	3	2	Single-point failure
Ricart & Agrawala	$2(n-1)$	$2(n-1)$	Multiple-point failure
Token Ring	1 to ∞	0 to $n-1$	Multiple-point failure