Distributed Systems
Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science
Room R4.20, steen@cs.vu.nl

Chapter 06: Synchronization

Version: November 19, 2012

Clock Synchronization

- Physical clocks
- Logical clocks
- Vector clocks

Physical clocks

Problem

Sometimes we simply need the exact time, not just an ordering.

Solution

Universal Coordinated Time (UTC):

- Based on the number of transitions per second of the cesium 133 atom (pretty accurate).
- At present, the real time is taken as the average of some 50 cesium-clocks around the world.
- Introduces a leap second from time to time to compensate that days are getting longer.

Note

UTC is broadcast through short wave radio and satellite. Satellites can give an accuracy of about ±0.5 ms.

Physical clocks

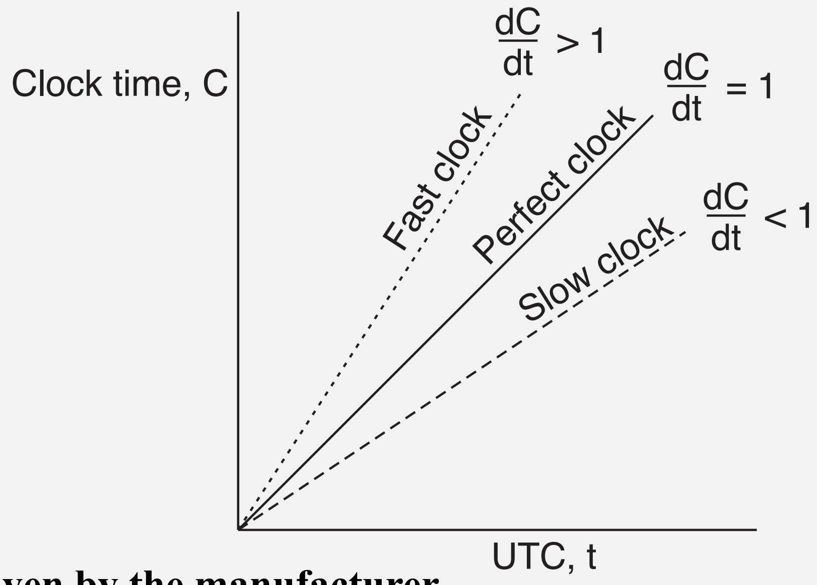
Problem

Suppose we have a distributed system with a UTC-receiver somewhere in it $\Rightarrow$ we still have to distribute its time to each machine.

Basic principle

- Every machine has a timer that generates an interrupt H times per second.
- There is a clock in machine p that ticks on each timer interrupt. Denote the value of that clock by $C_p(t)$, where t is UTC time.
- Ideally, we have that for each machine p, $C_p(t) = t$, or, in other words, $dC/dt = 1$.

Physical clocks



ρ is the maximum drift rate given by the manufacturer

In practice: $1-\rho \leq \frac{dC}{dt} \leq 1+\rho$.

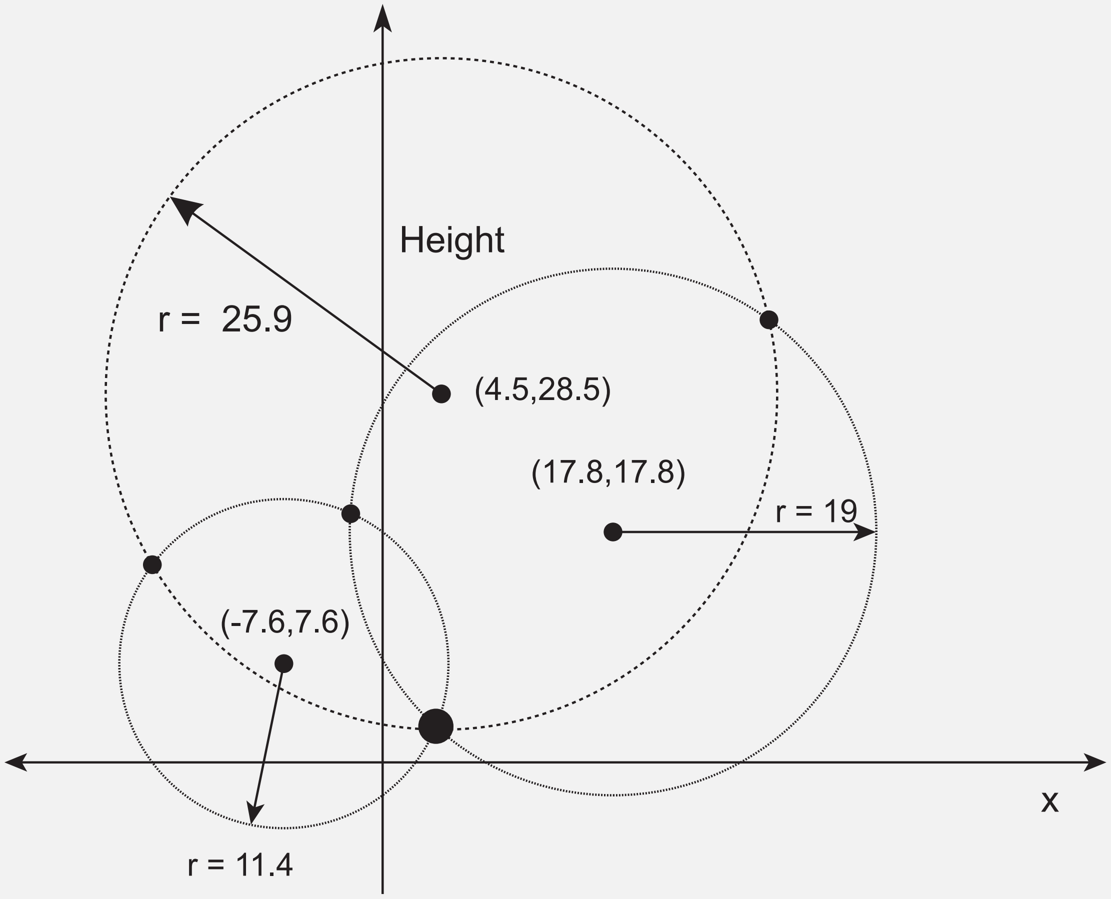
Goal

Never let two clocks in any system differ by more than δ time units $\Rightarrow$ synchronize at least every $\delta/(2\rho)$ seconds.

Global positioning system

Basic idea

You can get an accurate account of time as a side-effect of GPS.



Global positioning system

Problem

Assuming that the clocks of the satellites are accurate and synchronized:

- It takes a while before a signal reaches the receiver
- The receiver's clock is definitely out of synch with the satellite

Global positioning system

Principal operation

- Δ_r: unknown deviation of the receiver's clock.
- x_r, y_r, z_r: unknown coordinates of the receiver.
- T_i: timestamp on a message from satellite i
- $\Delta_i = (T_{now} - T_i) + \Delta_r$: measured delay of the message sent by satellite i.
- Measured distance to satellite i: $c \times \Delta_i$
  (c is speed of light)
- Real distance is

$$d_i = c\Delta_i - c\Delta_r = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

Observation

4 satellites ⇒ 4 equations in 4 unknowns (with Δ_r as one of them)

Clock synchronization principles

Principle I

Every machine asks a time server for the accurate time at least once every $\delta/(2\rho)$ seconds (Network Time Protocol).

Note

Okay, but you need an accurate measure of round trip delay, including interrupt handling and processing incoming messages.

Clock synchronization principles

Principle II

Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time.

Note

Okay, you'll probably get every machine in sync. You don't even need to propagate UTC time.

Fundamental

You'll have to take into account that setting the time back is never allowed ⇒ smooth adjustments.

The Happened-before relationship

Problem

We first need to introduce a notion of ordering before we can order anything.

The happened-before relation

- If a and b are two events in the same process, and a comes before b, then $a \rightarrow b$.
- If a is the sending of a message, and b is the receipt of that message, then $a \rightarrow b$
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

Note

This introduces a partial ordering of events in a system with concurrently operating processes.

Logical clocks

Problem

How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Solution

Attach a timestamp $C(e)$ to each event e, satisfying the following properties:

P1 If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.

P2 If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

Problem

How to attach a timestamp to an event when there's no global clock $\Rightarrow$ maintain a consistent set of logical clocks, one per process.

Logical clocks

Solution

Each process P_i maintains a local counter C_i and adjusts this counter according to the following rules:

1: For any two successive events that take place within P_i, C_i is incremented by 1.

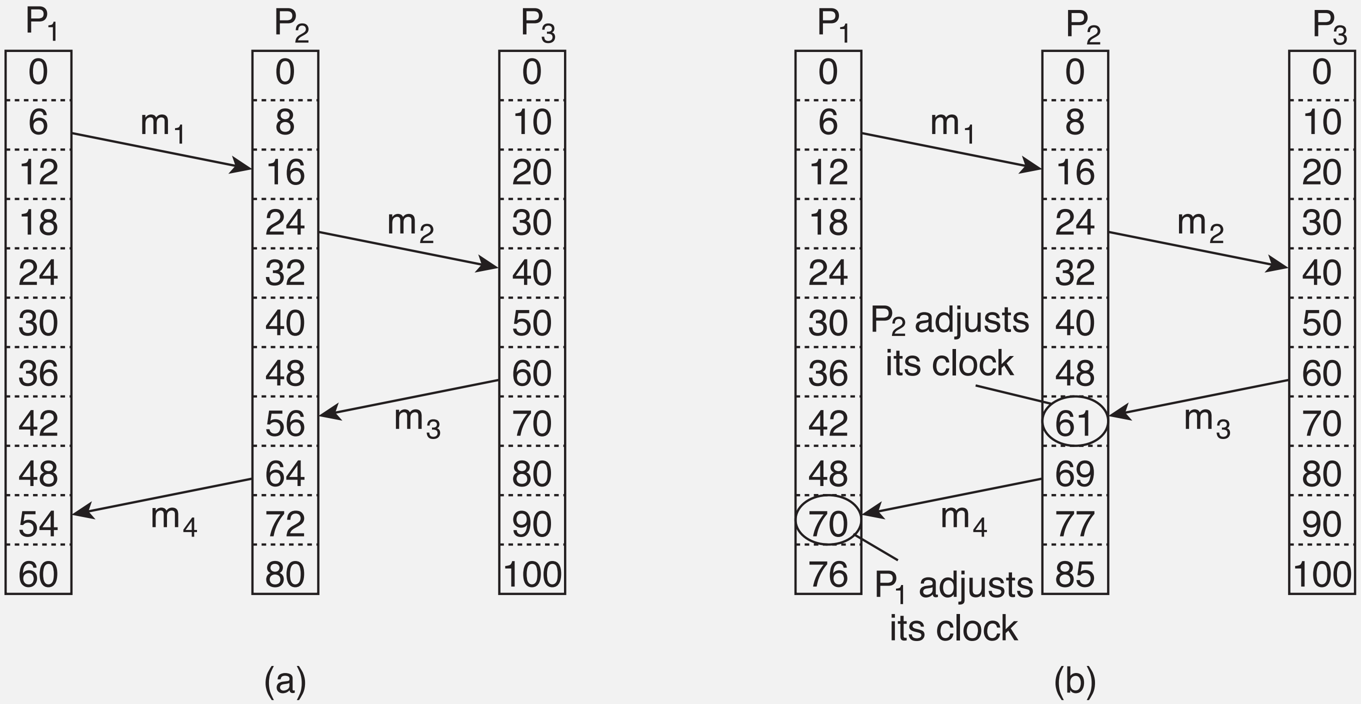
2: Each time a message m is sent by process P_i, the message receives a timestamp $ts(m) = C_i$.

3: Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

Notes

- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

Logical clocks – example



P_1	P_2	P_3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	56	70
48	64	80
54	72	90
60	80	100

(a)

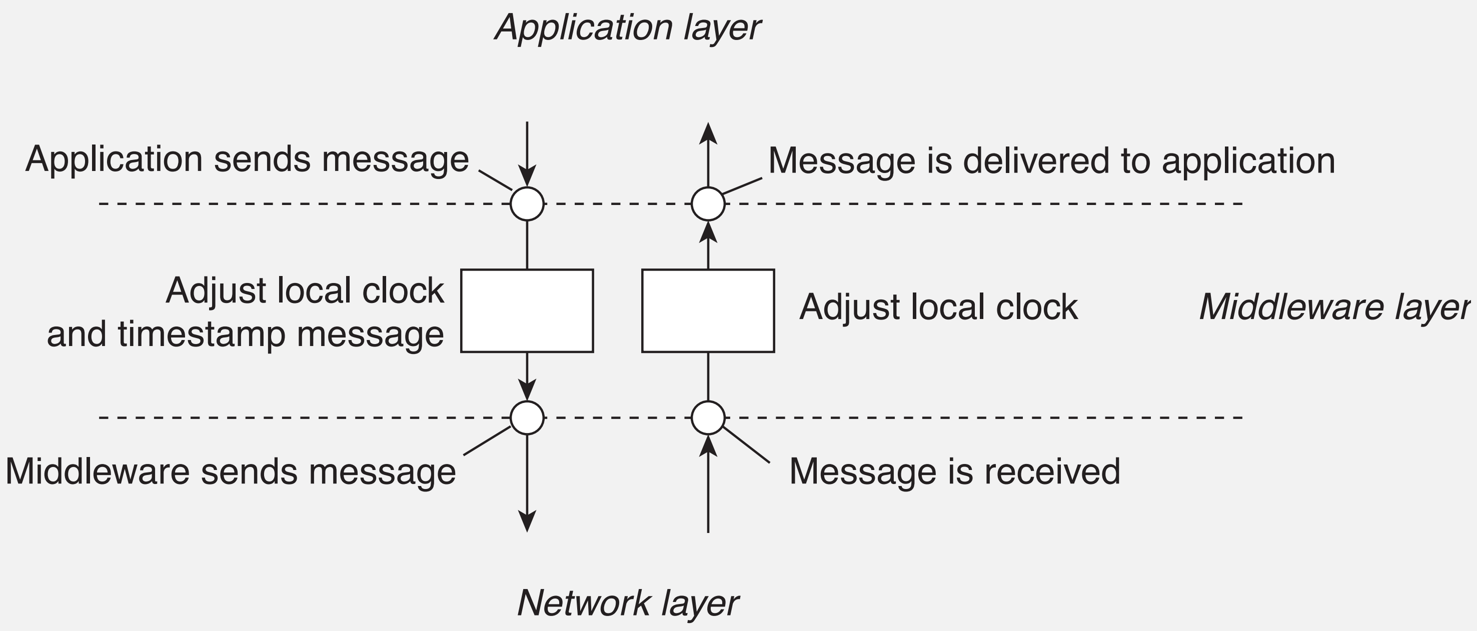
P_1	P_2	P_3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

(b)

Logical clocks – example

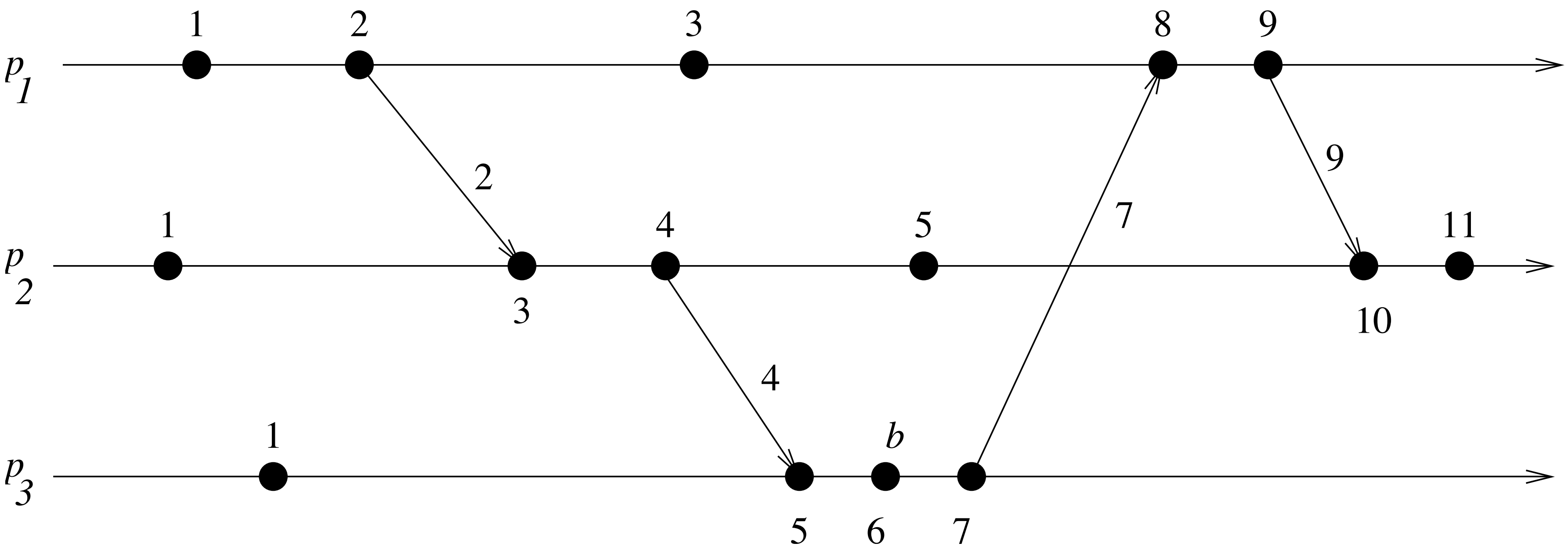
Note

Adjustments take place in the middleware layer



Scalar Time

Evolution of scalar time: The space-time diagram of a distributed execution.



Total Ordering

- Scalar clocks can be used to totally order events in a distributed system.
- The main problem in totally ordering events is that two or more events at different processes may have identical timestamp.
- For example in Figure 3.1, the third event of process P1 and the second event of process P2 have identical scalar timestamp.

Total Ordering

A tie-breaking mechanism is needed to order such events. A tie is broken as follows:

- Process identifiers are linearly ordered and tie among events with identical scalar timestamp is broken on the basis of their process identifiers.
- The lower the process identifier in the ranking, the higher the priority.
- The timestamp of an event is denoted by a tuple (t, i) where t is its time of occurrence and i is the identity of the process where it occurred.
- The total order relation $\prec$ on two events x and y with timestamps *(h,i)* and *(k,j)*, respectively, is defined as follows:

$$x \ \prec \ y \Leftrightarrow \ (h < k \ \textbf{or} \ (h = k \ \textbf{and} \ i < j))$$

Properties. . .

Event counting

- If the increment value *d* is always 1, the scalar time has the following interesting property: if event *e* has a timestamp *h*, then *h-1* represents the minimum logical duration, counted in units of events, required before producing the event *e*;
- We call it the height of the event *e*.
- In other words, *h-1* events have been produced sequentially before the event *e* regardless of the processes that produced these events.
- For example, in Figure 3.1, five events precede event *b* on the longest causal path ending at *b*.

Properties. . .

No Strong Consistency

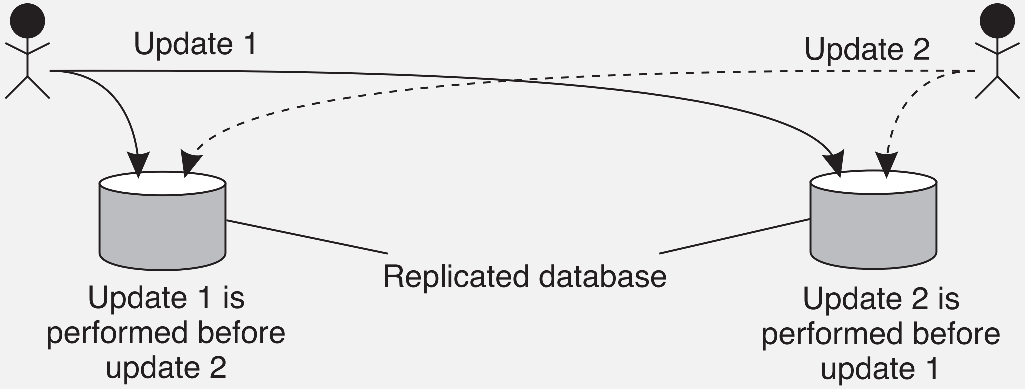
- The system of scalar clocks is not strongly consistent; that is, for two events e_i and e_j, $C(e_i) < C(e_j) \not\Longrightarrow e_i \rightarrow e_j$.
- For example, in Figure 3.1, the third event of process P_1 has smaller scalar timestamp than the third event of process P_2.However, the former did not happen before the latter.
- The reason that scalar clocks are not strongly consistent is that the logical local clock and logical global clock of a process are squashed into one, resulting in the loss causal dependency information among events at different processes.
- For example, in Figure 3.1, when process P_2 receives the first message from process P_1, it updates its clock to 3, forgetting that the timestamp of the latest event at P_1 on which it depends is 2.

Example: Totally ordered multicast

Problem

We sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere:

- P_1 adds \$100 to an account (initial value: \$1000)
- P_2 increments account by 1%
- There are two replicas


Update 1
Update 2
Replicated database
Update 1 is performed before update 2
Update 2 is performed before update 1

Result

In absence of proper synchronization:
replica #1 $\leftarrow$ \$1111, while replica #2 $\leftarrow$ \$1110.

Example: Totally ordered multicast

Solution

- Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue $queue_i$.
- Any incoming message at P_j is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

P_j passes a message msg_i to its application if:

(1) msg_i is at the head of $queue_j$

(2) for each process P_k, there is a message msg_k in $queue_j$ with a larger timestamp.

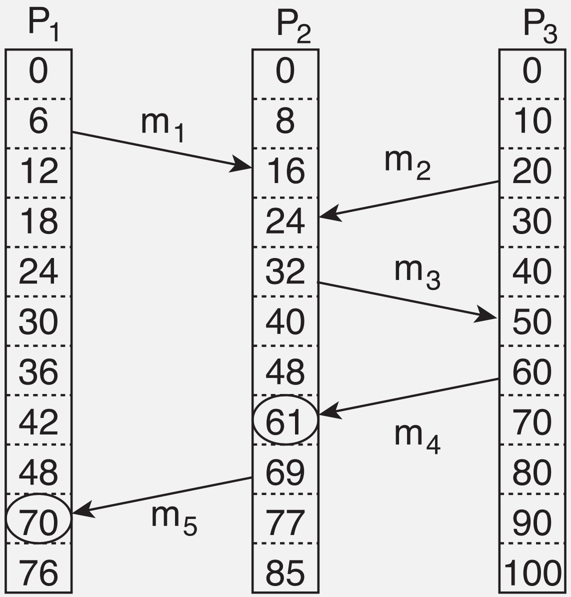
Note

We are assuming that communication is reliable and FIFO ordered.

Vector clocks

Observation

Lamport's clocks do not guarantee that if $C(a) < C(b)$ that a causally preceded b



P_1	P_2	P_3
0	0	0
6	8	10
12	16	20
18	24	30
24	32	40
30	40	50
36	48	60
42	61	70
48	69	80
70	77	90
76	85	100

Observation

Event a: m_1 is received at $T = 16$;
Event b: m_2 is sent at $T = 20$.

Note

We cannot conclude that a causally precedes b.

Vector clocks

Solution

- Each process P_i has an array $VC_i[1..n]$, where $VC_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j.
- When P_i sends a message m, it adds 1 to $VC_i[i]$, and sends VC_i along with m as vector timestamp $vt(m)$. Result: upon arrival, recipient knows P_i's timestamp.
- When a process P_j delivers a message m that it received from P_i with vector timestamp $ts(m)$, it

  (1) updates each $VC_j[k]$ to $\max\{VC_j[k], ts(m)[k]\}$ for all k = 1 to n
  (2) increments $VC_j[j]$ by 1.

Question

What does $VC_i[j] = k$ mean in terms of messages sent and received?

Causally ordered multicasting

Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

Adjustment

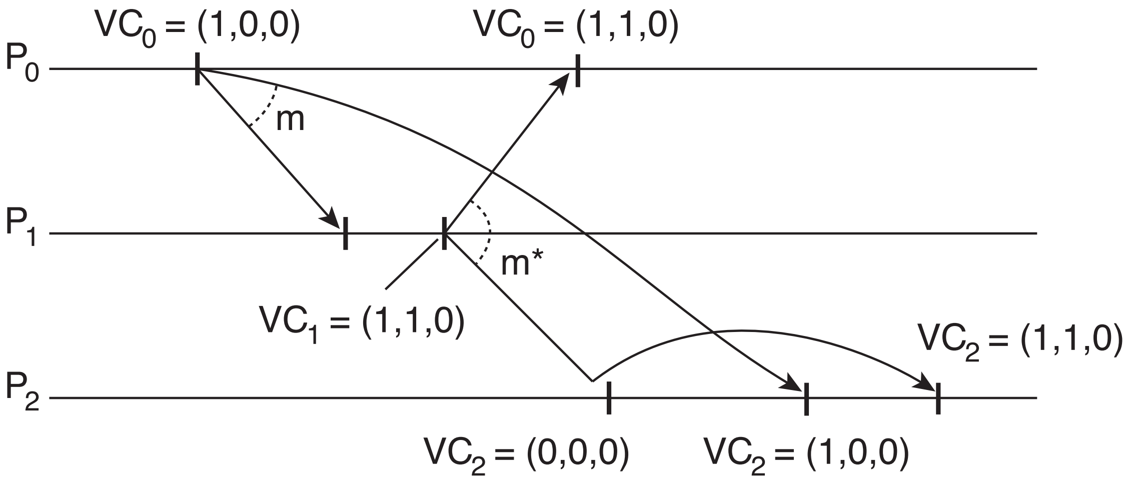
P_i increments $VC_i[i]$ only when sending a message, and P_j “adjusts” VC_j when receiving a message (i.e., effectively does not change $VC_j[j]$).

P_j postpones delivery of m until:

- $ts(m)[i] = VC_j[i] + 1$.
- $ts(m)[k] \leq VC_j[k]$ for $k \neq i$.

Causally ordered multicasting

Example



Example

Take $VC_2 = [0,2,2]$, $ts(m) = [1,3,0]$ from P_0. What information does P_2 have, and what will it do when receiving m (from P_0)?

Thread Synchronisation

Synchronizing Statements vs. Methods

Any synchronized instance method can be converted into a synchronized statement. Suppose that the following is a synchronized instance method:

```
public synchronized void xMethod() {
  // method body
}
```

This method is equivalent to

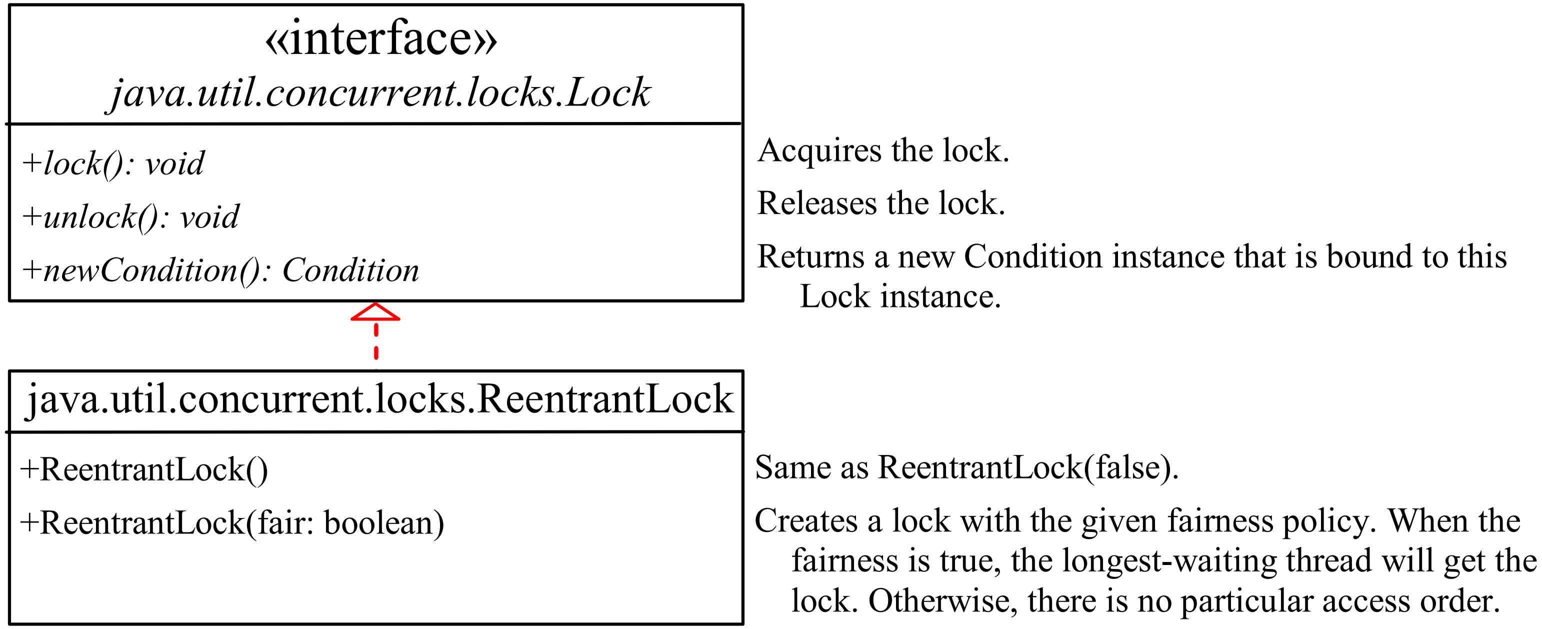
```
public void xMethod() {
  synchronized (this) {
    // method body
  }
}
```

Synchronization Using Locks

Optional

A synchronized instance method implicitly acquires a lock on the instance before it executes the method.

JDK 1.5 enables you to use locks explicitly. The new locking features are flexible and give you more control for coordinating threads. A lock is an instance of the Lock interface, which declares the methods for acquiring and releasing locks, as shown in Figure 29.14. A lock may also use the newCondition() method to create any number of Condition objects, which can be used for thread communications.



«interface»
java.util.concurrent.locks.Lock

+lock(): void — Acquires the lock.

+unlock(): void — Releases the lock.

+newCondition(): Condition — Returns a new Condition instance that is bound to this Lock instance.

java.util.concurrent.locks.ReentrantLock

+ReentrantLock() — Same as ReentrantLock(false).

+ReentrantLock(fair: boolean) — Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

Fairness Policy

Optional

ReentrantLock is a concrete implementation of Lock for creating mutual exclusive locks. You can create a lock with the specified fairness policy. True fairness policies guarantee the longest-wait thread to obtain the lock first. False fairness policies grant a lock to a waiting thread without any access order. Programs using fair locks accessed by many threads may have poor overall performance than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation.

Example: Using Locks

Optional

This example revises AccountWithoutSync.java in Listing 29.7 to synchronize the account modification using explicit locks.

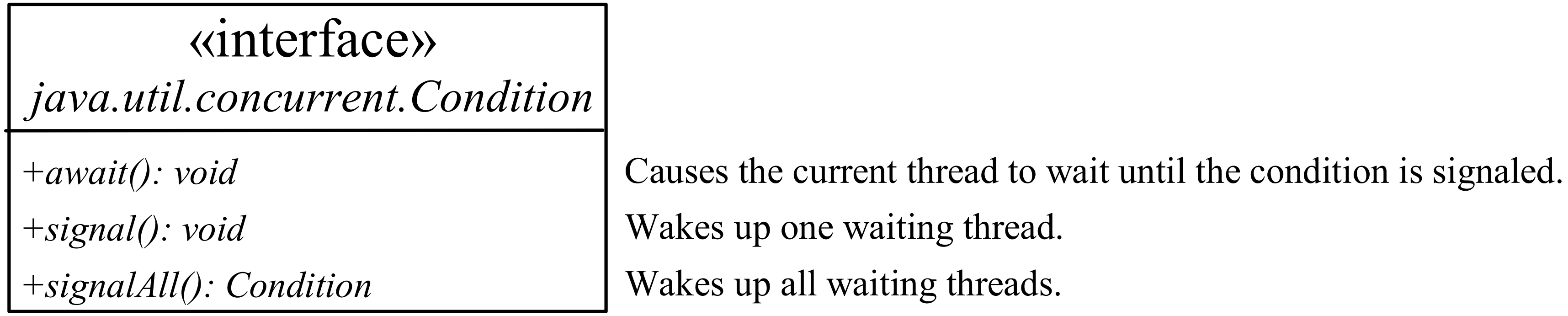
```
private static Lock lock = new ReentrantLock(); //
Create a lock
.
. // Critical Section
.
lock.lock(); // Acquire the lock
lock.unlock(); // Release the lock
```

AccountWithSyncUsingLock

Cooperation Among Threads

Optional

The conditions can be used to facilitate communications among threads. A thread can specify what to do under a certain condition. Conditions are objects created by invoking the newCondition() method on a Lock object. Once a condition is created, you can use its await(), signal(), and signalAll() methods for thread communications, as shown in Figure 29.15. The await() method causes the current thread to wait until the condition is signaled. The signal() method wakes up one waiting thread, and the signalAll() method wakes all waiting threads.



«interface»
java.util.concurrent.Condition

+await(): void Causes the current thread to wait until the condition is signaled.

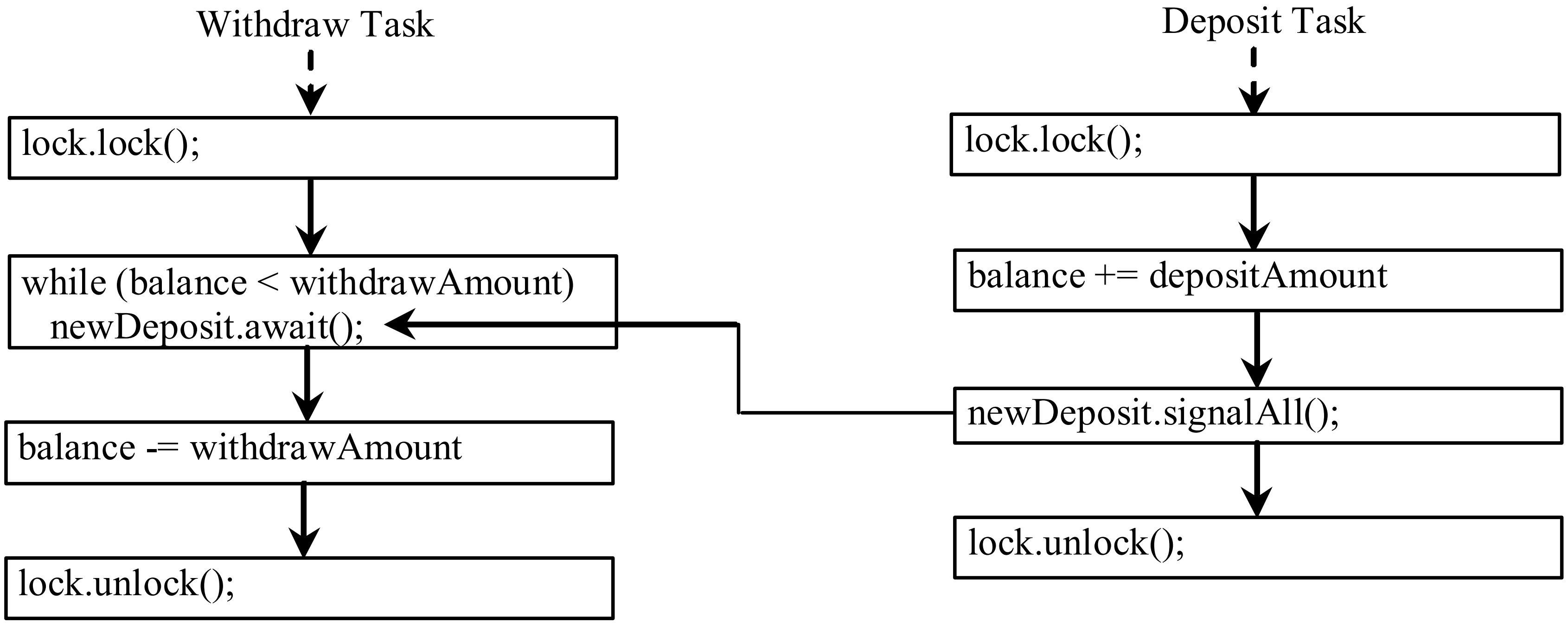
+signal(): void Wakes up one waiting thread.

+signalAll(): Condition Wakes up all waiting threads.

Cooperation Among Threads

Optional

To synchronize the operations, use a lock with a condition: newDeposit (i.e., new deposit added to the account). If the balance is less than the amount to be withdrawn, the withdraw task will wait for the newDeposit condition. When the deposit task adds money to the account, the task signals the waiting withdraw task to try again. The interaction between the two tasks is shown in Figure 29.16.



Withdraw Task

```
lock.lock();
```

```
while (balance < withdrawAmount)
  newDeposit.await();
```

```
balance -= withdrawAmount
```

```
lock.unlock();
```

Deposit Task

```
lock.lock();
```

```
balance += depositAmount
```

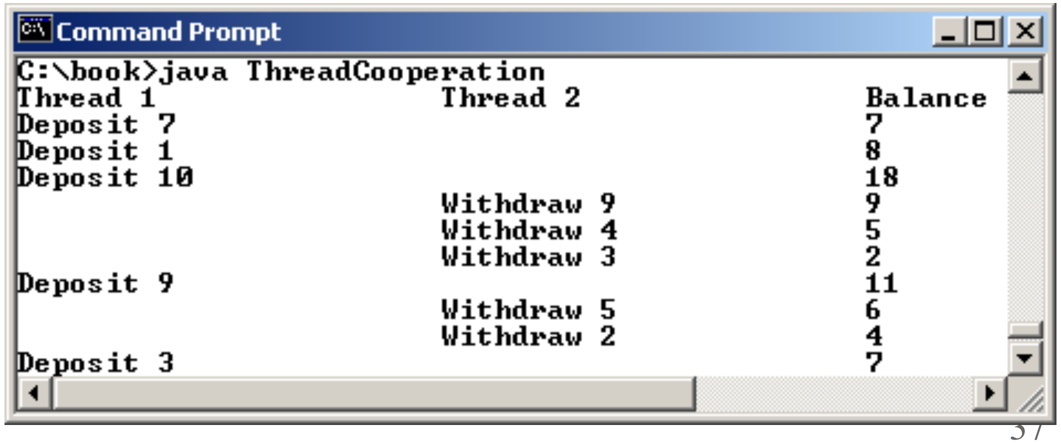
```
newDeposit.signalAll();
```

```
lock.unlock();
```

Example: Thread Cooperation

Optional

Write a program that demonstrates thread cooperation. Suppose that you create and launch two threads, one deposits to an account, and the other withdraws from the same account. The second thread has to wait if the amount to be withdrawn is more than the current balance in the account. Whenever new fund is deposited to the account, the first thread notifies the second thread to resume. If the amount is still not enough for a withdrawal, the second thread has to continue to wait for more fund in the account. Assume the initial balance is 0 and the amount to deposit and to withdraw is randomly generated.



```
C:\book>java ThreadCooperation
Thread 1                 Thread 2                 Balance
Deposit 7                                         7
Deposit 1                                         8
Deposit 10                                        18
                         Withdraw 9               9
                         Withdraw 4               5
                         Withdraw 3               2
Deposit 9                                         11
                         Withdraw 5               6
                         Withdraw 2               4
Deposit 3                                         7
```

ThreadCooperation

Java’s Built-in Monitors

Optional

Locks and conditions are new in Java 5. Prior to Java 5, thread communications are programmed using object’s built-in monitors. Locks and conditions are more powerful and flexible than the built-in monitor. For this reason, this section can be completely ignored. However, if you work with legacy Java code, you may encounter the Java’s built-in monitor. A *monitor* is an object with mutual exclusion and synchronization capabilities. Only one thread can execute a method at a time in the monitor. A thread enters the monitor by acquiring a lock on the monitor and exits by releasing the lock. *Any object can be a monitor*. An object becomes a monitor once a thread locks it. Locking is implemented using the synchronized keyword on a method or a block. A thread must acquire a lock before executing a synchronized method or block. A thread can wait in a monitor if the condition is not right for it to continue executing in the monitor.

wait(), notify(), and notifyAll()

Optional

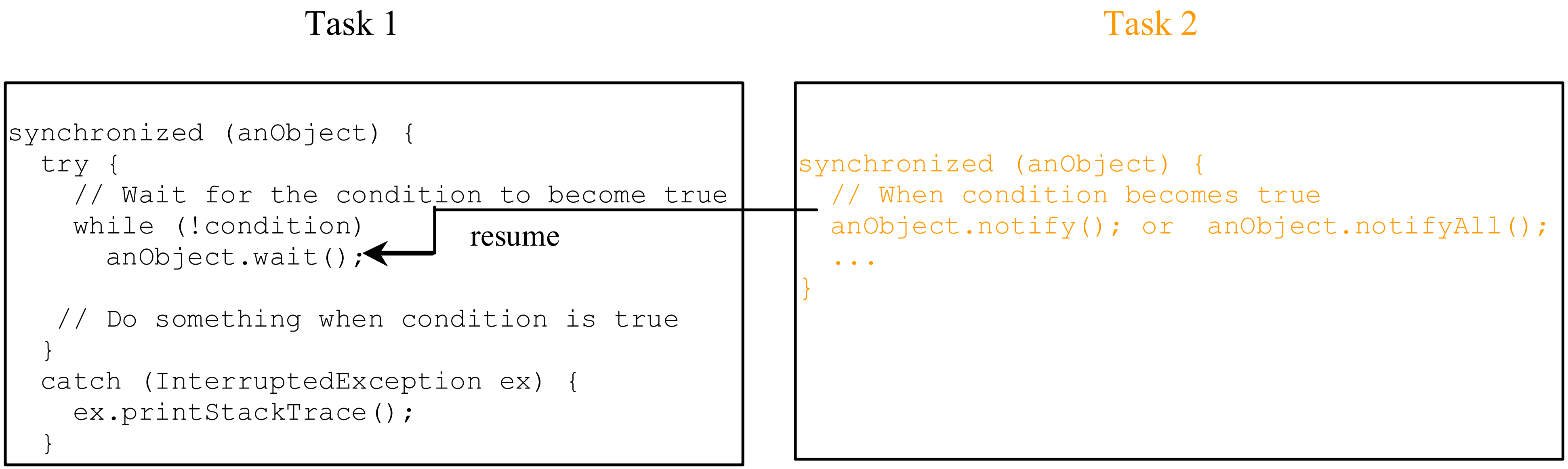
Use the wait(), notify(), and notifyAll() methods to facilitate communication among threads.

The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the calling object of these methods. Otherwise, an IllegalMonitorStateException would occur.

The wait() method lets the thread wait until some condition occurs. When it occurs, you can use the notify() or notifyAll() methods to notify the waiting threads to resume normal execution. The notifyAll() method wakes up all waiting threads, while notify() picks up only one thread from a waiting queue.

Example: Using Monitor

Optional



Task 1

```
synchronized (anObject) {
  try {
    // Wait for the condition to become true
    while (!condition)
      anObject.wait();

   // Do something when condition is true
  }
  catch (InterruptedException ex) {
    ex.printStackTrace();
  }
```

Task 2

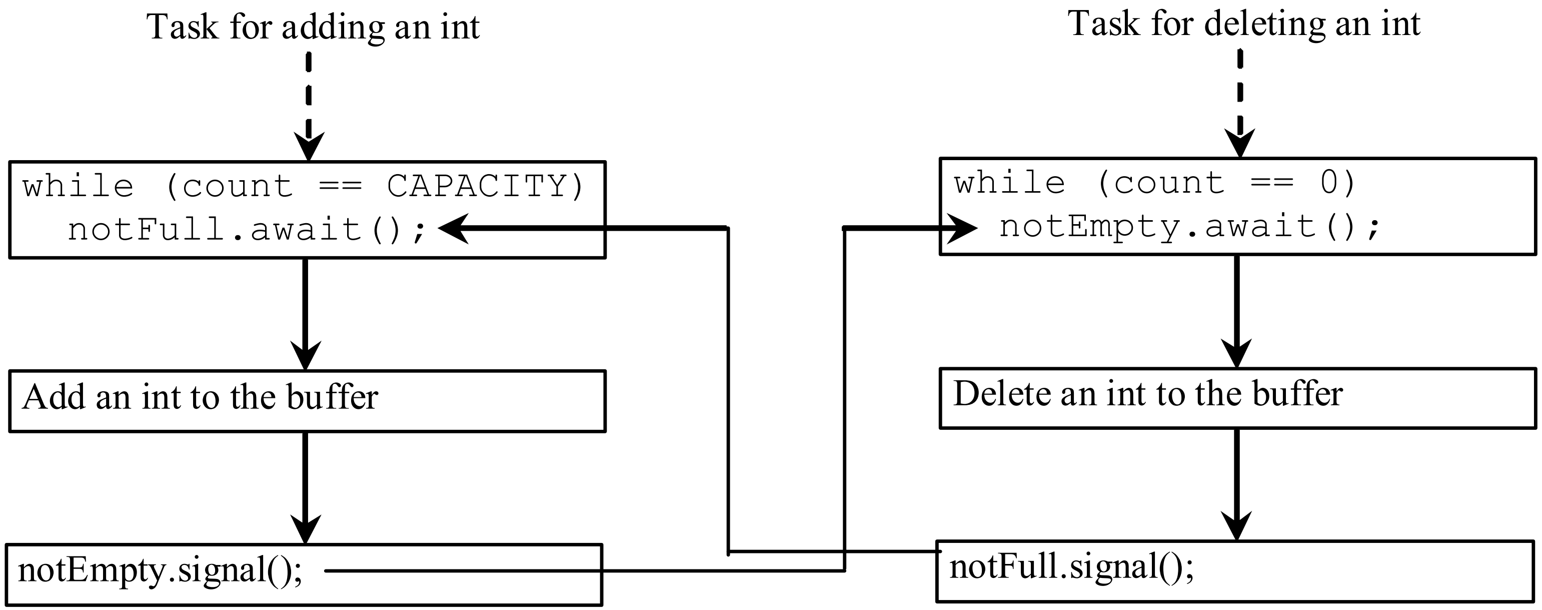
```
synchronized (anObject) {
  // When condition becomes true
  anObject.notify(); or  anObject.notifyAll();
  ...
}
```

- The wait(), notify(), and notifyAll() methods must be called in a synchronized method or a synchronized block on the receiving object of these methods. Otherwise, an IllegalMonitorStateException will occur.
- When wait() is invoked, it pauses the thread and simultaneously releases the lock on the object. When the thread is restarted after being notified, the lock is automatically reacquired.
- The wait(), notify(), and notifyAll() methods on an object are analogous to the await(), signal(), and signalAll() methods on a condition.

Case Study: Producer/Consumer

Optional

Consider the classic Consumer/Producer example. Suppose you use a buffer to store integers. The buffer size is limited. The buffer provides the method write(int) to add an int value to the buffer and the method read() to read and delete an int value from the buffer. To synchronize the operations, use a lock with two conditions: notEmpty (i.e., buffer is not empty) and notFull (i.e., buffer is not full). When a task adds an int to the buffer, if the buffer is full, the task will wait for the notFull condition. When a task deletes an int from the buffer, if the buffer is empty, the task will wait for the notEmpty condition. The interaction between the two tasks is shown in Figure 29.19.



Case Study: Producer/Consumer

Optional

Listing 29.10 presents the complete program. The program contains the Buffer class (lines 43-89) and two tasks for repeatedly producing and consuming numbers to and from the buffer (lines 15-41). The write(int) method (line 58) adds an integer to the buffer. The read() method (line 75) deletes and returns an integer from the buffer.

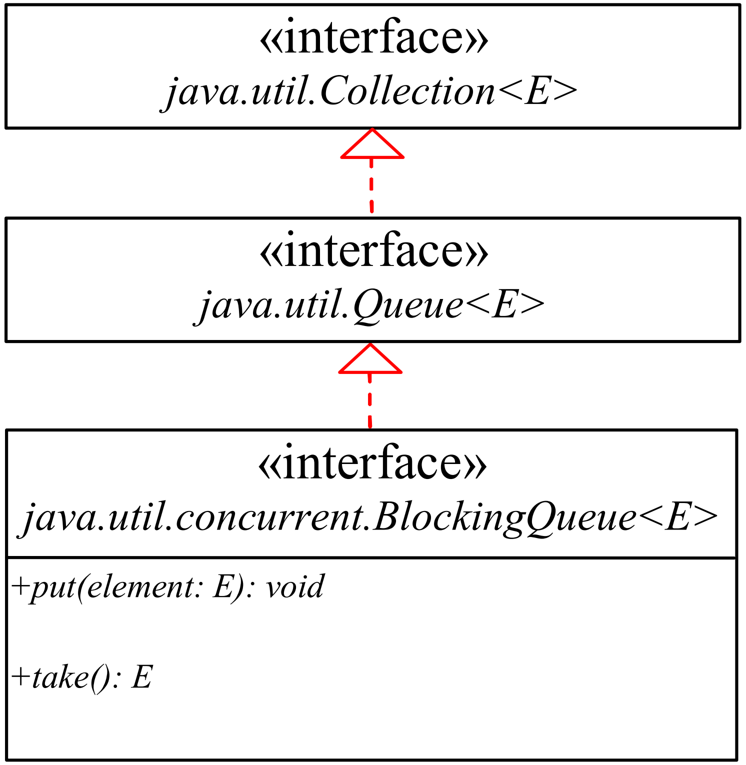
For simplicity, the buffer is implemented using a linked list (lines 48-49). Two conditions notEmpty and notFull on the lock are created in lines 55-56. The conditions are bound to a lock. A lock must be acquired before a condition can be applied. If you use the wait() and notify() methods to rewrite this example, you have to designate two objects as monitors.

ConsumerProducer

Blocking Queues

Optional

§22.8 introduced queues and priority queues. A *blocking queue* causes a thread to block when you try to add an element to a full queue or to remove an element from an empty queue.



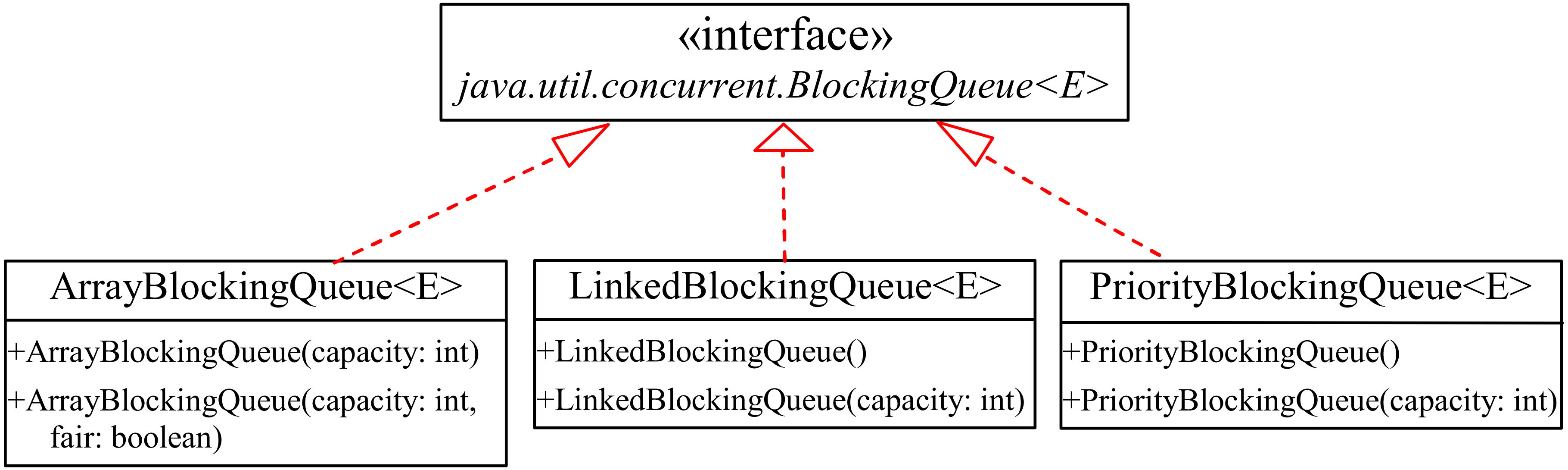
Inserts an element to the tail of the queue. Waits if the queue is full.

Retrieves and removes the head of this queue. Waits if the queue is empty.

Concrete Blocking Queues

Optional

Three concrete blocking queues ArrayBlockingQueue, LinkedBlockingQueue, and PriorityBlockingQueue are supported in JDK 1.5, as shown in Figure 29.22. All are in the java.util.concurrent package. ArrayBlockingQueue implements a blocking queue using an array. You have to specify a capacity or an optional fairness to construct an ArrayBlockingQueue. LinkedBlockingQueue implements a blocking queue using a linked list. You may create an unbounded or bounded LinkedBlockingQueue. PriorityBlockingQueue is a priority queue. You may create an unbounded or bounded priority queue.



«interface»
java.util.concurrent.BlockingQueue<E>

ArrayBlockingQueue<E>
+ArrayBlockingQueue(capacity: int)
+ArrayBlockingQueue(capacity: int, fair: boolean)

LinkedBlockingQueue<E>
+LinkedBlockingQueue()
+LinkedBlockingQueue(capacity: int)

PriorityBlockingQueue<E>
+PriorityBlockingQueue()
+PriorityBlockingQueue(capacity: int)

Producer/Consumer Using Blocking Queues

Listing 29.11 gives an example of using an <u>ArrayBlockingQueue</u> to simplify the Consumer/Producer example in Listing 29.11.

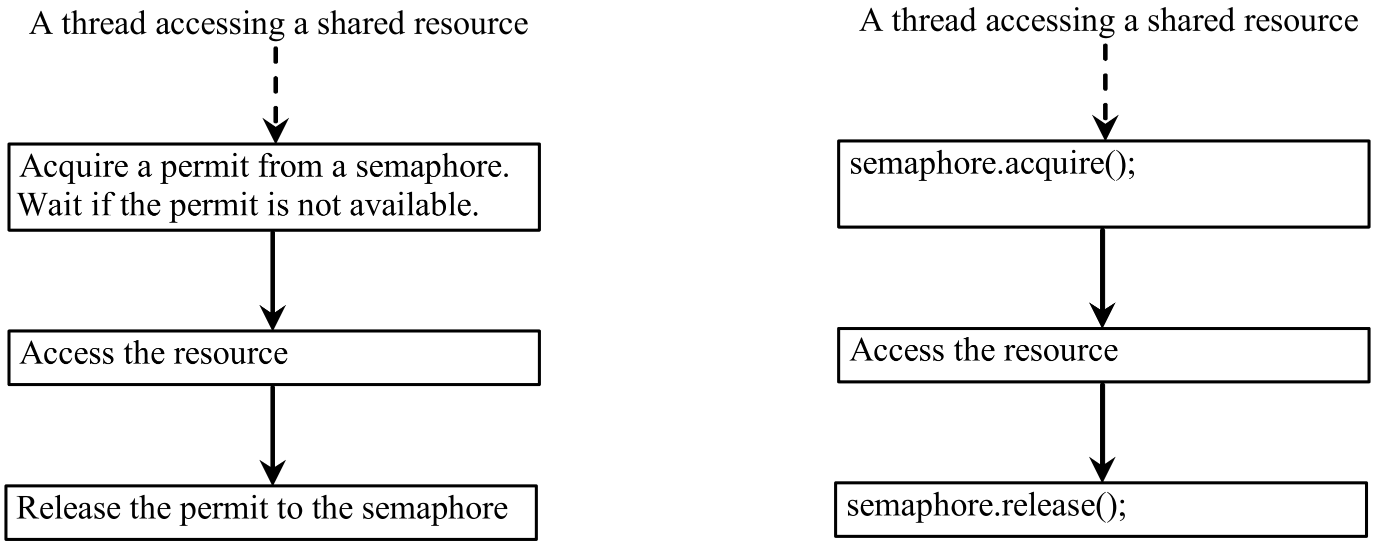
```
import java.util.concurrent.*;
public class ConsumerProducerUsingBlockingQueue {
  private static ArrayBlockingQueue<Integer> buffer =
    new ArrayBlockingQueue<Integer>(2);
  public static void main(String[] args) {
    // Create a thread pool with two threads
    ExecutorService executor = Executors.newFixedThreadPool(2);
    executor.execute(new ProducerTask());
    executor.execute(new ConsumerTask());
    executor.shutdown();
  }
  // A task for adding an int to the buffer
  private static class ProducerTask implements Runnable {
    public void run() {
      try {
        int i = 1;
        while (true) {
          System.out.println("Producer writes " + i);
          buffer.put(i++); // Add any value to the buffer, say, 1
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      } catch (InterruptedException ex) {/* … */}
    }
  }
  // A task for reading and deleting an int from the buffer
  private static class ConsumerTask implements Runnable {
    public void run() {
      try {
        while (true) {
          System.out.println("\t\t\tConsumer reads " + buffer.take());
          // Put the thread into sleep
          Thread.sleep((int)(Math.random() * 10000));
        }
      } catch (InterruptedException ex) {/* … */}
    }
  }
}
```

ConsumerProducerUsingBlockingQueue

Semaphores

Optional

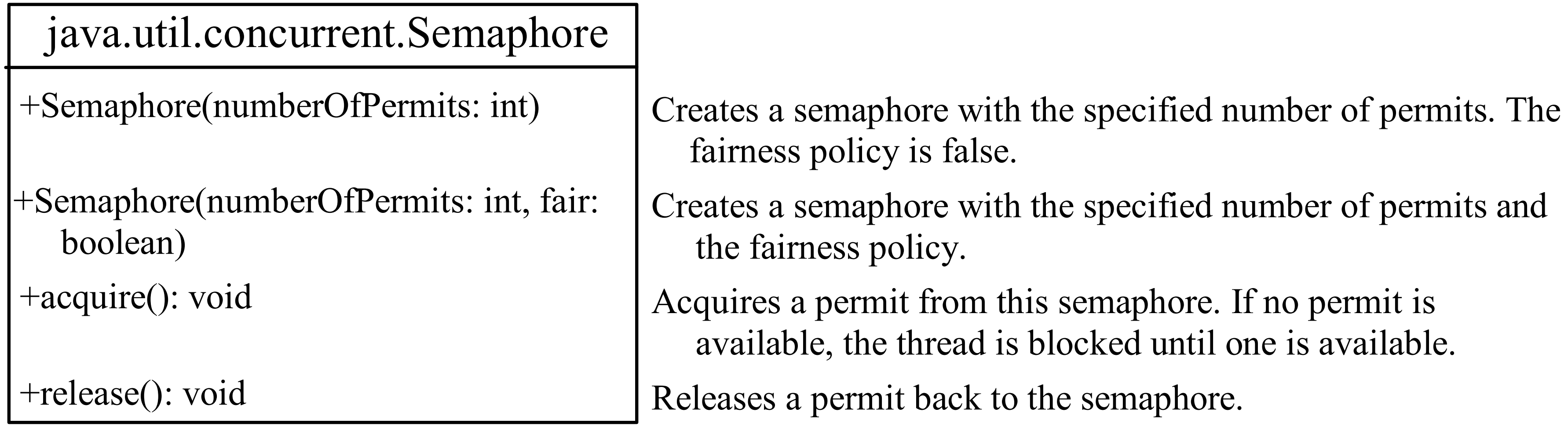
Semaphores can be used to restrict the number of threads that access a shared resource. Before accessing the resource, a thread must acquire a permit from the semaphore. After finishing with the resource, the thread must return the permit back to the semaphore, as shown in Figure 29.29.


A thread accessing a shared resource
Acquire a permit from a semaphore. Wait if the permit is not available.
Access the resource
Release the permit to the semaphore
A thread accessing a shared resource
semaphore.acquire();
Access the resource
semaphore.release();

Creating Semaphores

Optional

To create a semaphore, you have to specify the number of permits with an optional fairness policy, as shown in Figure 29.29. A task acquires a permit by invoking the semaphore's acquire() method and releases the permit by invoking the semaphore's release() method. Once a permit is acquired, the total number of available permits in a semaphore is reduced by 1. Once a permit is released, the total number of available permits in a semaphore is increased by 1.



java.util.concurrent.Semaphore

+Semaphore(numberOfPermits: int)
Creates a semaphore with the specified number of permits. The fairness policy is false.

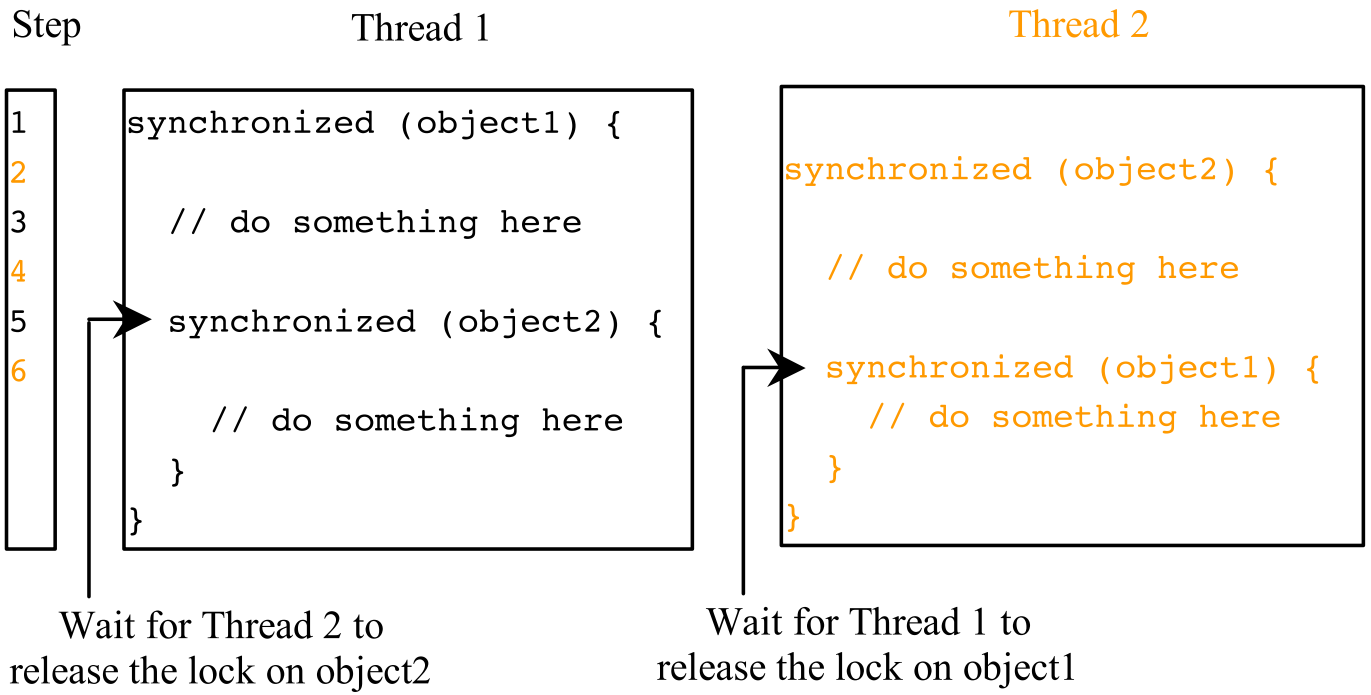
+Semaphore(numberOfPermits: int, fair: boolean)
Creates a semaphore with the specified number of permits and the fairness policy.

+acquire(): void
Acquires a permit from this semaphore. If no permit is available, the thread is blocked until one is available.

+release(): void
Releases a permit back to the semaphore.

Deadlock

Sometimes two or more threads need to acquire the locks on several shared objects. This could cause *deadlock*, in which each thread has the lock on one of the objects and is waiting for the lock on the other object. Consider the scenario with two threads and two objects, as shown in Figure 29.15 (32.25). Thread 1 acquired a lock on object1 and Thread 2 acquired a lock on object2. Now Thread 1 is waiting for the lock on object2 and Thread 2 for the lock on object1. The two threads wait for each other to release the in order to get the lock, and neither can continue to run.



```
synchronized (object1) {

  // do something here

  synchronized (object2) {

    // do something here
  }
}
```

```

synchronized (object2) {

  // do something here

  synchronized (object1) {
    // do something here
  }
}
```

Wait for Thread 2 to
release the lock on object2

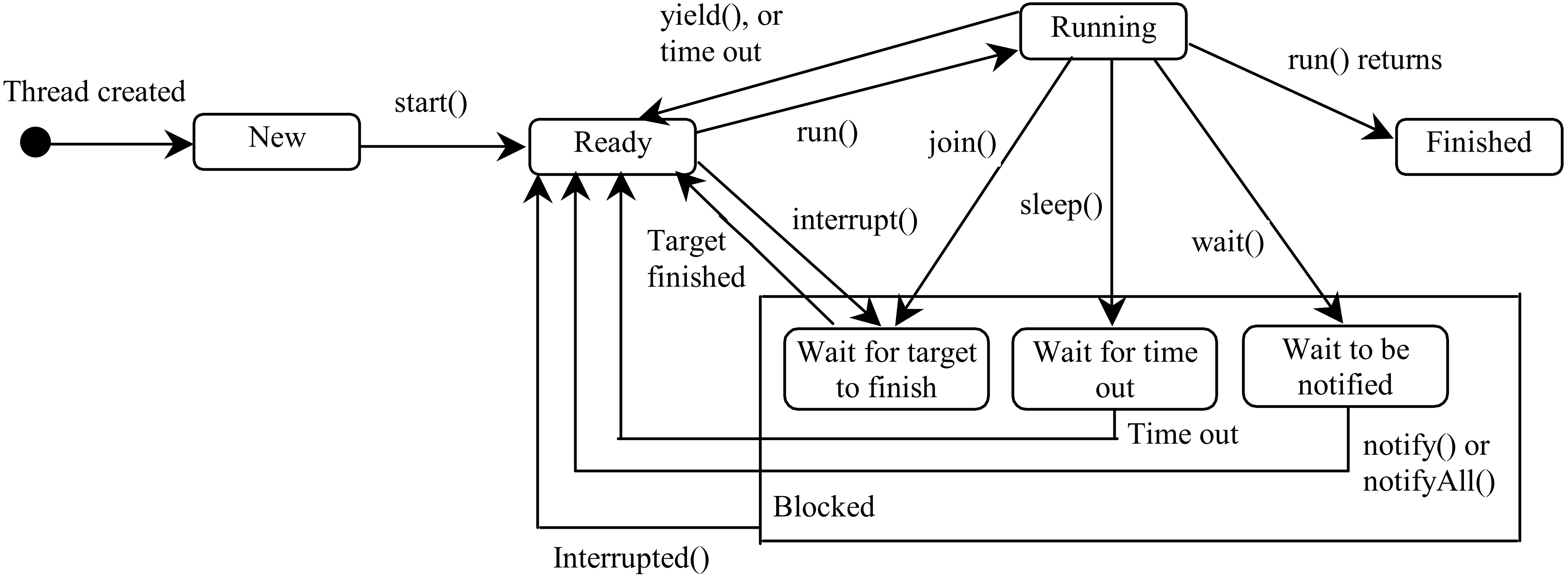
Wait for Thread 1 to
release the lock on object1

Preventing Deadlock

Deadlock can be easily avoided by using a simple technique known as resource ordering. With this technique, you assign an order on all the objects whose locks must be acquired and ensure that each thread acquires the locks in that order. For the example in Figure 29.15 (32.25) , suppose the objects are ordered as object1 and object2. Using the resource ordering technique, Thread 2 must acquire a lock on object1 first, then on object2. Once Thread 1 acquired a lock on object1, Thread 2 has to wait for a lock on object1. So Thread 1 will be able to acquire a lock on object2 and no deadlock would occur.

Thread States

A thread can be in one of five states: New, Ready, Running, Blocked, or Finished.

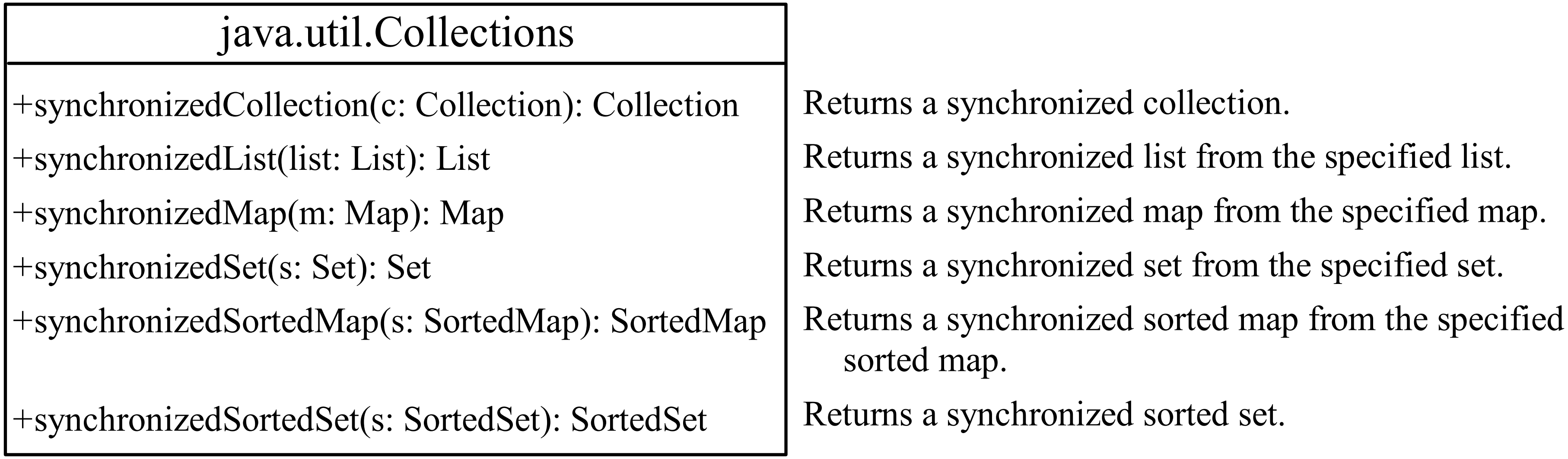


Synchronized Collections

Optional

The classes in the Java Collections Framework are not thread-safe, i.e., the contents may be corrupted if they are accessed and updated concurrently by multiple threads. You can protect the data in a collection by locking the collection or using synchronized collections.

The <u>Collections</u> class provides six static methods for wrapping a collection into a synchronized version, as shown in Figure 29.27. The collections created using these methods are called *synchronization wrappers*.



java.util.Collections	
+synchronizedCollection(c: Collection): Collection	Returns a synchronized collection.
+synchronizedList(list: List): List	Returns a synchronized list from the specified list.
+synchronizedMap(m: Map): Map	Returns a synchronized map from the specified map.
+synchronizedSet(s: Set): Set	Returns a synchronized set from the specified set.
+synchronizedSortedMap(s: SortedMap): SortedMap	Returns a synchronized sorted map from the specified sorted map.
+synchronizedSortedSet(s: SortedSet): SortedSet	Returns a synchronized sorted set.

Vector, Stack, and Hashtable

Optional

Invoking synchronizedCollection(Collection c) returns a new Collection object, in which all the methods that access and update the original collection c are synchronized. These methods are implemented using the synchronized keyword. For example, the add method is implemented like this:

```
public boolean add(E o) {
  synchronized (this) { return c.add(o); }
}
```

The synchronized collections can be safely accessed and modified by multiple threads concurrently.

The methods in java.util.Vector, java.util.Stack, and Hashtable are already synchronized. These are old classes introduced in JDK 1.0. In JDK 1.5, you should use java.util.ArrayList to replace Vector, java.util.LinkedList to replace Stack, and java.util.Map to replace Hashtable. If synchronization is needed, use a synchronization wrapper.

Fail-Fast

Optional

The synchronization wrapper classes are thread-safe, but the iterator is *fail-fast*. This means that if you are using an iterator to traverse a collection while the underlying collection is being modified by another thread, then the iterator will immediately fail by throwing java.util.ConcurrentModificationException, which is a subclass of RuntimeException. To avoid this error, you need to create a synchronized collection object and acquire a lock on the object when traversing it. For example, suppose you want to traverse a set, you have to write the code like this:

```
Set hashSet = Collections.synchronizedSet(new HashSet());
synchronized (hashSet) { // Must synchronize it
  Iterator iterator = hashSet.iterator();
  while (iterator.hasNext()) {
    System.out.println(iterator.next());
  }
}
```

-

Failure to do so may result in nondeterministic behavior, such as ConcurrentModificationException.

Mutual exclusion

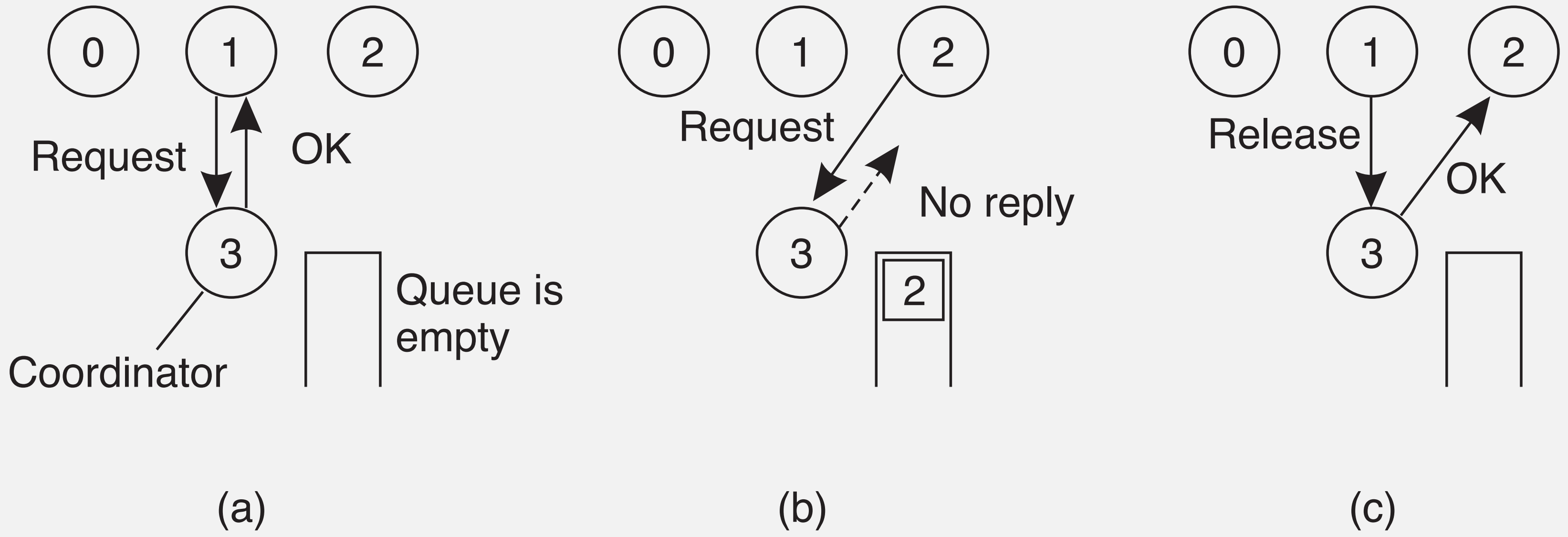
Problem

A number of processes in a distributed system want exclusive access to some resource.

Basic solutions

- Via a centralized server.
- Completely decentralized, using a peer-to-peer system.
- Completely distributed, with no topology imposed.
- Completely distributed along a (logical) ring.

Mutual exclusion: centralized



Decentralized mutual exclusion

Principle

Assume every resource is replicated n times, with each replica having its own coordinator $\Rightarrow$ access requires a majority vote from $m > n/2$ coordinators. A coordinator always responds immediately to a request.

Assumption

When a coordinator crashes, it will recover quickly, but will have forgotten about permissions it had granted.

Decentralized mutual exclusion

Issue

How robust is this system? Let $p = \Delta t / T$ denote the probability that a coordinator crashes and recovers in a period Δt while having an average lifetime T ⇒ probability that k out m coordinators reset:

$$P[\text{violation}] = p_v = \sum_{k=2m-n}^{n} \binom{m}{k} p^k (1-p)^{m-k}$$

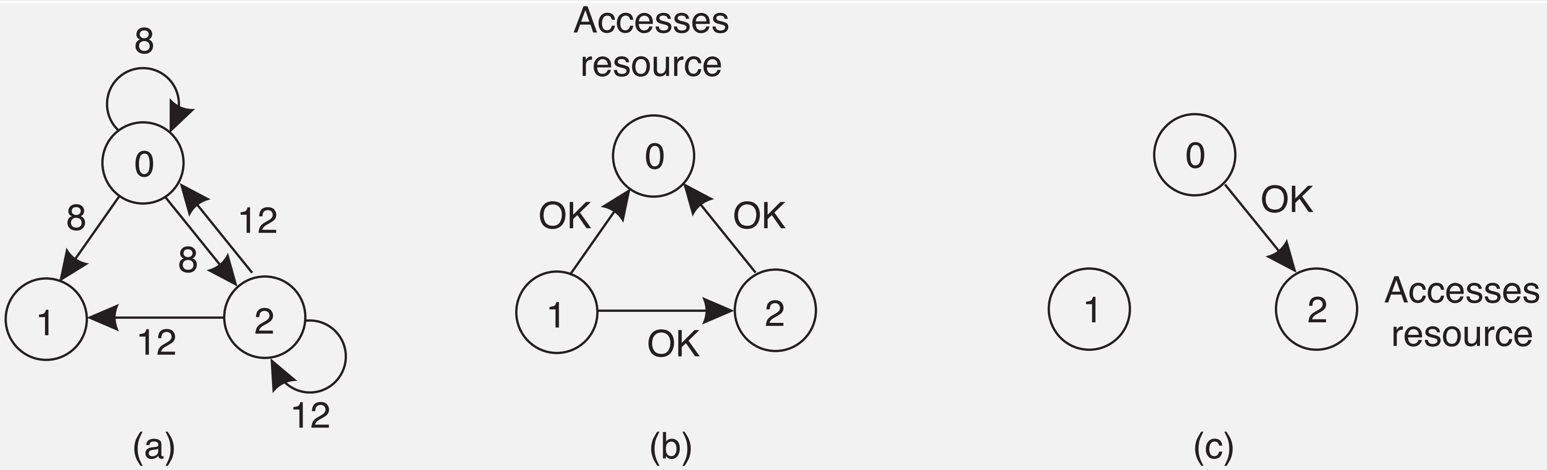
With $p = 0.001$, $n = 32$, $m = 0.75n$, $p_v < 10^{-40}$

Mutual exclusion Ricart & Agrawala

Principle

The same as Lamport except that acknowledgments aren't sent. Instead, replies (i.e. grants) are sent only when

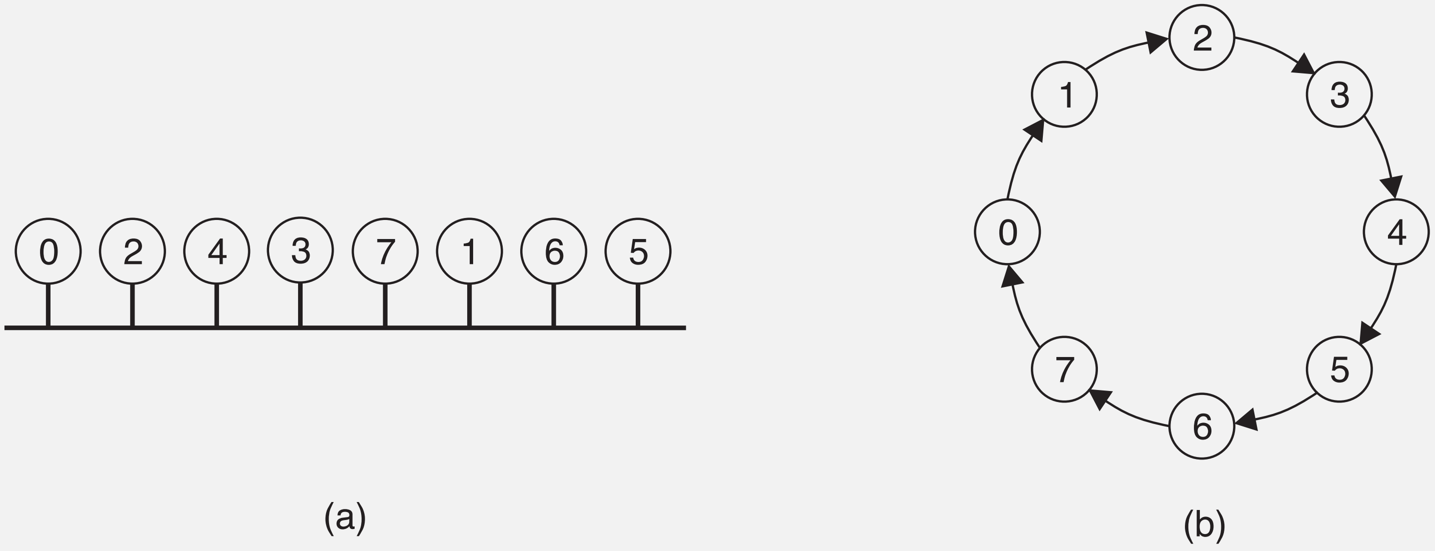
- The receiving process has no interest in the shared resource; or
- The receiving process is waiting for the resource, but has lower priority (known through comparison of timestamps).
- In all other cases, reply is deferred, implying some more local administration.



Mutual exclusion: Token ring algorithm

Essence

Organize processes in a *logical* ring, and let a token be passed between them. The one that holds the token is allowed to enter the critical region (if it wants to).



(a)

(b)

Mutual exclusion: comparison

Algorithm	# msgs per entry/exit	Delay before entry (in msg times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	2mk + m, k = 1,2,...	2mk	Starvation, low eff.
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, proc. crash

- Finally, all algorithms except the decentralized one suffer badly in the event of crashes.
- Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system.
- It is ironic that the distributed algorithms are even more sensitive to crashes than the centralized one.
- In a system that is designed to be fault tolerant, none of these would be suitable, but if crashes are very infrequent, they might do.
- The decentralized algorithm is less sensitive to crashes, but processes may suffer from starvation and special measures are needed to guarantee efficiency.

Global positioning of nodes

Problem

How can a single node efficiently estimate the latency between any two other nodes in a distributed system?

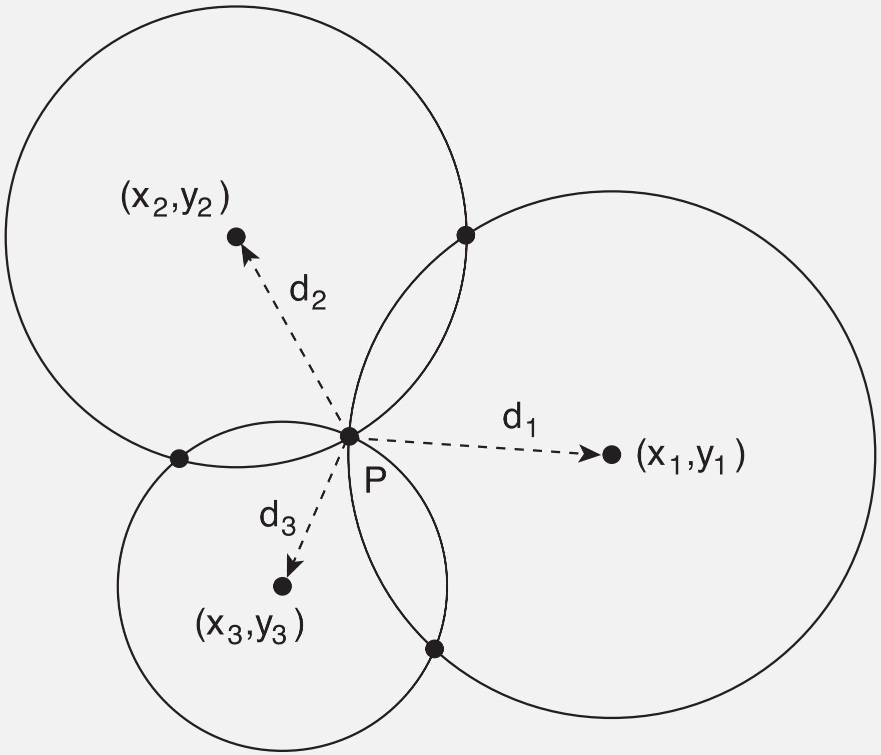
Solution

Construct a geometric overlay network, in which the distance $d(P, Q)$ reflects the actual latency between P and Q.

Computing position

Observation

A node *P* needs $k+1$ landmarks to compute its own position in a *d*-dimensional space. Consider two-dimensional case.



Solution

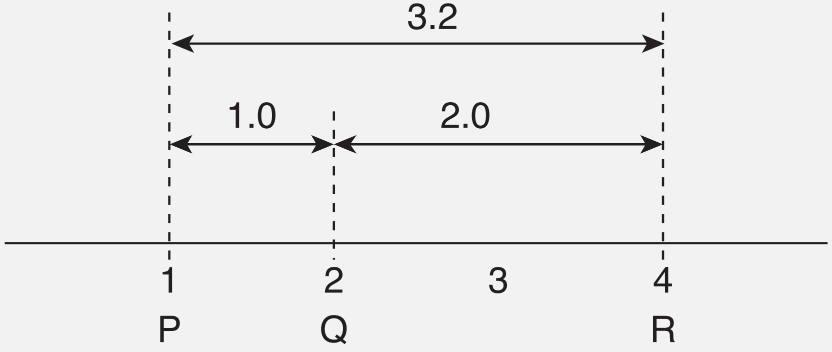
P needs to solve three equations in two unknowns (x_P, y_P):

$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2}$$

Computing position

Problems

- measured latencies to landmarks fluctuate
- computed distances will not even be consistent:



Solution

Let the L landmarks measure their pairwise latencies $d(b_i, b_j)$ and let each node P minimize

$$\sum_{i=1}^{L}\left[\frac{d(b_i,P)-\hat{d}(b_i,P)}{d(b_i,P)}\right]^2$$

where $\hat{d}(b_i,P)$ denotes the distance to landmark b_i given a computed coordinate for P.

Election algorithms

Principle

An algorithm requires that some process acts as a coordinator. The question is how to select this special process dynamically.

Note

In many systems the coordinator is chosen by hand (e.g. file servers). This leads to centralized solutions ⇒ single point of failure.

Question

If a coordinator is chosen dynamically, to what extent can we speak about a centralized or distributed solution?

Question

Is a fully distributed solution, i.e. one without a coordinator, always more robust than any centralized/coordinated solution?

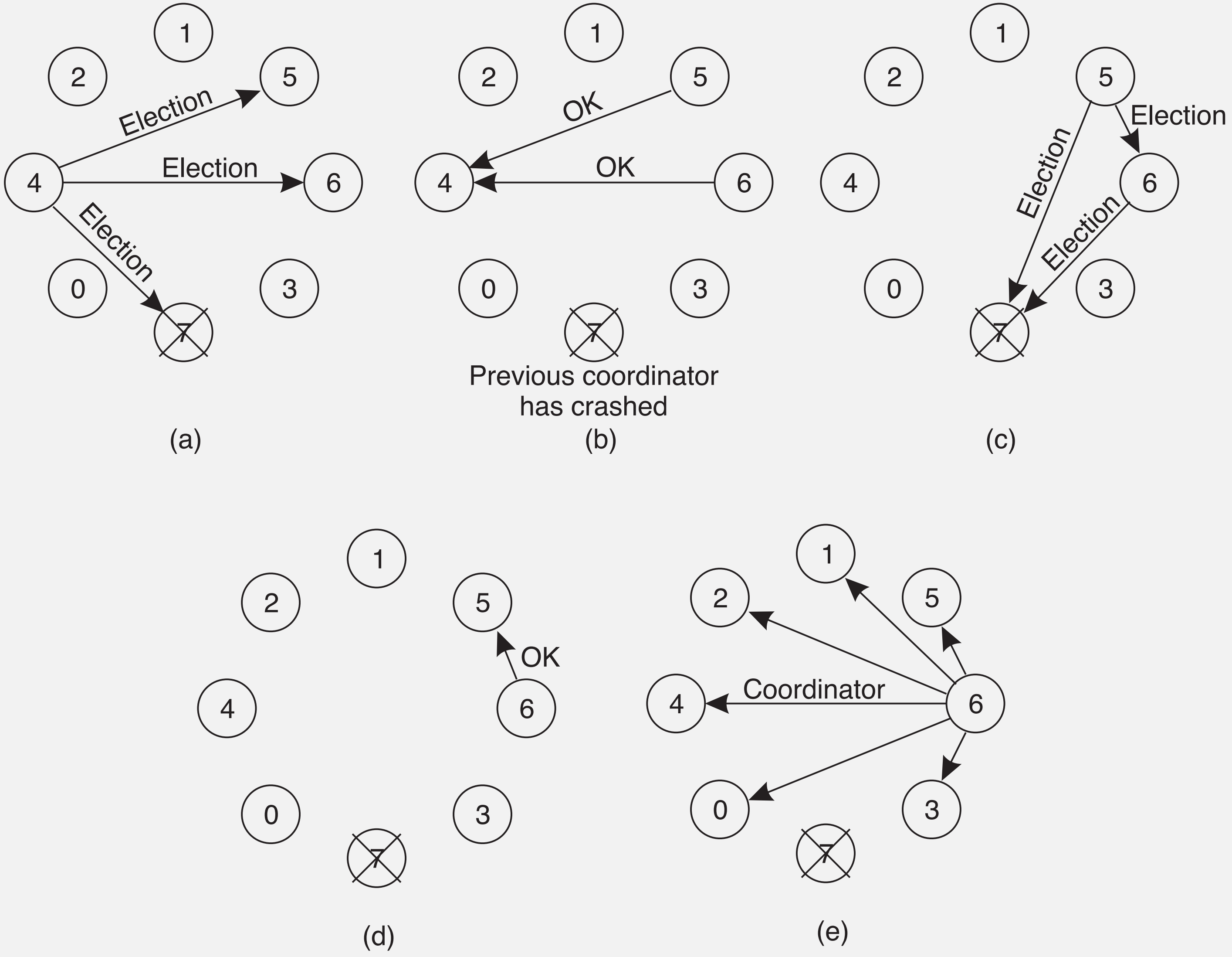
Election by bullying

Principle

Each process has an associated priority (weight). The process with the highest priority should always be elected as the coordinator. Issue How do we find the heaviest process?

- Any process can just start an election by sending an election message to all other processes (assuming you don't know the weights of the others).
- If a process P_{heavy} receives an election message from a lighter process P_{light}, it sends a take-over message to P_{light}. P_{light} is out of the race.
- If a process doesn't get a take-over message back, it wins, and sends a victory message to all other processes.

Election by bullying



(a) (b) (c) (d) (e)

Election in a ring

Principle

Process priority is obtained by organizing processes into a (logical) ring. Process with the highest priority should be elected as coordinator.

- Any process can start an election by sending an election message to its successor. If a successor is down, the message is passed on to the next successor.
- If a message is passed on, the sender adds itself to the list. When it gets back to the initiator, everyone had a chance to make its presence known.
- The initiator sends a coordinator message around the ring containing a list of all living processes. The one with the highest priority is elected as coordinator.

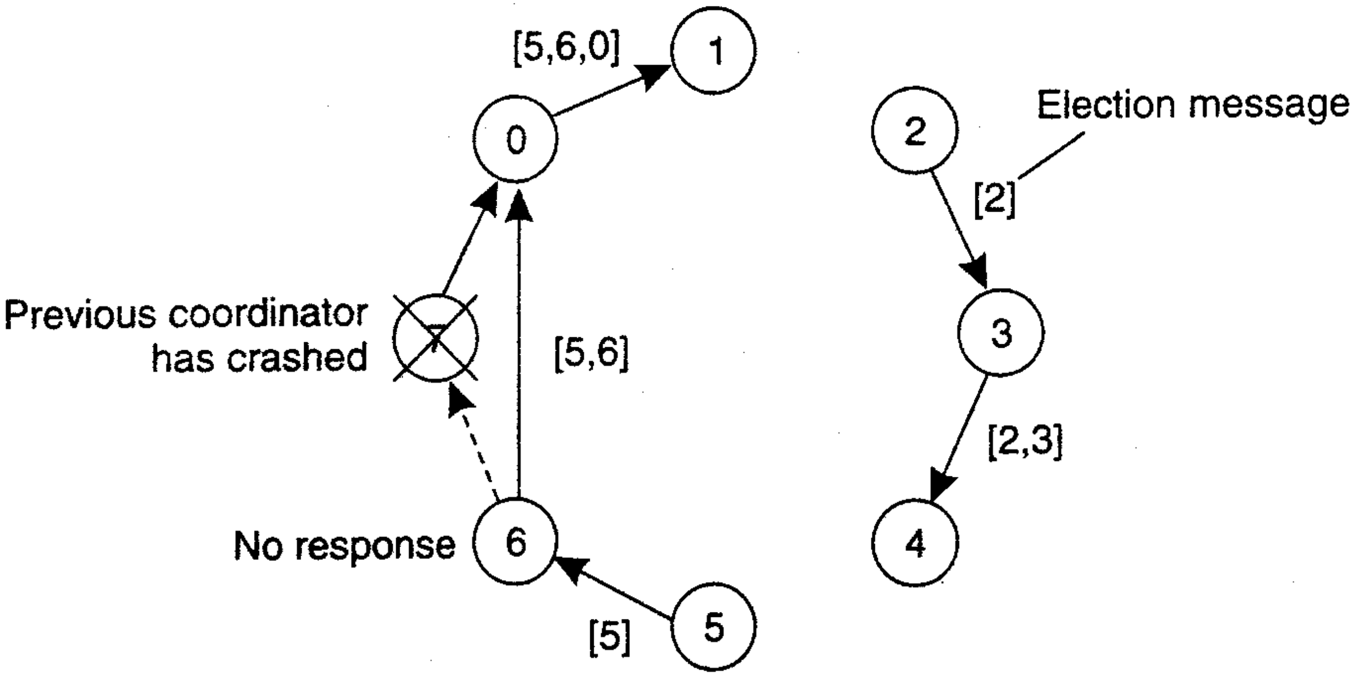
Election in a ring

Question

Does it matter if two processes initiate an election?

Question

What happens if a process crashes *during* the election?


[5,6,0]
1
0
2
Election message
[2]
Previous coordinator
has crashed
7
[5,6]
3
[2,3]
No response
6
4
[5]
5

Superpeer election

Issue

How can we select superpeers such that:

- Normal nodes have low-latency access to superpeers
- Superpeers are evenly distributed across the overlay network
- There is be a predefined fraction of superpeers
- Each superpeer should not need to serve more than a fixed number of normal nodes

Superpeer election

DHTs

Reserve a fixed part of the ID space for superpeers. Example: if S superpeers are needed for a system that uses m-bit identifiers, simply reserve the $k = \lceil \log_2 S \rceil$ leftmost bits for superpeers. With N nodes, we'll have, on average, $2^{k-m}N$ superpeers.

Routing to superpeer

Send message for key p to node responsible for

$$p \text{ AND } \underbrace{11\cdots11}_{k}\underbrace{00\cdots00}_{m-k}$$