Chapter 6: Process Synchronization

Module 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions

Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the **consistency of shared data**
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of **an atomic transaction** and describe mechanisms to ensure atomicity

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure **the orderly execution of cooperating processes**
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having **an integer count** that keeps track of the number of full buffers. Initially, count is set to 0. It **is incremented** by the producer after it produces a new buffer and **is decremented** by the consumer after it consumes a buffer.

Producer

```
while (true) {
    /*  produce an item and put in nextProduced  */
        while (count == BUFFER_SIZE)
                ; // do nothing
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)
              ; // do nothing
       /*  consume the item in nextConsumed */
        nextConsumed =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
}
```

Race Condition

- count++ could be implemented as (producer)

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as (consumer)

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. Bounded Waiting - **A bound** must exist on the number of times that **other processes** are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the N processes

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Algorithm for Process P_i

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j);
                    critical section
        flag[i] = FALSE;
                    remainder section
} while (TRUE);
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Data Structure for Hardware Solutions

```
public class HardwareData
{
    private boolean data;
    public HardwareData(boolean data) {
                    this.data = data;
    }
    public boolean get() {
                    return data;
    }
    public void set(boolean data) {
                    this.data = data;
    }
    // Continued on Next Slide
```

Data Structure for Hardware Solutions

```
public boolean getAndSet(boolean data) {
                        boolean oldValue = this.get();
                        this.set(data);
                        return oldValue;
}
public void swap(HardwareData other) {
                        boolean temp = this.get();
                        this.set(other.get());
                        other.set(temp);
}
}
```

Atomic instruction (read-modify-write)

Atomic instruction

Thread Using get-and-set Lock

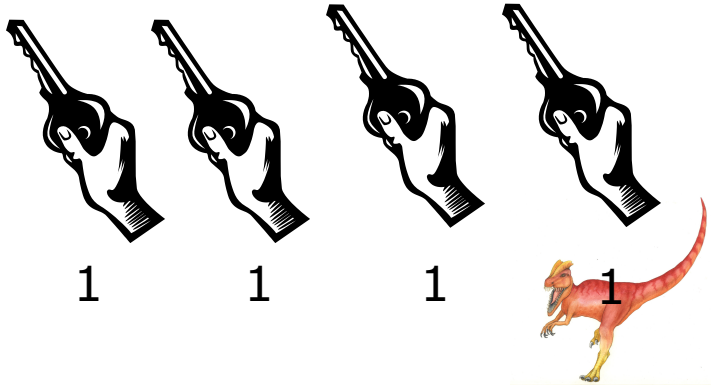
```
while (true) {
        acquire lock
            critical section
        release lock
            remainder section
    }
```

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);

while (true) {
    //acquire lock, get and set lock true
    while (lock.getAndSet(true))
                Thread.yield();

    //criticalSection
        lock.set(false);        //release lock
    //remainder section
}
```

Thread Using swap Instruction

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);
// each thread has a local copy of key
HardwareData key = new HardwareData(true);

    while (true) {
                key.set(true);
                do {
                    lock.swap(key);    //acquire lock
                }
                while (key.get() == true);

                // criticalSection
                  lock.set(false);   //release lock
                //remainder section
    }
```

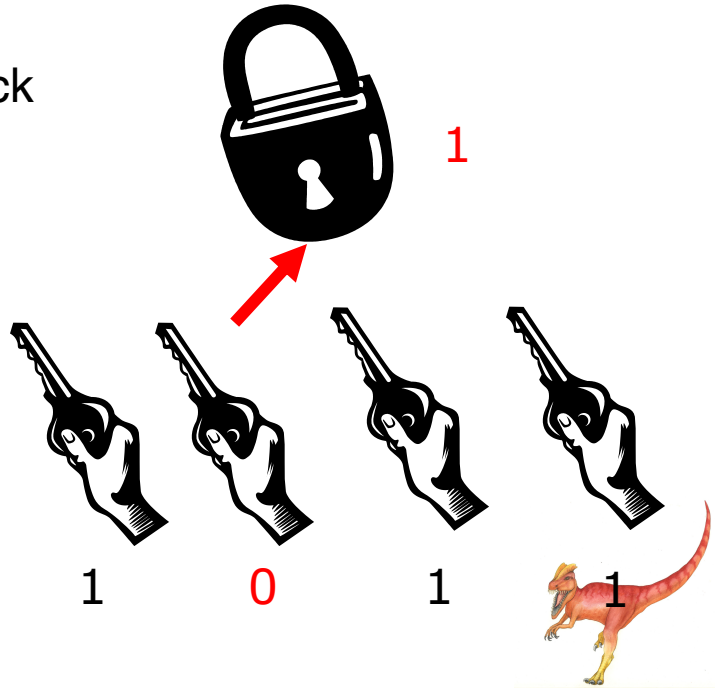
0



1 1 1 1

Thread Using swap Instruction

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);
// each thread has a local copy of key
HardwareData key = new HardwareData(true);

    while (true) {
                key.set(true);
                do {
                    lock.swap(key);    //acquire lock
                }
                while (key.get() == true);

                // criticalSection
                    lock.set(false);   //release lock
                //remainder section
}
```

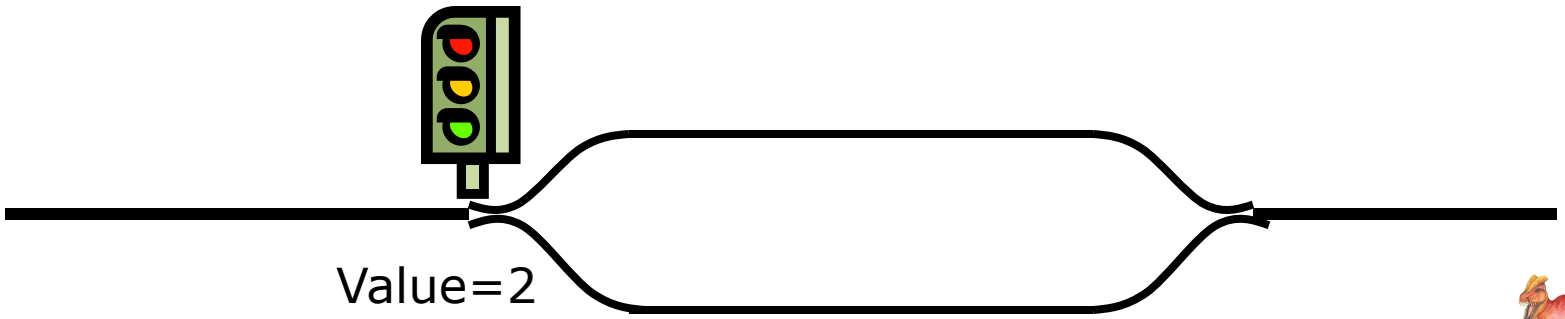


Semaphore

- Synchronization tool that does not require busy waiting
- Semaphores are like integers, except NO negative values
- Accessed only through two standard operations acquire() and release() **or** wait() and signal(). Originally called P() –to test and V() – to increment
- Can only be accessed via two **indivisible (atomic)** operations
  - acquire () {
    while (value <=0)
    ; // no-op
    value--;
    }
  - release () {
    value ++;
    }

Semaphores Like Integers Except

- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore sem = new Semaphore(1);     //  initialized to 1
do {
    sem.acquire();
        // Critical Section
    sem.release();
        // remainder section
} while (TRUE);
```

Example: Use Semaphore to enforce order

- Consider two concurrently running processes P1 with statement S1 and P2 with statement S2.
- If we require that S2 be executed only after S1 has completed.
- We let P1 and P2 share a common semaphore synch, initialized to 0.

In process P1:

```
S1;
synch.release();
```

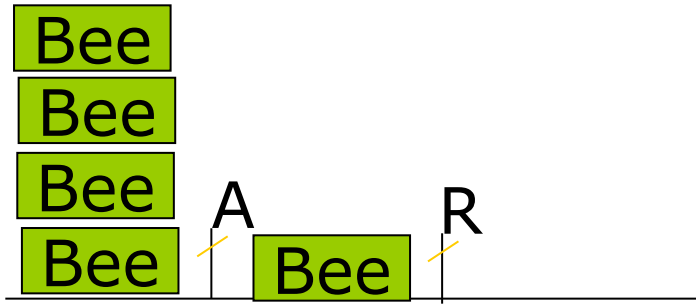
In process P2:

```
synch.acquire();
S2;
```

Thread Using Semaphore

```
public class Worker implements Runnable {
   private Semaphore sem;
   private String name;
   public Worker(Semaphore sem, String name) {
      this.sem = sem;
      this.name = name;
   }
   public void run() {
      while (true) {
         sem.acquire();
         MutualExclusionUtilities.criticalSection(name);
         sem.release();
         MutualExclusionUtilities.nonCriticalSection(name);
} } }
```

```
public class SemaphoreFactory {
   public static void main(String args[]) {
      Semaphore sem = new Semaphore(1);
      Thread[] bees = new Thread[5];
      for (int i = 0; i < 5; i++)
         bees[i] = new Thread(
                        new Worker(sem,
                        "Worker " + (new Integer(i)).toString() ));
      for (int i = 0; i < 5; i++)
         bees[i].start();
} }
```



Semaphore Implementation

- Main disadvantage of the semaphore definition given here is that it requires **busy waiting**.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles. This type of semaphore is also called **spinlock**.
- Advantage is that no context switch. Used on multiprocessor system.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
  - an integer value
  - a list of processes

- Two operations:
  - block – place a process into a waiting queue associated with the semaphore.
  - wakeup – changes the process from the waiting state to ready state.

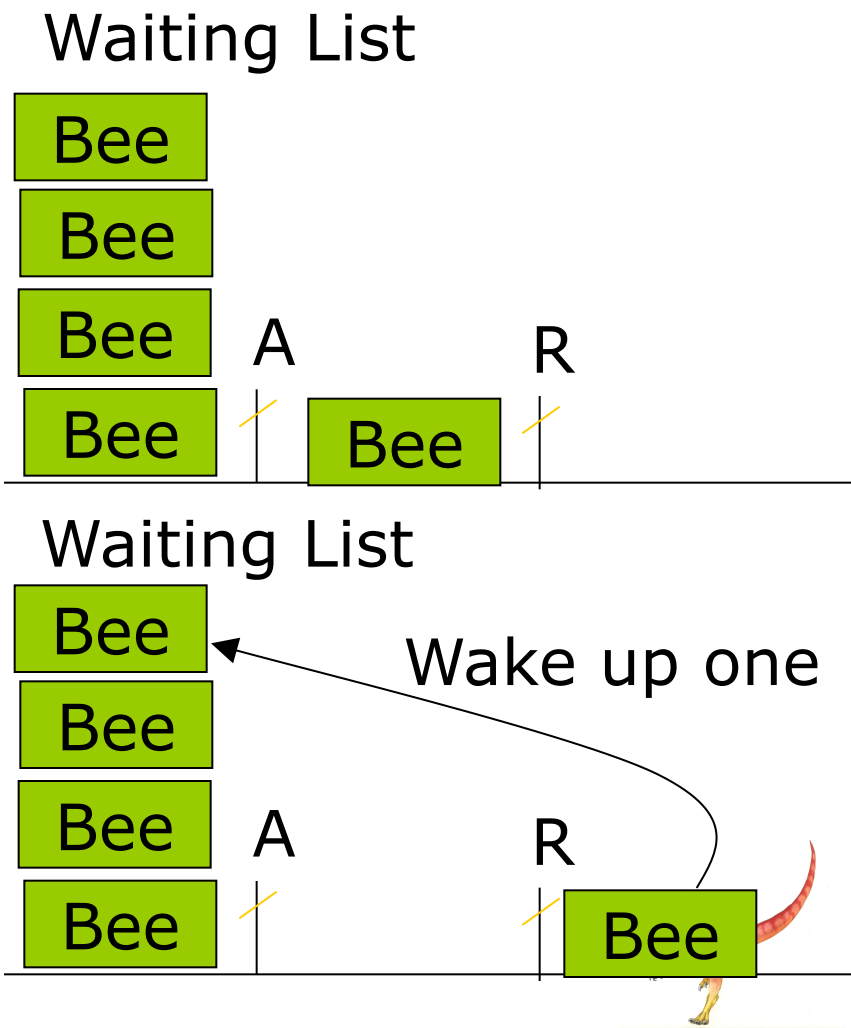
Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
acquire () {
        value--;
        if (value < 0) {
            add this process to list;
            block();  }
}
```

- Implementation of signal:

```
release () {
        value++;
        if (value <= 0) {
            remove a process P from list;
            wakeup(P);    }
}
```



Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
S.acquire();	Q.acquire();
Q.acquire();	S.acquire();
.	.
.	.
.	.
S.release();	Q.release();
W.release();	S.release();

- P0 executes S.acquire() and then P1 executes Q.acquire().
- When P0 executes Q.acquire(), it must wait until P1 executes Q.release().
- Similarly, when P1 execute S.acquire(), it must wait until P0 executes S.release()
- Since these signal operations cannot be executed, P0 and P1 are deadlocked.

Deadlock and Starvation

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

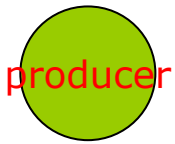
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

```
public class BoundedBuffer  {
    public BoundedBuffer( ) {
        // buffer is initially empty
        in = 0;  out = 0;
        buffer = new Object[BUFFER_SIZE];
        mutex = new Semaphore( 1 );
        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore( 0 );
    }
    public void insert( ) { /* see next slides */  }
    public Object remove( ) { /* see next slides */  }
    private static final int   BUFFER_SIZE = 5;
    private Semaphore mutex, empty, full;
    private int in, out;
    private Object[] buffer;
}
```

// Shared buffer can store five objects.
// mutex allows only one thread to enter
// empty blocks producer while empty=0
// full blocks consumer while full=0



BoundedBuffer.insert(Object item)

```
public void insert(Object item) {
    // blocked while empty = 0, check if there are empty buffers
    empty.acquire();

    // blocked while someone is using mutex, (i.e., in CS)
    mutex.acquire();

    // add an item to the buffer, this is CS
      buffer[in] = item;
      in = (in + 1) % BUFFER_SIZE;

    // releasing mutex, (i.e., exited from CS)
    mutex.release();

    full.release();          // increment full
}
```

BoundedBuffer.remove()

```
public Object remove( ) {
    // block consumer while full = 0, check if there are items in the buffer
    full.acquire();

    // blocked while someone is using mutex, (i.e., in CS)
    mutex.acquire();

    // remove an item from the buffer, this is CS
    Object item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();        // releasing mutex, (i.e., exited from CS)
    empty.release();        // increment empty
    return item; }
```

Producer Threads

```
import java.util.Date;
public class Producer implements Runnable {
   private Buffer buffer;

   public Producer(Buffer buffer) {
      this.buffer = buffer;
   }

   public void run() {
      Date message;
      while (true) {
         // nap for awhile
         SleepUtilities.nap();
         // produce an item & enter it into the buffer
         message = new Date();
         buffer.insert(message);
} } }
```

Consumer Threads

```
public class Consumer implements Runnable {
        private Buffer buffer;

        public Consumer(Buffer buffer) {
            this.buffer = buffer;
        }

        public void run() {
            Date message;
            while (true) {
                // nap for awhile
                SleepUtilities.nap();
                // consume an item from the buffer
                message = (Date)buffer.remove();

}   }   }
```

Bounded Buffer Problem: Factory

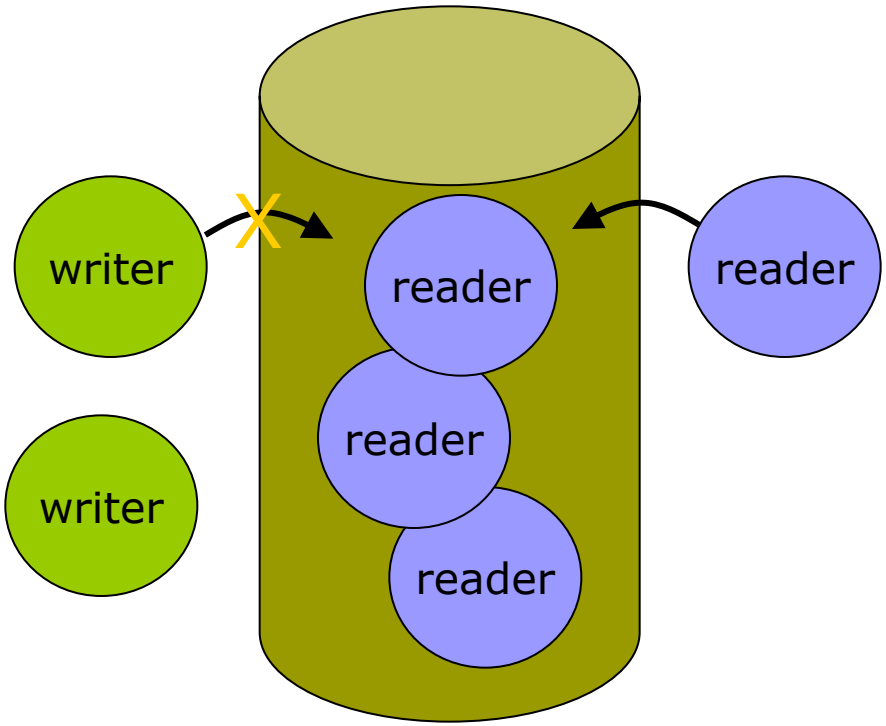
```
public class Factory
{
   public static void main(String args[]) {
        Buffer buffer = new BoundedBuffer();

        // now create the producer and consumer threads
        Thread producer = new Thread(new Producer(buffer));
        Thread consumer = new Thread(new Consumer(buffer));
        producer.start();
        consumer.start();
   }
}
```

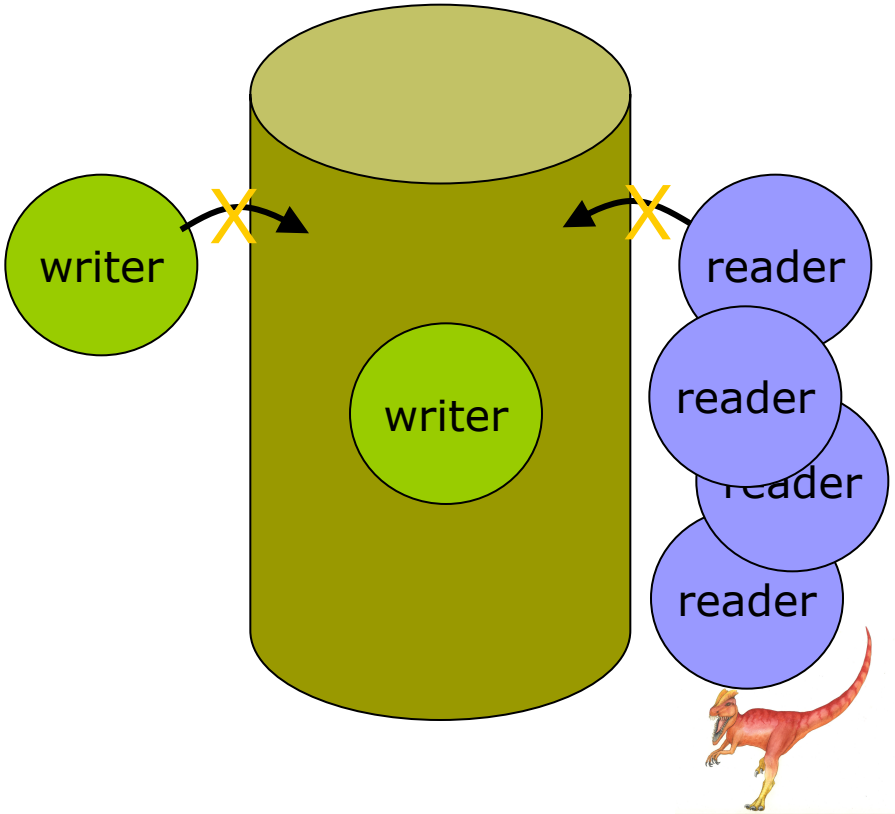
Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1 to protect update of readcount
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

The Readers-Writers Problem

- Multiple readers or a single writer can use DB.





Database

```
public class Database implements ReadWriteLock {
    public Database( ) {
        readerCount = 0;                    // # readers in database access
        // ensure mutual exclusion when reader count is updated
        mutex = new Semaphore(1);
        //mutual exclusion for writers and prevent writers from enter database when
              //readers are reading
        db = new Semaphore(1);
}
    public void acquireReadLock( ) {    }
    public void releaseReadLock( ) {    }
    public void acquireWriteLock( ) {    }
    public void releaseWriteLock( ) {    }
    private int readerCount;
    private Semaphore mutex;
    private Semaphore db;
}
```

Readers

```
Public void acquireReadLock( ) {
        mutex.acquire();
        /* the first reader indicates that
the database is being read */
        ++readerCount;
        if (readerCount == 1)
                db.acquire();

        mutex.release();
}
```

```
Public void releaseReadLock( ) {
        mutex.acquire();
        /* the last reader indicates that
the database is no longer being read */
        --readerCount;

        if (readerCount == 0)
                db.release();

        mutex.release();
}
```

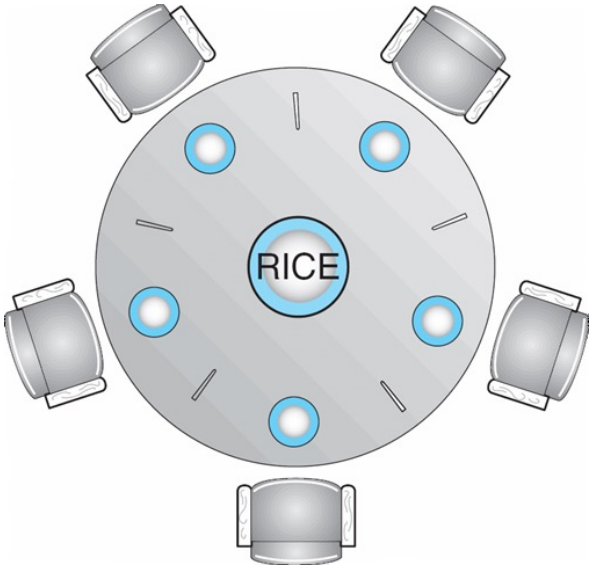
Writers

```
public void acquireWriteLock() {
    db.acquire();
}

public void releaseWriteLock() {
    db.release();
}
```

If a writer is active in the database and n readers are waiting, then one reader is queued on db and n-1 readers are queued on mutes.
If a writer executes db.release(), we may resume the execution of either the waiting readers or a single waiting writer.

Dining Philosophers Problem

- A hungry philosopher picks up two chopsticks closest to her to eat.
- A philosopher picks one chopstick at a time.
- Can not pick up a chopstick that is already in the hand of a neighbor.



Thinking
Hungry
Eating

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopStick[] = new Semaphore[5];
  - Semaphore chopStick [i] initialized to 1

Dining Philosophers Problem

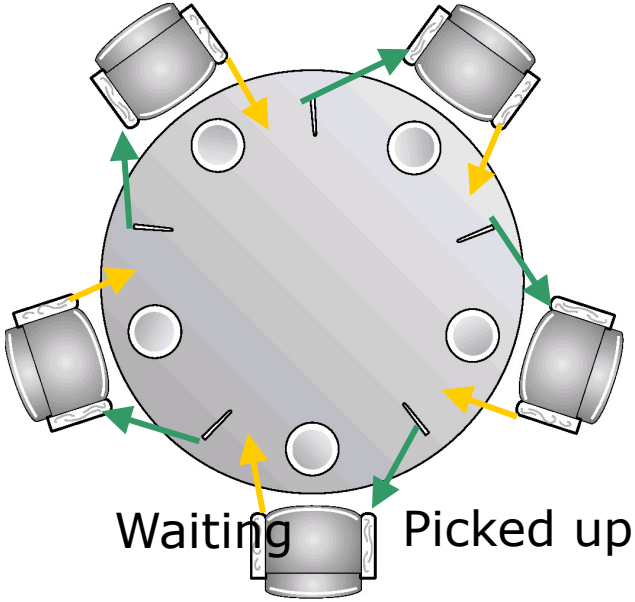
- Significance
  - Is an examples of a large class of concurrency-control problems.
  - Is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
  - Simple solution is to represent each chopstick with a semaphore

```
Semaphore chopStick[] = new Semaphore[5];
for (int i = 0; i < 5; i++)
    chopStick [i] = new Semaphore(1);
```

The Structure of Philosopher i

- Philosopher *i*

```
while ( true ) {
// get left chopstick
chopStick[i].acquire();
// get right chopstick
chopStick[(i + 1) % 5].acquire();

eating();

//return left chopstick
chopStick[i].release( );
// return right chopstick
chopStick[(i + 1) % 5].release( );

thinking();
}
```



A deadlock occurs!

If **all** five philosophers grabs her **left** chopsticks **simultaneously**.

Possible Remedies

- Placing restrictions on the philosophers
  - Allow at most four philosophers to be sitting simultaneously at the table
  - Allow a philosopher to pick up her chopsticks only if both are available
  - An odd philosopher picks up first her left chopsticks and even philosopher picks up her right chopstick and then right

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements **a variety of locks** to support multitasking, multithreading (including real-time threads), and multiprocessing
- For short code-segment
  - Uses adaptive mutexes for efficiency when protecting data from short code segments
  - In multiple CPUs, the thread spin-and-wait if lock is held by a thread running in another CPU; the thread block-and-sleep if the thread holding the lock is not in run state.
  - In single CPU, the thread always sleep rather than spin

Solaris Synchronization

- For longer code segment
  - Uses condition variables and readers-writers locks when longer sections of code need access to data
  - Read-writer locks are more efficient than semaphores because multiple threads can read data concurrently whereas semaphores always serialize access to the data.
  - Uses turnstile which is a queue structure containing threads blocked on a lock.

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

End of Chapter 16