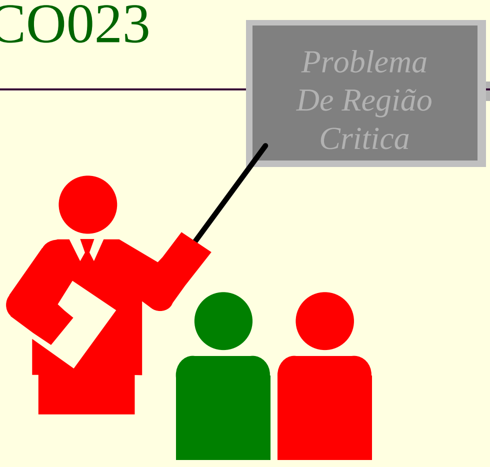
Modulo I - Sincronização Programação Concorrente

Prof. Ismael H F Santos

Ementa

- Programação Concorrente - Processos e Threads
  - The Critical-Section Problem
  - Two-task software solution
  - Peterson's Solution
  - Synchronization Hardware
  - Semaphores
  - Classic Problems of Synchronization
    - Bounded-Buffer
    - Readers and Writers
    - Dining Philosophers
  - Monitors
  - Java Synchronization
  - Synchronization Examples
  - Atomic Transactions
  - Client Server Communication

SCD – CO023



Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Background

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

Race Condition

- count++ could be implemented as

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

- count-- could be implemented as

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

Race Condition (cont.)

- 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}

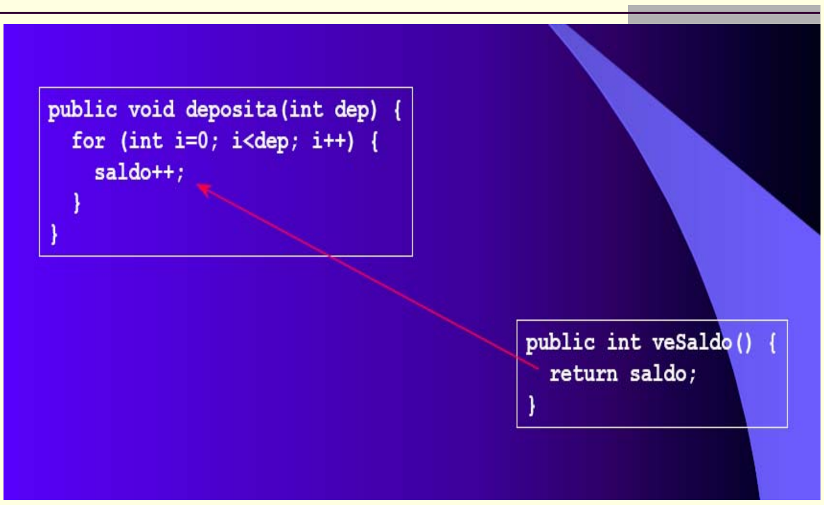
Outro Exemplo RaceCondition

- Exemplo: acesso a saldo bancário

```
public class Conta {
  private int saldo;
  public Conta (int ini) {
     saldo = ini;
  }
  public int veSaldo() {
    return saldo;
  }
  public void deposita(int dep) {
    for (int i=0; i<dep; i++) {
        try {
          Thread.sleep(10); // da chance ao escalonador!
        }
        catch(InterruptedException exception) {
          System.err.println(exception);
        }
      saldo++;
    }
  }
}
```

```
public class ThreadsEnxeridas {
  public static void main(String[] args) {
    int repet = 20;
    Conta cc = new Conta(0);
    (new ThreadEnxerida("1", cc, repet)).start();
    (new ThreadEnxerida("2", cc, repet)).start();
    (new ThreadEnxerida("3", cc, repet)).start();
    (new ThreadEnxerida("4", cc, repet)).start();
  }
}
public class ThreadEnxerida extends Thread {
  private int vezes;
  private Conta cc;
  public ThreadEnxerida(String id, Conta a, int qtas) {
     super(id);  cc = a;  vezes = qtas;
  }
  public void run() {
     for (int i=0; i<vezes; i++) {
        System.out.println("thread" + getName() +
        "-saldo: " + cc.veSaldo());
        cc.deposita(100);
     }
  }
}
```

Captura de estados Captura de estados inconsistentes

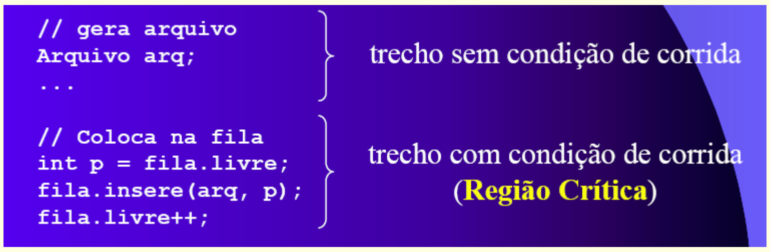


```
public void deposita(int dep) {
  for (int i=0; i<dep; i++) {
    saldo++;
  }
}
```

```
public int veSaldo() {
  return saldo;
}
```

Condições de Corrida

- Situação onde o resultado final depende do momento em que ocorrerem as trocas de contexto



```
// gera arquivo
Arquivo arq;
...
```

trecho sem condição de corrida

```
// Coloca na fila
int p = fila.livre;
fila.insere(arq, p);
fila.livre++;
```

trecho com condição de corrida (**Região Crítica**)

Solution to Critical-Section Problem

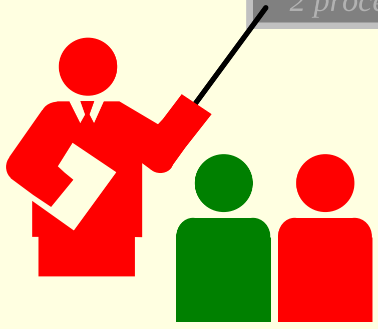
1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

Solution to Critical-Section Problem

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

SCD – CO023

Solução Software p/ 2 processos



Two-task Solution

- Two tasks, T_0 and T_1 (T_i and T_j)
- Three solutions presented. All implement this MutualExclusion interface:

```
public interface MutualExclusion
{
    public static final int TURN_0 = 0;
    public static final int TURN_1 = 1;

    public abstract void enteringCriticalSection(int turn);
    public asbtract void leavingCriticalSection(int turn);
}
```

Algorithm Factory class

- Used to create two threads and to test each algorithm

```
public class AlgorithmFactory {
    public static void main(String args[]) {
      MutualExclusion alg = new Algorithm 1();
      Thread first = new Thread( new Worker("Worker 0", 0, alg));
      Thread second = new Thread(new Worker("Worker 1", 1, alg));

      first.start();
      second.start();
    }
}
```

Worker Thread

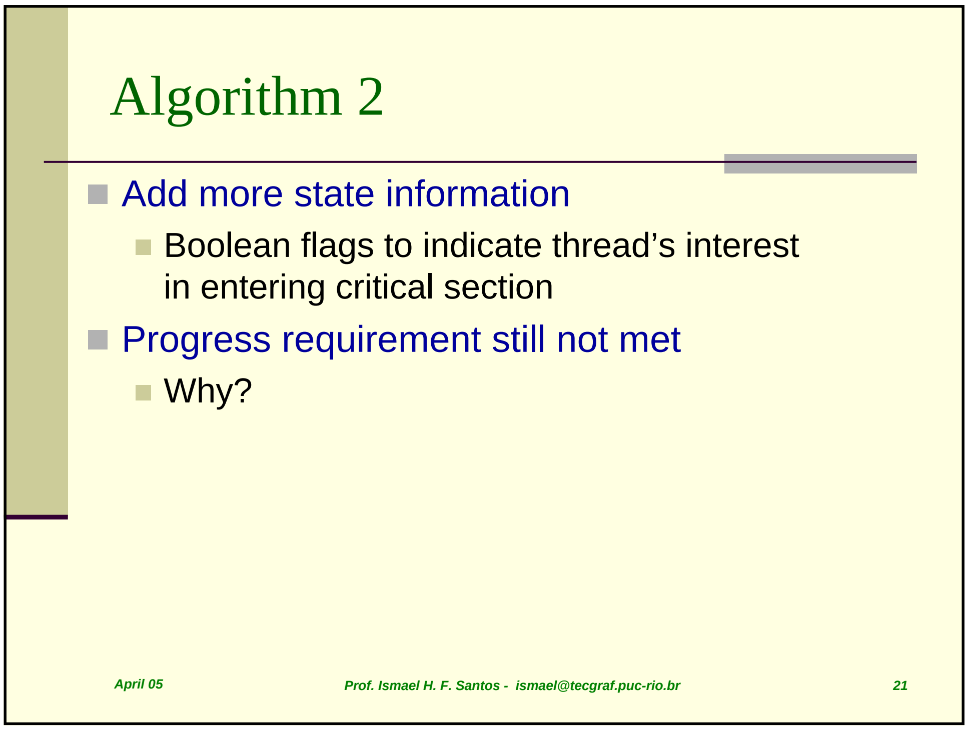
```
public class Worker implements Runnable {
    private String name;
    private int id;
    private MutualExclusion mutex;

    public Worker(String name, int id, MutualExclusion mutex) {
          this.name = name;  this.id = id;   this.mutex = mutex;
    }
    public void run() {
          while (true) {
                    mutex.enteringCriticalSection(id);
                    MutualExclusionUtilities.criticalSection(name);
                    mutex.leavingCriticalSection(id);
                    MutualExclusionUtilities.nonCriticalSection(name);
          }
    }
}
```

Algorithm 1

- Threads share a common integer variable turn
- If turn==i, thread i is allowed to execute
- Does not satisfy progress requirement
  - Why?

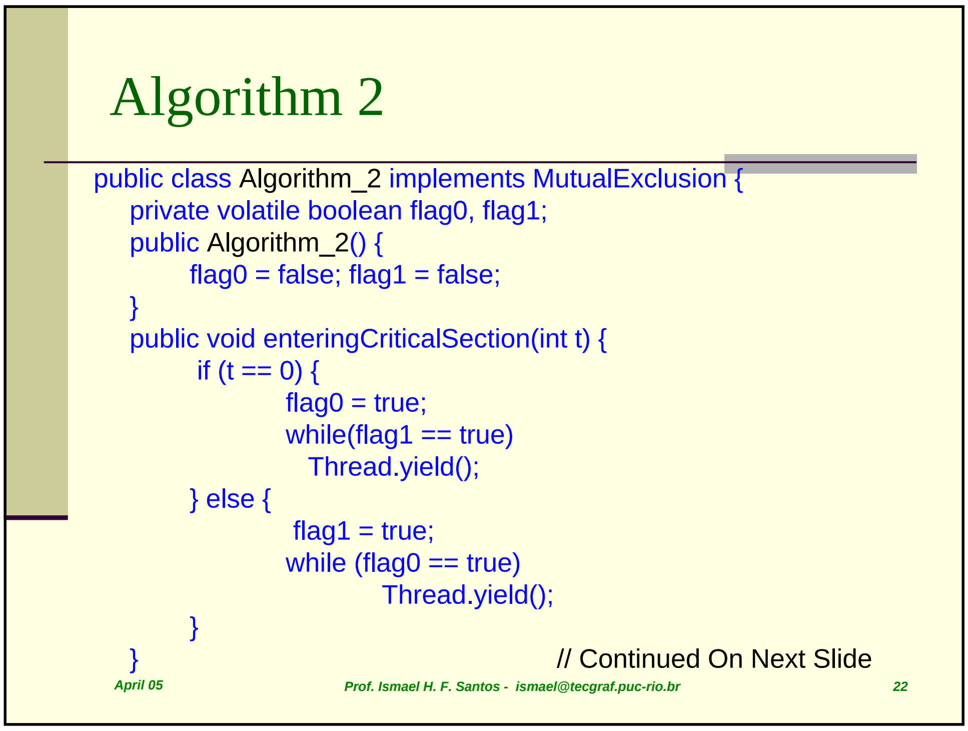
Algorithm 1

```
public class Algorithm_1 implements MutualExclusion {
   private volatile int turn;

   public Algorithm 1() {
        turn = TURN_0;
   }
   public void enteringCriticalSection(int t) {
        while (turn != t)
           Thread.yield();
   }
   public void leavingCriticalSection(int t) {
        turn = 1 - t;
   }
}
```



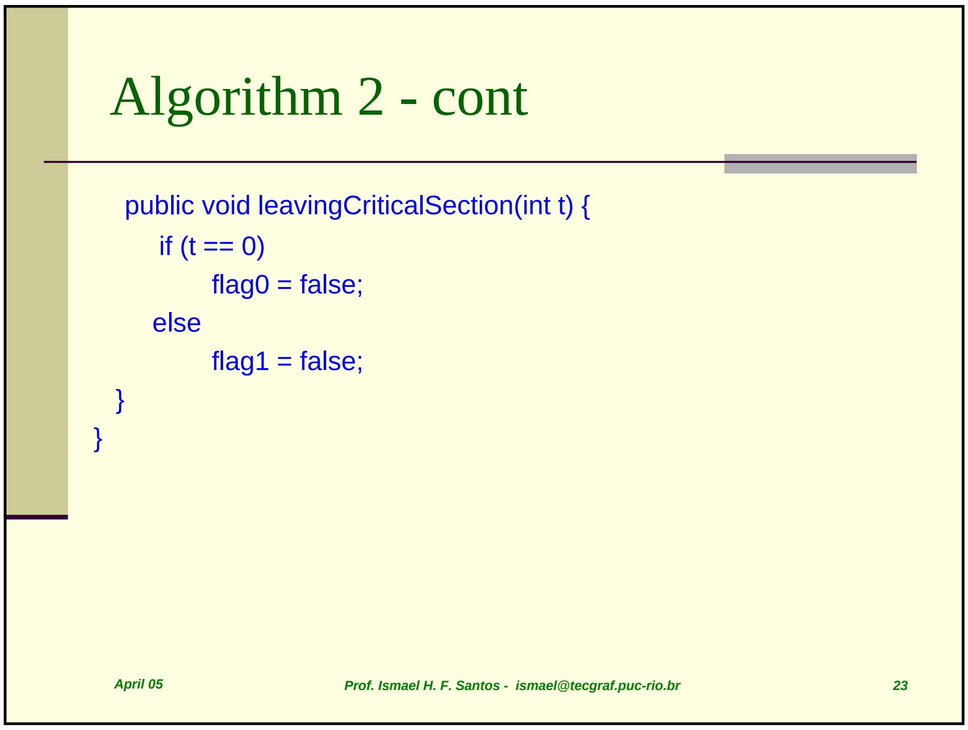
Algorithm 2

- Add more state information
  - Boolean flags to indicate thread’s interest in entering critical section
- Progress requirement still not met
  - Why?



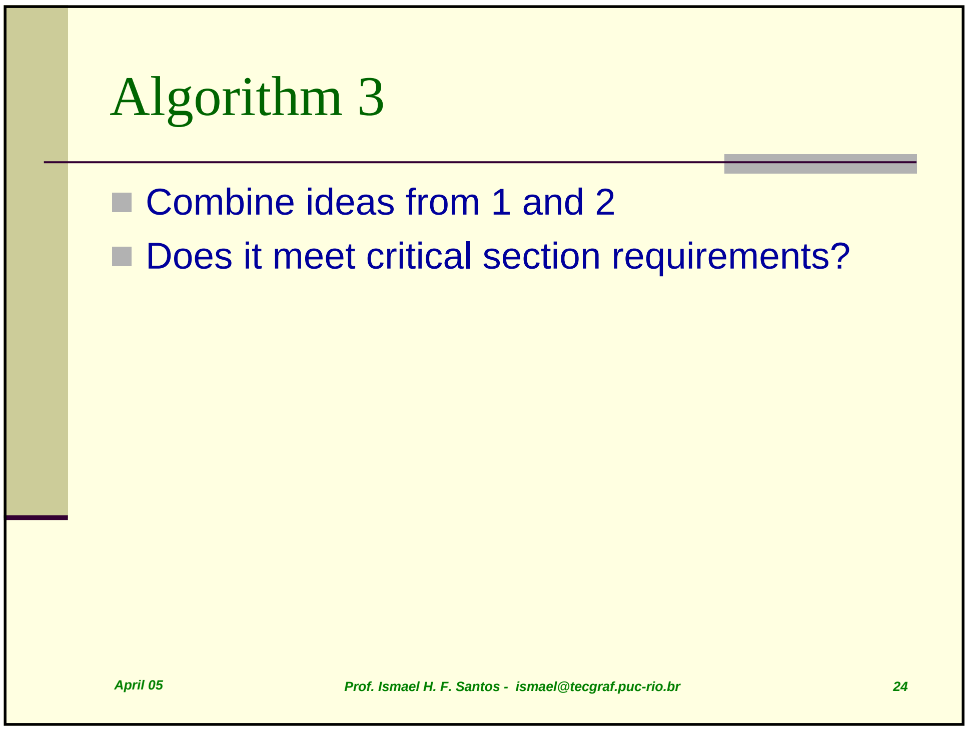
Algorithm 2

```
public class Algorithm_2 implements MutualExclusion {
    private volatile boolean flag0, flag1;
    public Algorithm_2() {
          flag0 = false; flag1 = false;
    }
    public void enteringCriticalSection(int t) {
           if (t == 0) {
                    flag0 = true;
                    while(flag1 == true)
                      Thread.yield();
          } else {
                     flag1 = true;
                    while (flag0 == true)
                              Thread.yield();
          }
    }                                       // Continued On Next Slide
```



Algorithm 2 - cont

```
 public void leavingCriticalSection(int t) {
     if (t == 0)
            flag0 = false;
    else
            flag1 = false;
 }
}
```



Algorithm 3

- Combine ideas from 1 and 2
- Does it meet critical section requirements?

Algorithm 3

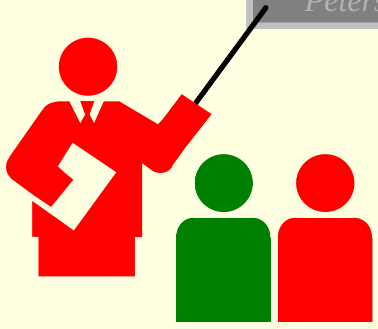
```
public class Algorithm_3 implements MutualExclusion {
    private volatile boolean flag0;
    private volatile boolean flag1;
    private volatile int turn;
    public Algorithm_3() {
          flag0 = false; flag1 = false; turn = TURN_0;
     }

     public void leavingCriticalSection(int t) {
        if (t == 0)
            flag0 = false;
        else
            flag1 = false;
     }                                             // Continued on Next Slide
```

Algorithm 3 - enteringCriticalSection

```
  public void enteringCriticalSection(int t) {
         int other = 1 - t;  turn = other;
         if (t == 0) {
                   flag0 = true;
                   while(flag1 == true && turn == other)
                      Thread.yield();
         }
         else {
                   flag1 = true;
                   while (flag0 == true && turn == other)
                       Thread.yield();
         }
  }
}
```

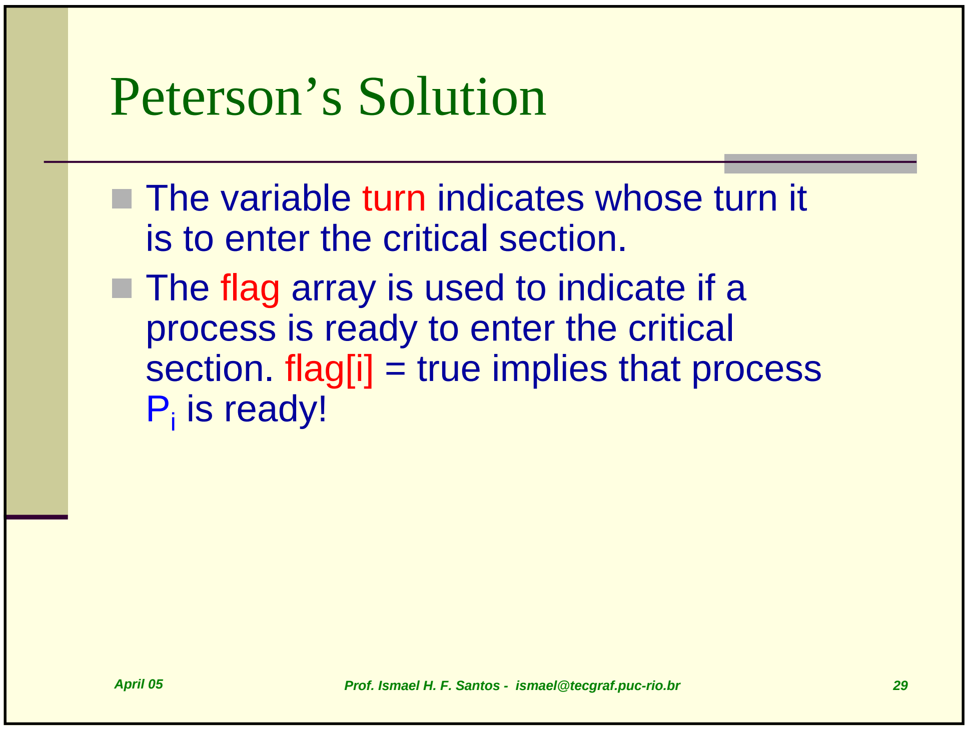
SCD – CO023

Solução
De
Peterson



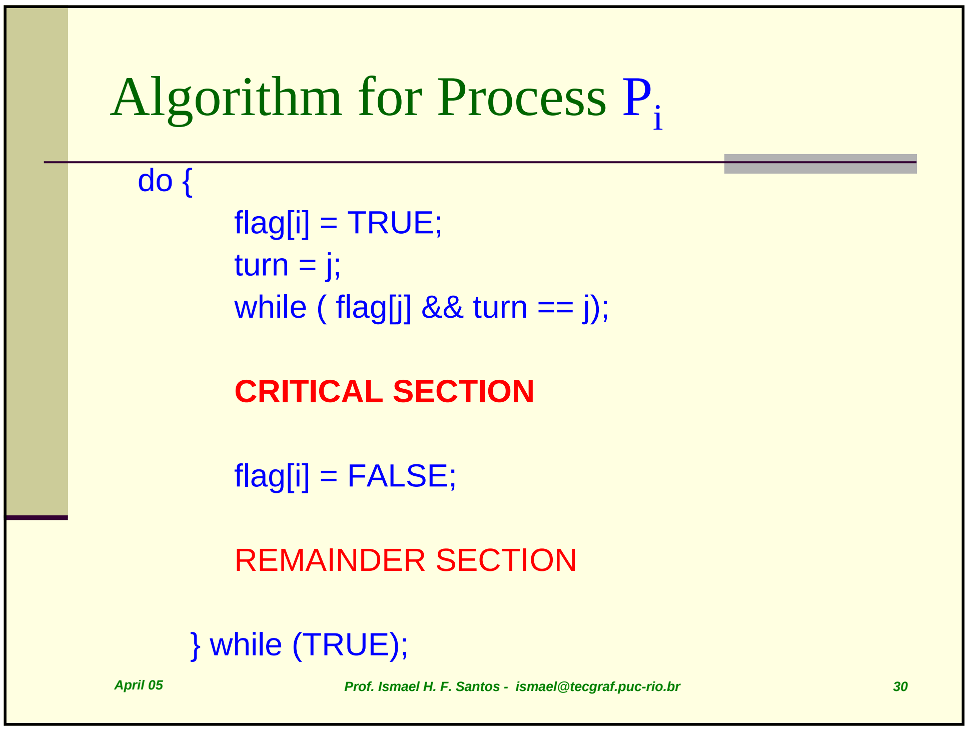
Peterson’s Solution

- 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]



Peterson’s Solution

- 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

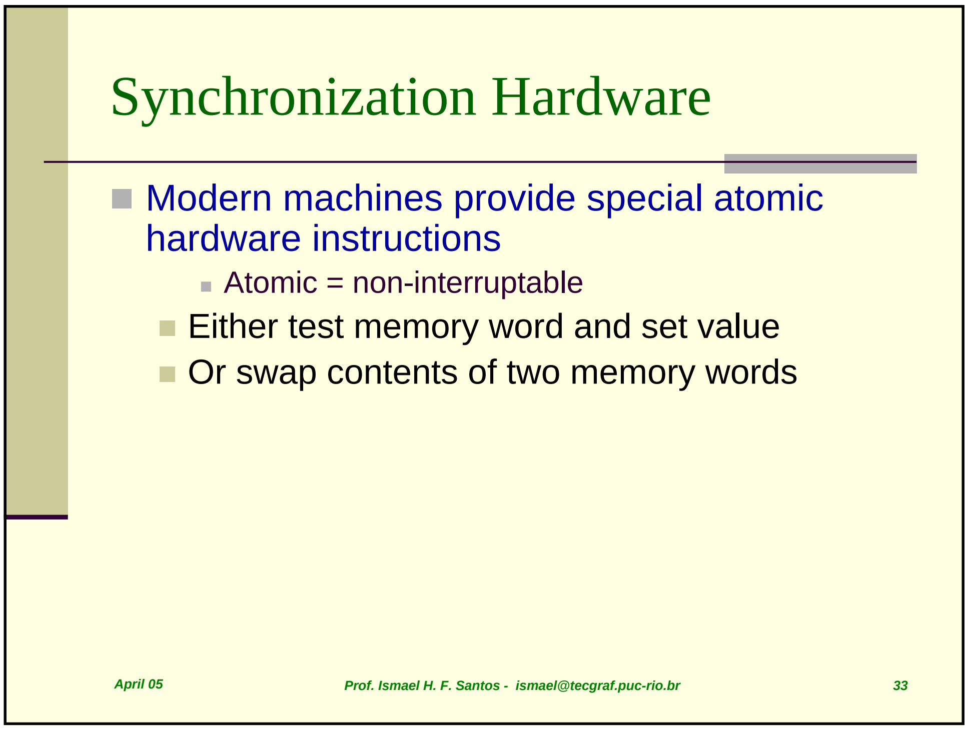
```
do {
        flag[i] = TRUE;
        turn = j;
        while ( flag[j] && turn == j);

        CRITICAL SECTION

        flag[i] = FALSE;

        REMAINDER SECTION

    } while (TRUE);
```

SCD – CO023

Sincronização com Hardware

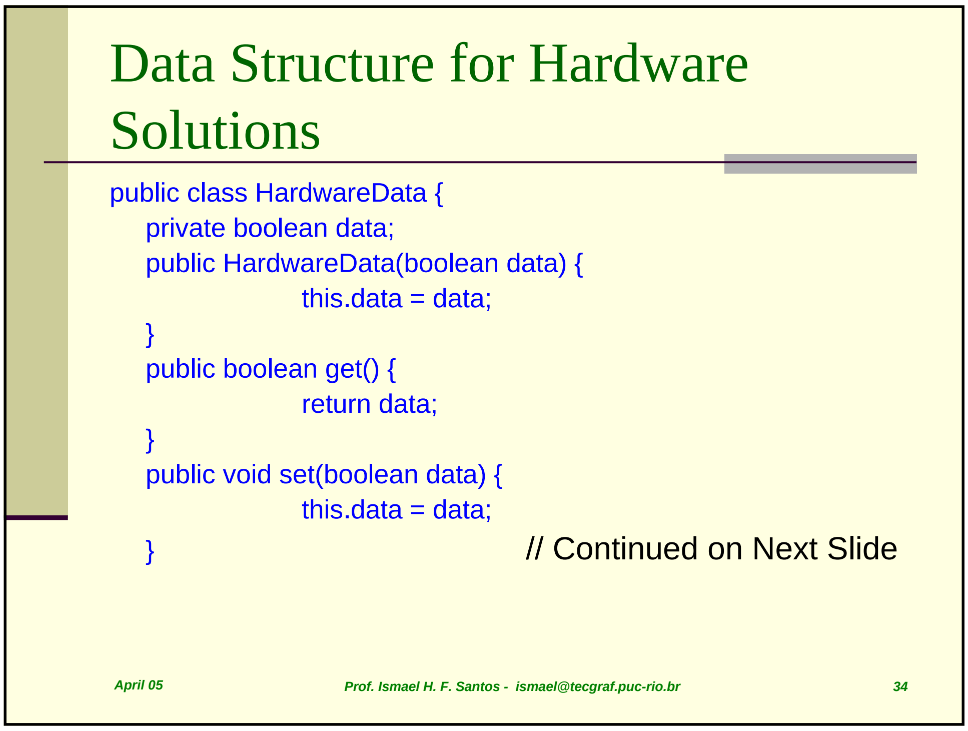
Synchronization Hardware

- 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



Synchronization Hardware

- 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

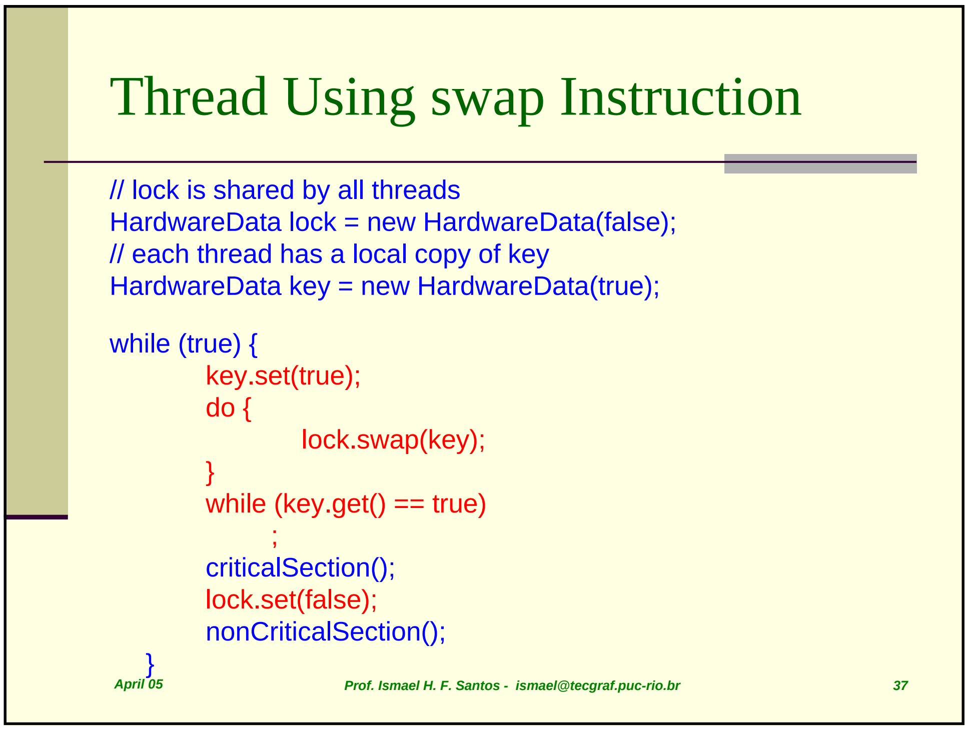
```
public class HardwareData {
    private boolean data;
    public HardwareData(boolean data) {
                  this.data = data;
    }
    public boolean get() {
                  return data;
    }
    public void set(boolean data) {
                  this.data = data;
    }                   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

Data Structure for Hardware Solutions - cont

```
  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);
  }
}
```

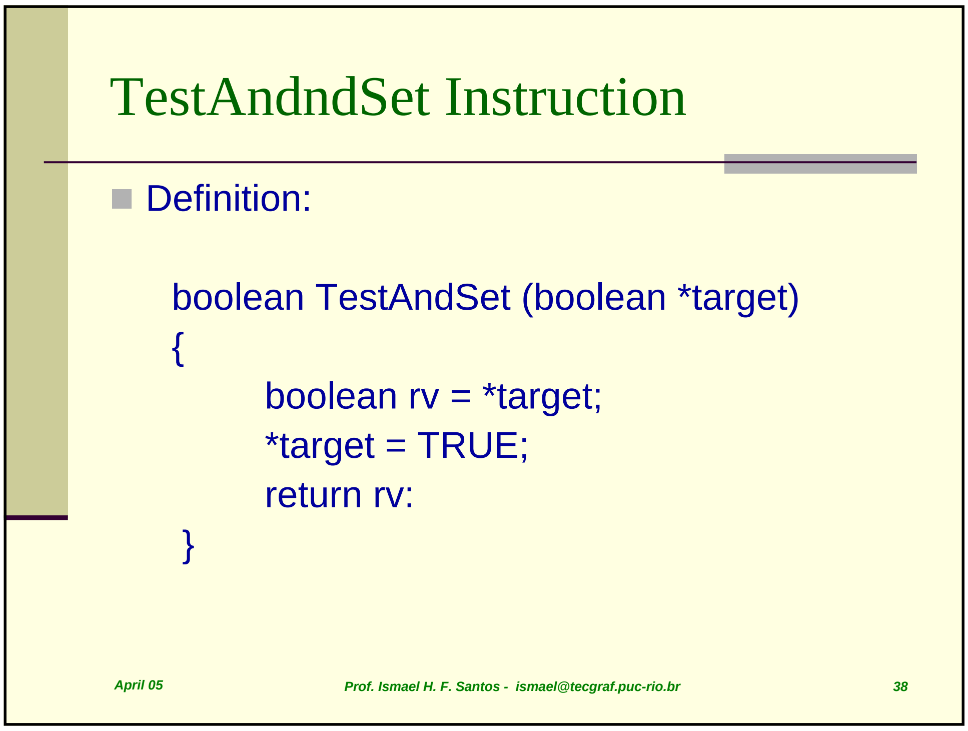
Thread Using get-and-set Lock

```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);
while (true) {
   while (lock.getAndSet(true))
                 Thread.yield();
   criticalSection();
   lock.set(false);
   nonCriticalSection();
}
```



Thread Using swap Instruction

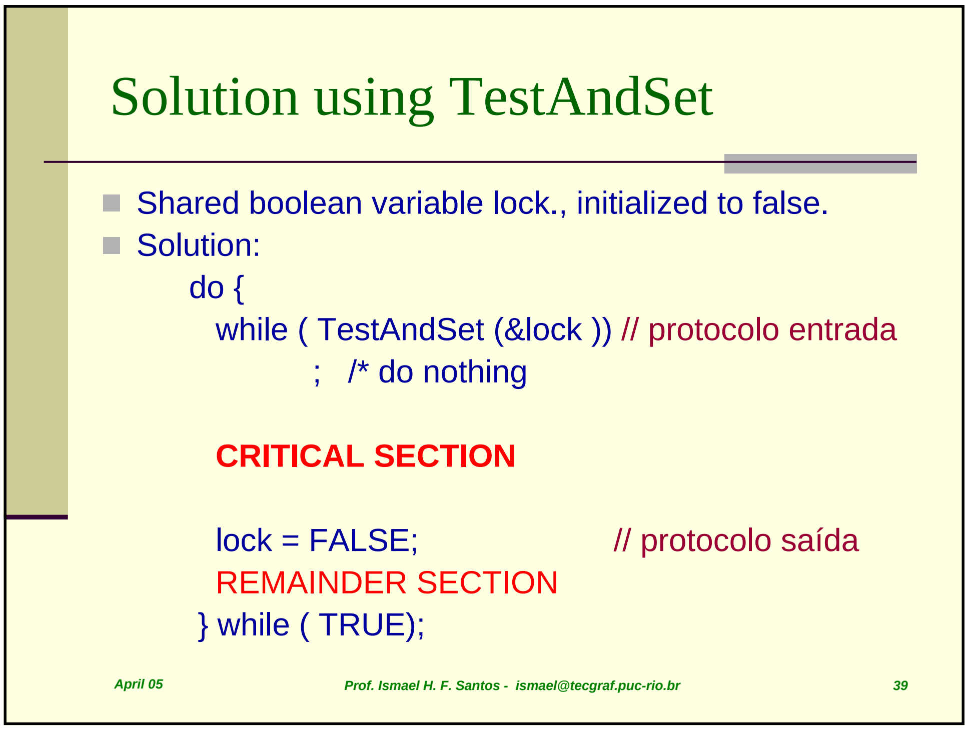
```
// lock is shared by all threads
HardwareData lock = new HardwareData(false);
// each thread has a local copy of key
HardwareData key = new HardwareData(true);

while (true) {
        key.set(true);
        do {
                lock.swap(key);
        }
        while (key.get() == true)
              ;
        criticalSection();
        lock.set(false);
        nonCriticalSection();
   }
```



TestAndndSet Instruction

- Definition:

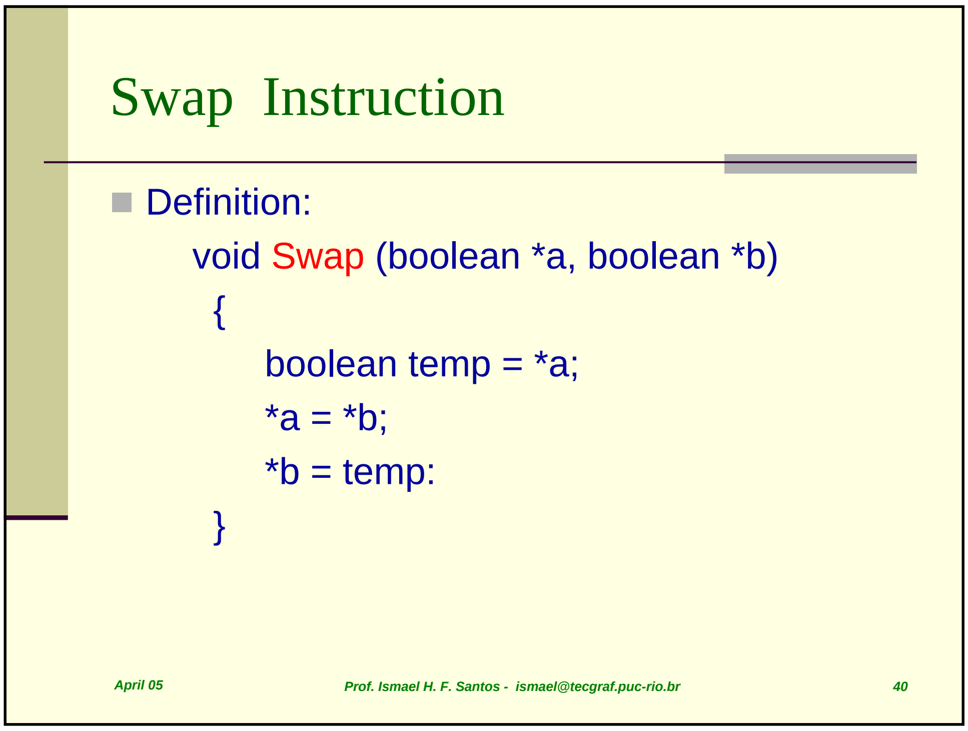
```
boolean TestAndSet (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv:
 }
```



Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
     do {
        while ( TestAndSet (&lock )) // protocolo entrada
                  ;   /* do nothing

        CRITICAL SECTION

        lock = FALSE;                  // protocolo saída
        REMAINDER SECTION
      } while ( TRUE);
```



Swap Instruction

- Definition:

```
     void Swap (boolean *a, boolean *b)
       {
            boolean temp = *a;
            *a = *b;
            *b = temp:
       }
```

Solution using Swap

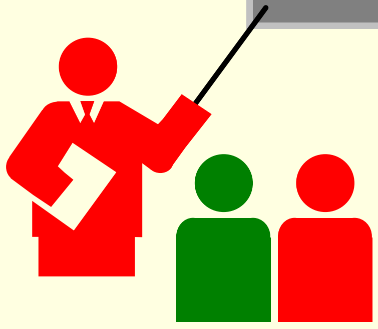
- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

Solution using Swap

```
do {
        key = TRUE;
        while ( key == TRUE)
            Swap (&lock, &key ); // protocolo entrada

        CRITICAL SECTION

        lock = FALSE;            // protocolo saída

        REMAINDER SECTION

} while ( TRUE);
```

SCD – CO023

Semaforos



Semaphore

- Synchronization tool that does not require busy waiting (also called spin lock)
- Semaphore *S* – integer variable
- Two standard operations modify S: acquire() and release()
  - Originally called P() and V()
- Less complicated

Semaphore

- Can only be accessed via two indivisible (atomic) operations
  - acquire() {

```
        while cont <= 0
            ; // no-op
          cont--;
    }
```

  - release() {

```
      cont++;
    }
```

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

Semaphore as General Synchronization Tool

- Provides mutual exclusion

```
Semaphore s;    //  initialized to 1 default

s.acquire();          // protocolo entrada

CRITICAL SECTION

s.release();           // protocolo saída
```

Synchronization using Semaphores Implementation - Worker

```
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);
          }
    }
}
```

Synchronization using Semaphores Implementation - SemaphoreFactory

```
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

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the crtical section.
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied

Semaphore Implementation

- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

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:
  - value (of type integer)
  - pointer to next record in the list

Semaphore Implementation with no Busy waiting

- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

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

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of signal:

```
Semaphore {
   release() {
        value++;
         if (value <= 0) {
              remove a process P from the waiting queue
               wakeup(P);
         }
```

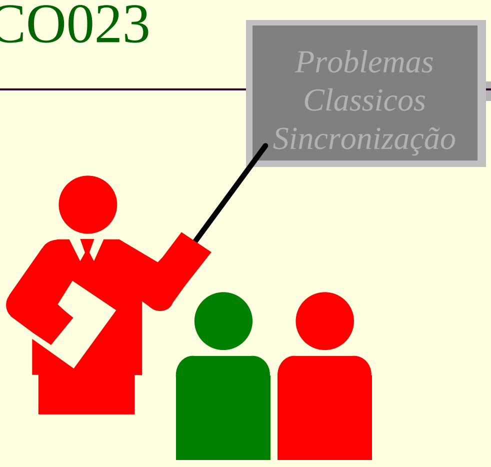
Deadlock and Starvation

- 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

```
          P0                              P1
     S.acquire();                    Q.acquire();
     Q.acquire();                    S.acquire();
          .                               .
          .                               .
          .                               .
     S.release();                    Q.release();
     Q.release();                    S.release();
```

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

SCD – CO023

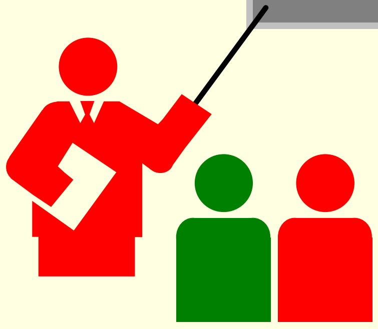


Classical Problems of Synchronization

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

SCD – CO023

Bounded Buffer



Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
    // produce an item
    empty.acquire();
    mutex.acquire();
    // add the item to the buffer
    mutex.release();
    full.release();
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    full.acquire();
    mutex.acquire();
    // remove an item from buffer
    mutex.release();
    empty.release();
    // consume the removed item
} while (true);
```

Bounded-Buffer Problem

```
public class BoundedBuffer implements Buffer {
    private static final int BUFFER SIZE = 5;
    private Object[] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    // Continued on next Slide
```

Bounded Buffer Constructor

```
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(Object item) { /* next slides */ }

public Object remove() { /* next slides */ }
}
```

Bounded Buffer Problem: insert() Method

```
public void insert(Object item) {
   empty.acquire();
   mutex.acquire();
   // add an item to the buffer
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
   mutex.release();
   full.release();
}
```

Bounded Buffer Problem: remove() Method

```
public Object remove() {
   full.acquire();
   mutex.acquire();
   // remove an item from the buffer
   Object item = buffer[out];
   out = (out + 1) % BUFFER SIZE;
   mutex.release();
   empty.release();
   return item;
}
```

Bounded Buffer Problem: Producer

```
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);
          }
    }
}
```

Bounded Buffer Problem: Consumer

```
import java.util.Date;
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();
    }
}
```

SCD – CO023

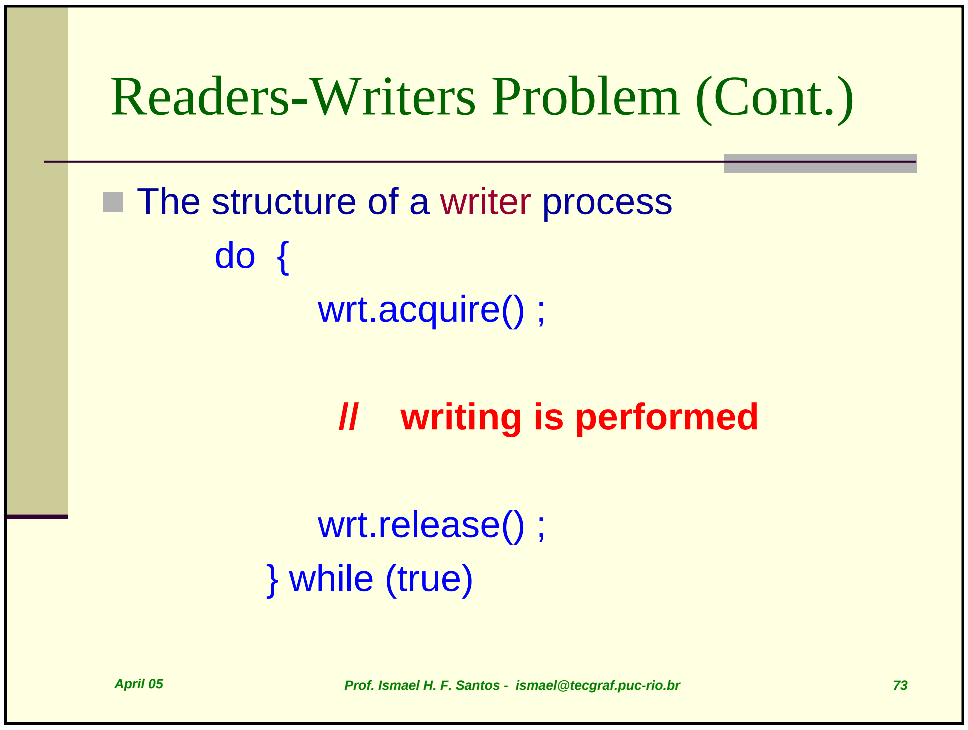
Readers
And
Writers

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.

Readers-Writers Problem (1ª solução)

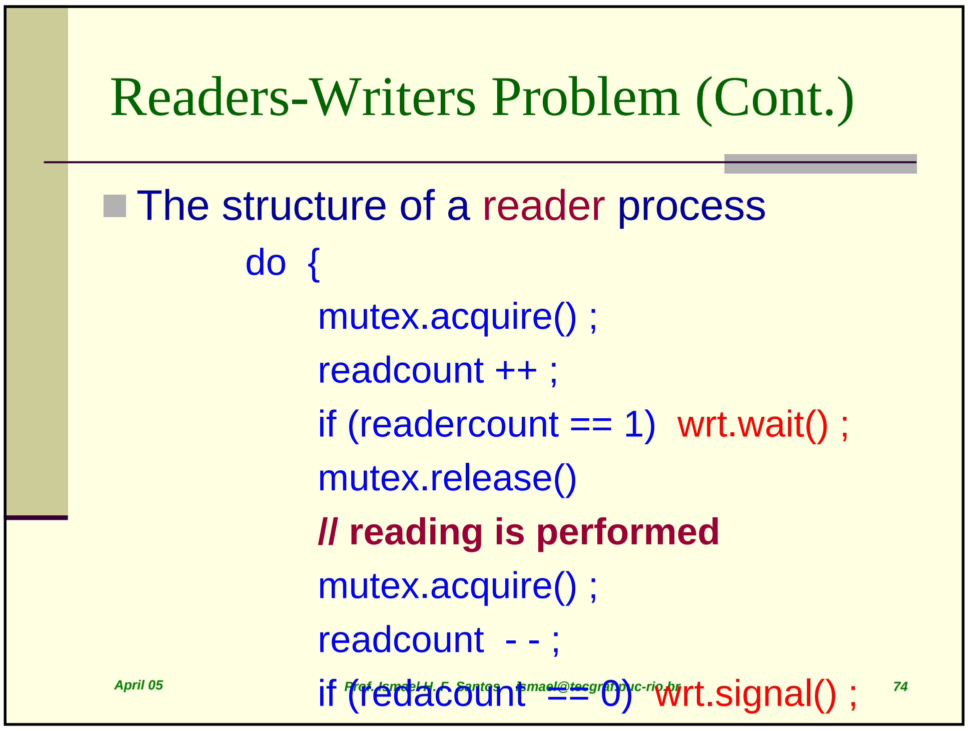
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.



Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do  {
        wrt.acquire() ;

          //    writing is performed

        wrt.release() ;
    } while (true)
```



Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do  {
      mutex.acquire() ;
      readcount ++ ;
      if (readercount == 1)  wrt.wait() ;
      mutex.release()
      // reading is performed
      mutex.acquire() ;
      readcount  - - ;
      if (redacount  == 0)  wrt.signal() ;
```

Readers-Writers Problem (2ª solução)

- Shared Data
  - Data set
  - Interface RWLock
    - Methods: acquire/release Read or Write lock.
  - Semaphore mutex initialized to 1
  - Semaphore db initialized to 1.
  - Integer readcount initialized to 0.

Readers-Writers Problem: Interface

```
public interface RWLock {
   public abstract void acquireReadLock();
   public abstract void acquireWriteLock();
   public abstract void releaseReadLock();
   public abstract void releaseWriteLock();
}
```

Readers-Writers Problem: Reader

```
public class Reader implements Runnable {
    private RWLock db;
    public Reader(RWLock db) {  this.db = db;      }
    public void run() {
          while (true) { // nap for awhile
                    db.acquireReadLock();
                    // you now have access to read from the database
                    // read from the database
                    db.releaseReadLock();
          }
    }
}
```

Readers-Writers Problem: Writer

```
public class Writer implements Runnable {
    private RWLock db;
    public Writer(RWLock db) {
                    this.db = db;
    }
    public void run() {
          while (true) {
                    db.acquireWriteLock();
                    // you have access to write to the database
                    // write to the database
                    db.releaseWriteLock();
          }
    }
}
```

Readers-Writers Problem: Database

```
public class Database implements RWLock {
    private int readerCount;
    private Semaphore mutex;
    private Semaphore db;
    public Database() {
          readerCount = 0;
          mutex = new Semaphore(1);
          db = new Semaphore(1);
    }
    public int acquireReadLock() { /* next slides */ }
    public int releaseReadLock() {/* next slides */ }
    public void acquireWriteLock() {/* next slides */ }
    public void releaseWriteLock() {/* next slides */ }
}
```

Readers-Writers Problem: Methods called by readers

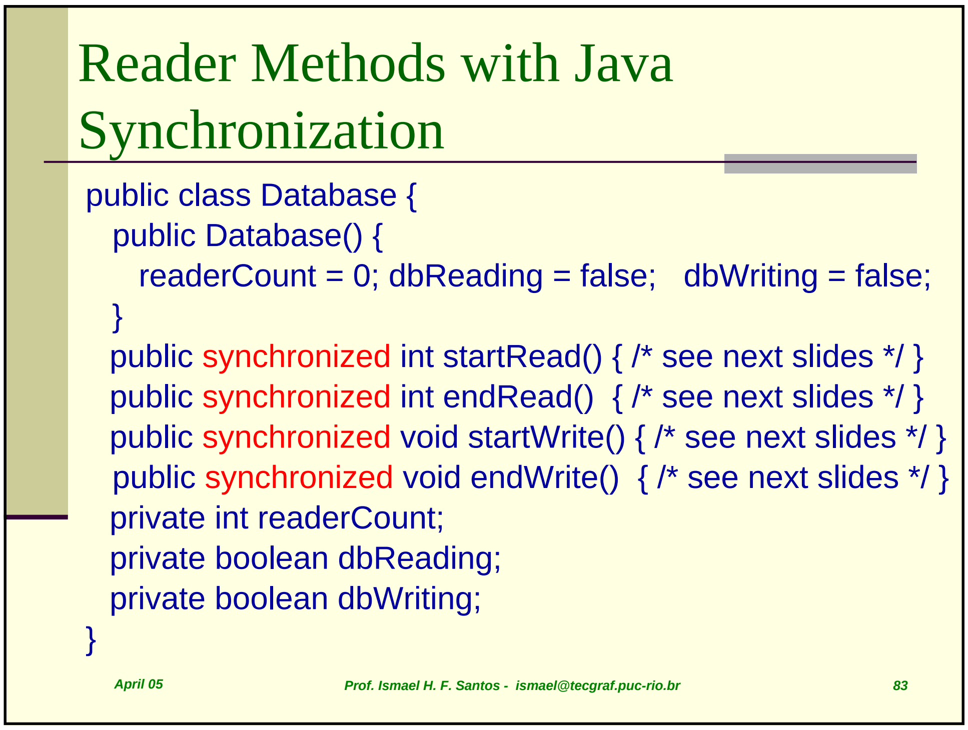
```
public void acquireReadLock() {
    mutex.acquire();
    ++readerCount;
    // if I am the first reader tell all others that the database is being read
    if (readerCount == 1)  db.acquire();
    mutex.release();
}

public void releaseReadLock() {
    mutex.acquire();
    --readerCount;
    // if I am the last reader tell all others that the database is
    // no longer being read
    if (readerCount == 0) db.release();
    mutex.release();
}
```

Readers-Writers Problem: Methods called by writers

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

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

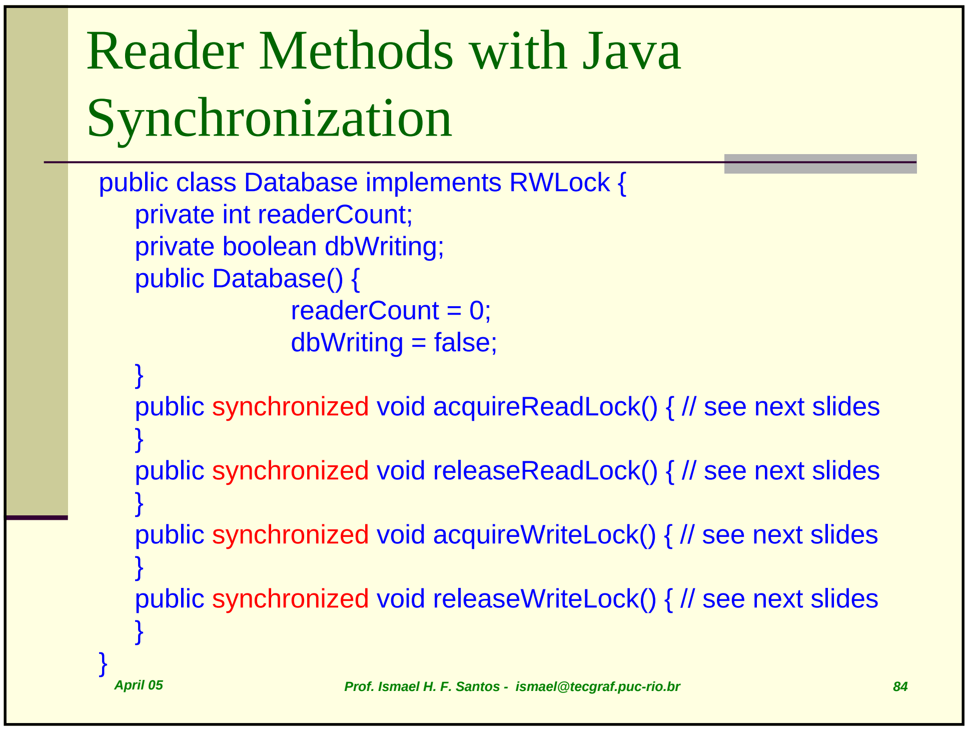
Readers-Writers Problem (3ª solução)

- Shared Data
  - Data set
  - Flags dbReading and dbWriting initialized to false.
  - Integer readcount initialized to 0.



Reader Methods with Java Synchronization

```
public class Database {
   public Database() {
      readerCount = 0; dbReading = false;   dbWriting = false;
   }
   public synchronized int startRead() { /* see next slides */ }
   public synchronized int endRead()  { /* see next slides */ }
   public synchronized void startWrite() { /* see next slides */ }
   public synchronized void endWrite()  { /* see next slides */ }
   private int readerCount;
   private boolean dbReading;
   private boolean dbWriting;
}
```



Reader Methods with Java Synchronization

```
public class Database implements RWLock {
    private int readerCount;
    private boolean dbWriting;
    public Database() {
                    readerCount = 0;
                    dbWriting = false;
    }
    public synchronized void acquireReadLock() { // see next slides
    }
    public synchronized void releaseReadLock() { // see next slides
    }
    public synchronized void acquireWriteLock() { // see next slides
    }
    public synchronized void releaseWriteLock() { // see next slides
    }
}
```

acquireReadLock() Method

```
public synchronized void acquireReadLock() {
   while (dbWriting == true) {
               try {
                       wait();
               }
               catch(InterruptedException e) { }
   }
   ++readerCount;
}
```

releaseReadLock() Method

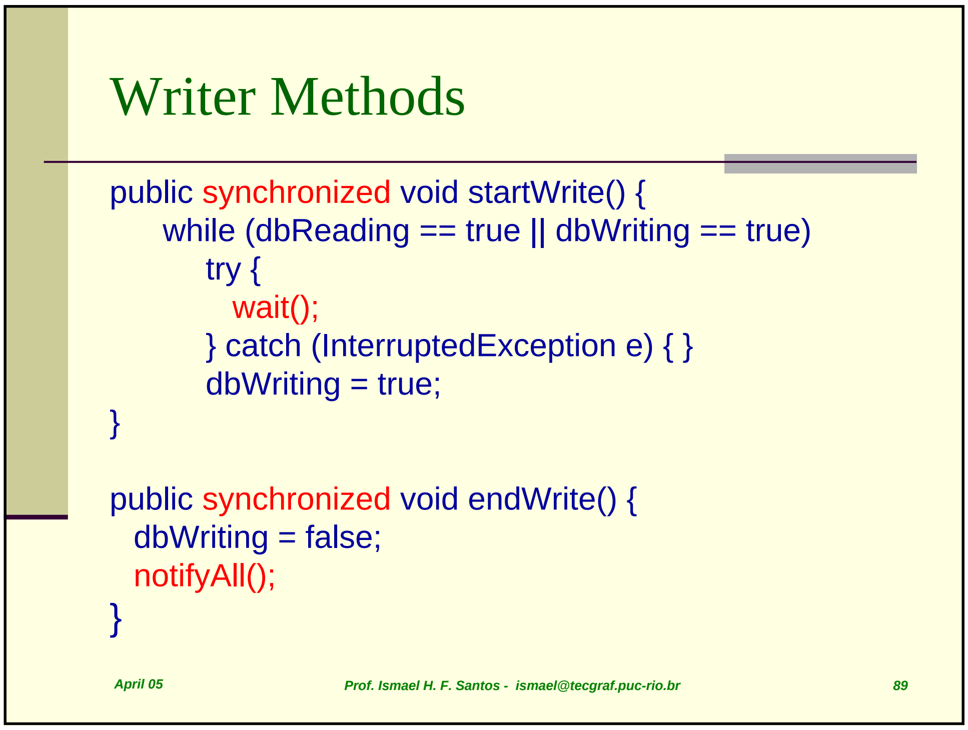
```
public synchronized void releaseReadLock() {
   --readerCount;
   // if I am the last reader tell writers
   // that the database is no longer being read
   if (readerCount == 0)
        notify();
}
```

startRead() Method

```
public synchronized int startRead() {
  while (dbWriting == true) {
        try {
          wait();
        } catch (InterruptedException e) { }
  }
  ++readerCount;
  if (readerCount == 1)
    dbReading = true;
  return readerCount;
}
```

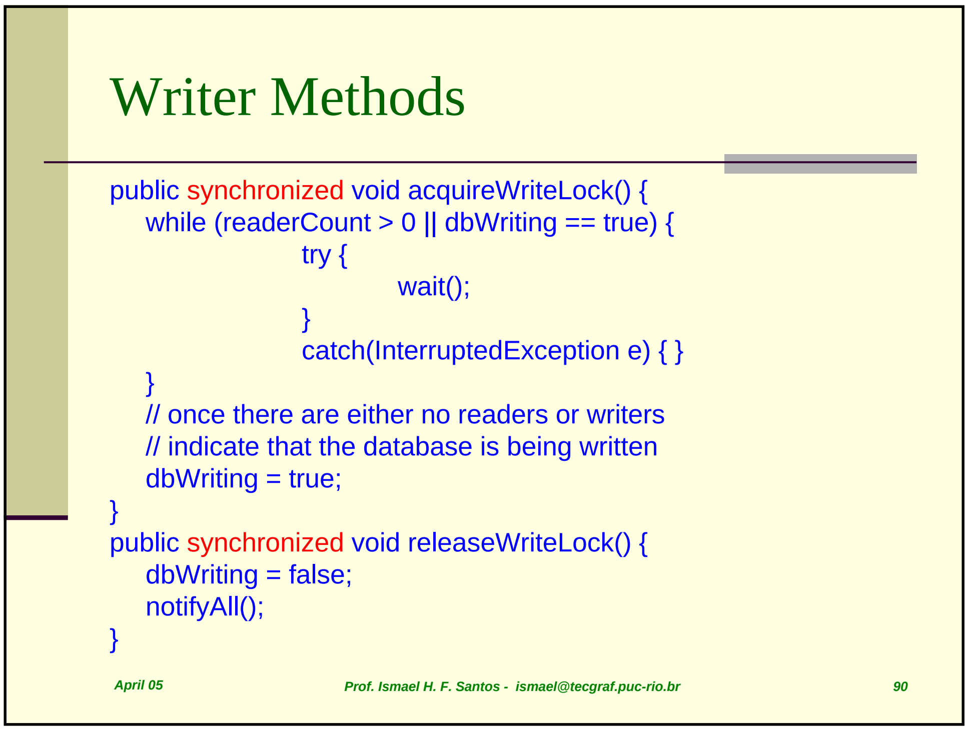
endRead() Method

```
public synchronized int endRead() {
    --readerCount
    if (readerCount == 0) {
      dbReading = false;
       notifyAll();
    }
    return readerCount;
  }
```



Writer Methods

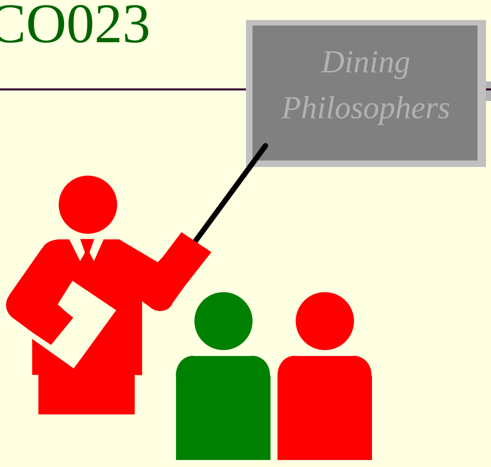
```
public synchronized void startWrite() {
     while (dbReading == true || dbWriting == true)
         try {
           wait();
         } catch (InterruptedException e) { }
         dbWriting = true;
}

public synchronized void endWrite() {
  dbWriting = false;
  notifyAll();
}
```



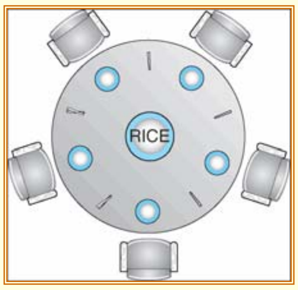
Writer Methods

```
public synchronized void acquireWriteLock() {
    while (readerCount > 0 || dbWriting == true) {
                  try {
                           wait();
                  }
                  catch(InterruptedException e) { }
    }
    // once there are either no readers or writers
    // indicate that the database is being written
    dbWriting = true;
}
public synchronized void releaseWriteLock() {
    dbWriting = false;
    notifyAll();
}
```

SCD – CO023

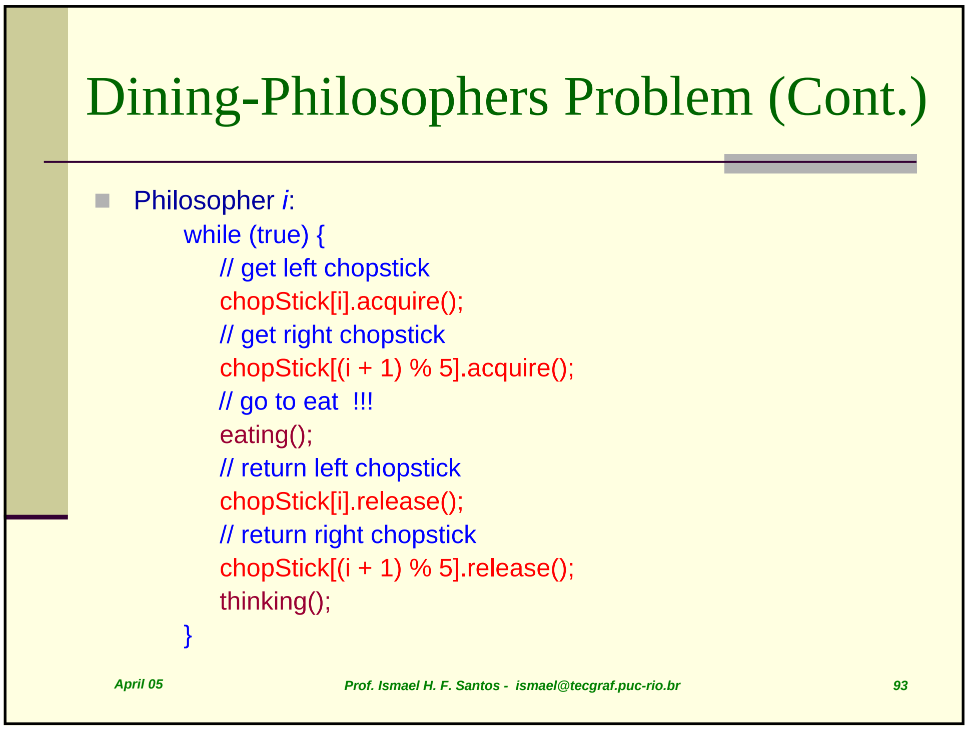


Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

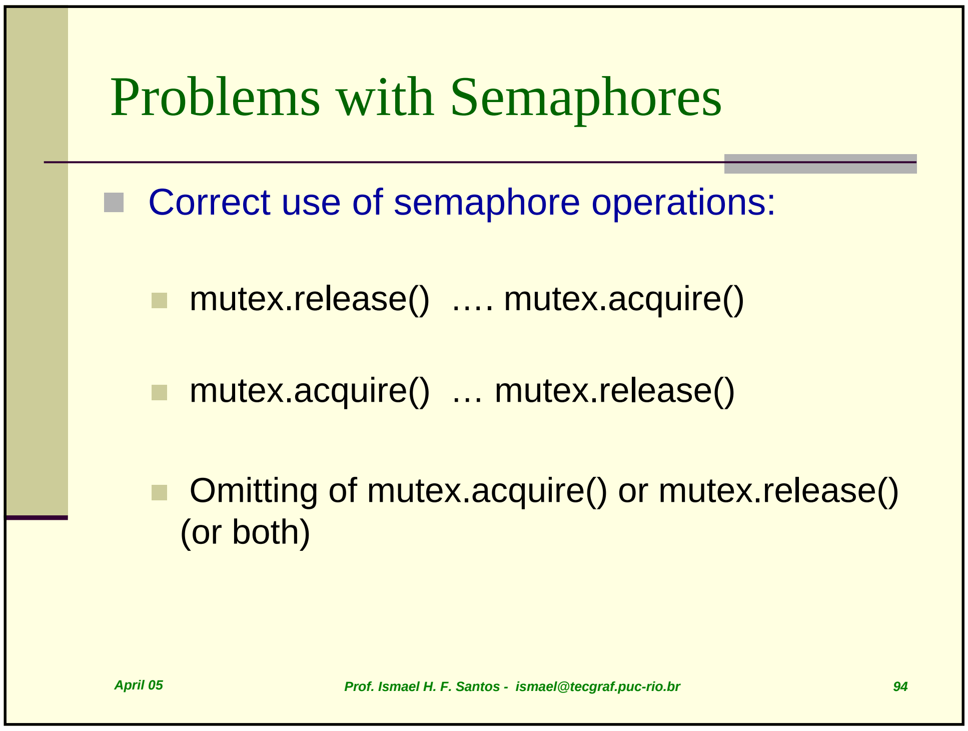
```
Semaphore chopStick[]  = new Semaphore[5];
```



Dining-Philosophers Problem (Cont.)

- Philosopher *i*:

```
while (true) {
    // get left chopstick
    chopStick[i].acquire();
    // get right chopstick
    chopStick[(i + 1) % 5].acquire();
    // go to eat  !!!
    eating();
    // return left chopstick
    chopStick[i].release();
    // return right chopstick
    chopStick[(i + 1) % 5].release();
    thinking();
}
```

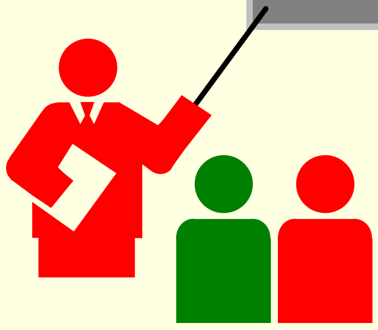


Problems with Semaphores

- Correct use of semaphore operations:
  - mutex.release() …. mutex.acquire()
  - mutex.acquire() … mutex.release()
  - Omitting of mutex.acquire() or mutex.release() (or both)

SCD – CO023

Monitores

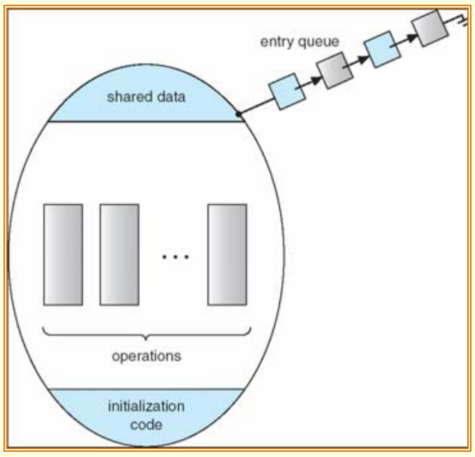


Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name {
  // shared variable declarations
  procedure P1 (…) { …. }
         …

  procedure Pn (…) {……}

  Initialization code ( ….) { … }
}
```

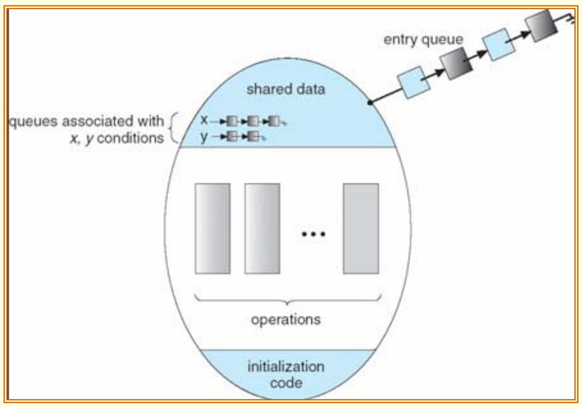
Schematic view of a Monitor



Condition Variables

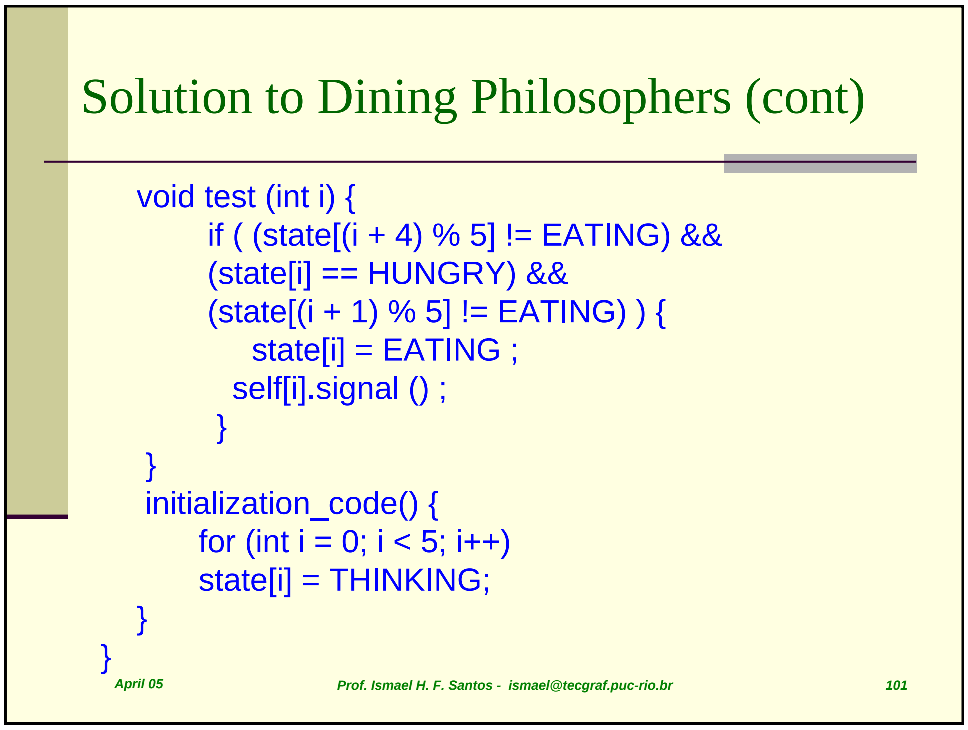
- condition x, y;
- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

Monitor with Condition Variables



Solution to Dining Philosophers

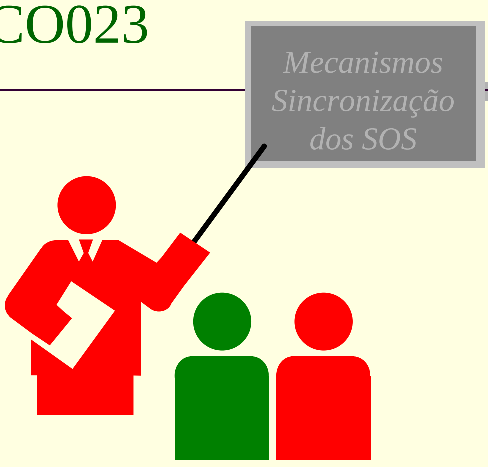
```
monitor DP  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
           state[i] = HUNGRY;   test(i);
           if (state[i] != EATING) self [i].wait;
    }
    void putdown (int i) {
           state[i] = THINKING  // test left and right neighbors
            test((i + 4) % 5);
            test((i + 1) % 5);
     }
```



Solution to Dining Philosophers (cont)

```
    void test (int i) {
            if ( (state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
                 state[i] = EATING ;
               self[i].signal () ;
             }
     }
     initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
    }
}
```

SCD – CO023

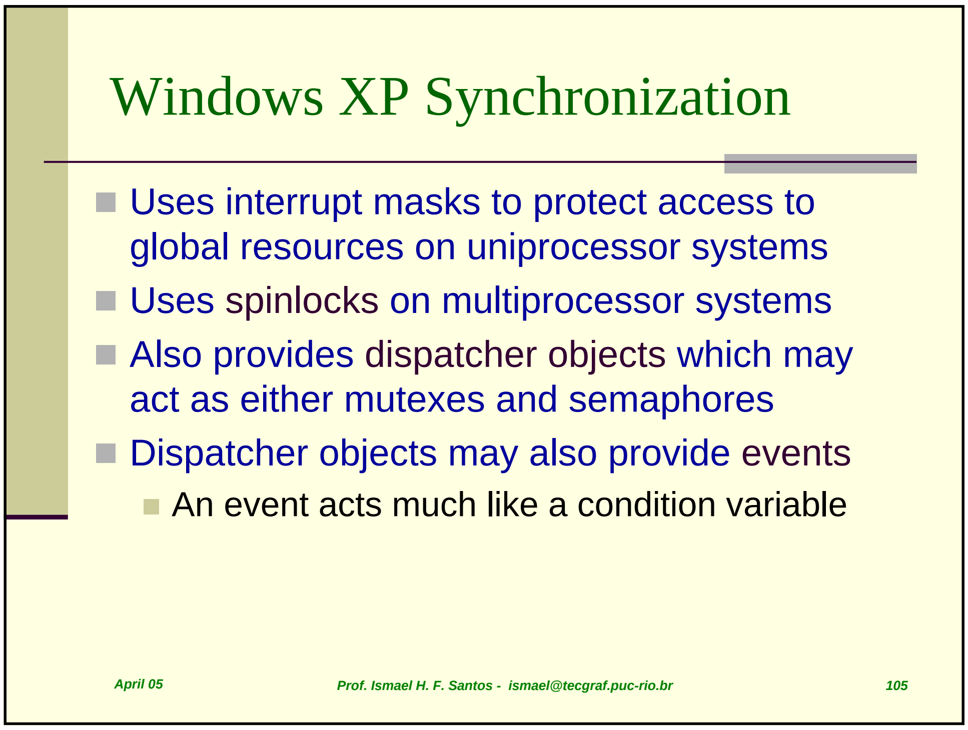


Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

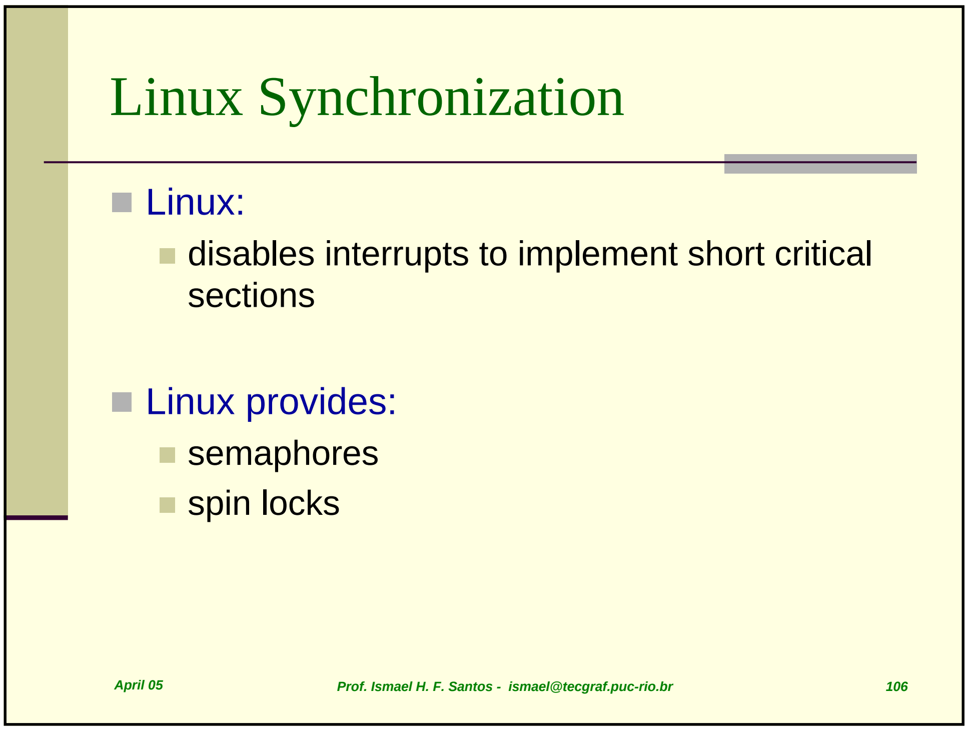
Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer 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



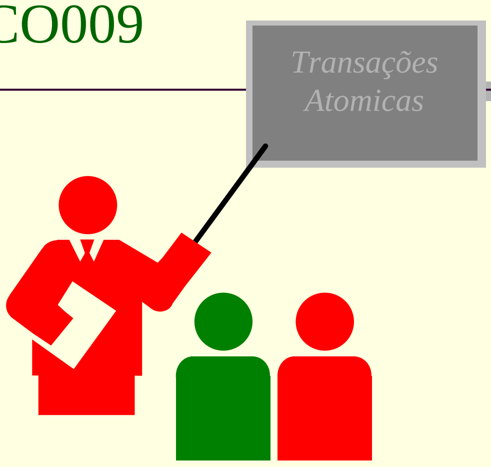
Linux Synchronization

- Linux:
  - disables interrupts to implement short critical sections
- 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

SOP – CO009



Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0; int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Insert() Method

```
while (true) {
     /* Produce an item */
      while( ((in = (in + 1) % BUFFER SIZE count)  == out)
       ;   /* do nothing -- no free buffers */
      buffer[in] = item;
      in = (in + 1) % BUFFER SIZE;
 }
```

Bounded Buffer – Remove() Method

```
while (true) {
     while (in == out)
          ; // do nothing -- nothing to consume

     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
    return item;
}
```

Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)

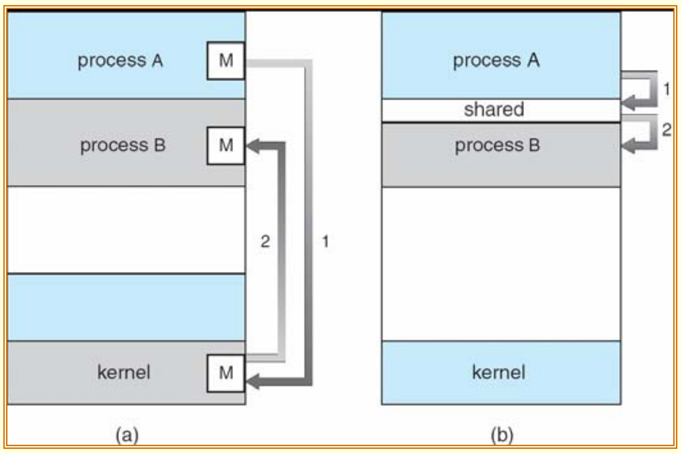
Interprocess Communication (IPC)

- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Communications Models



Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox

Indirect Communication

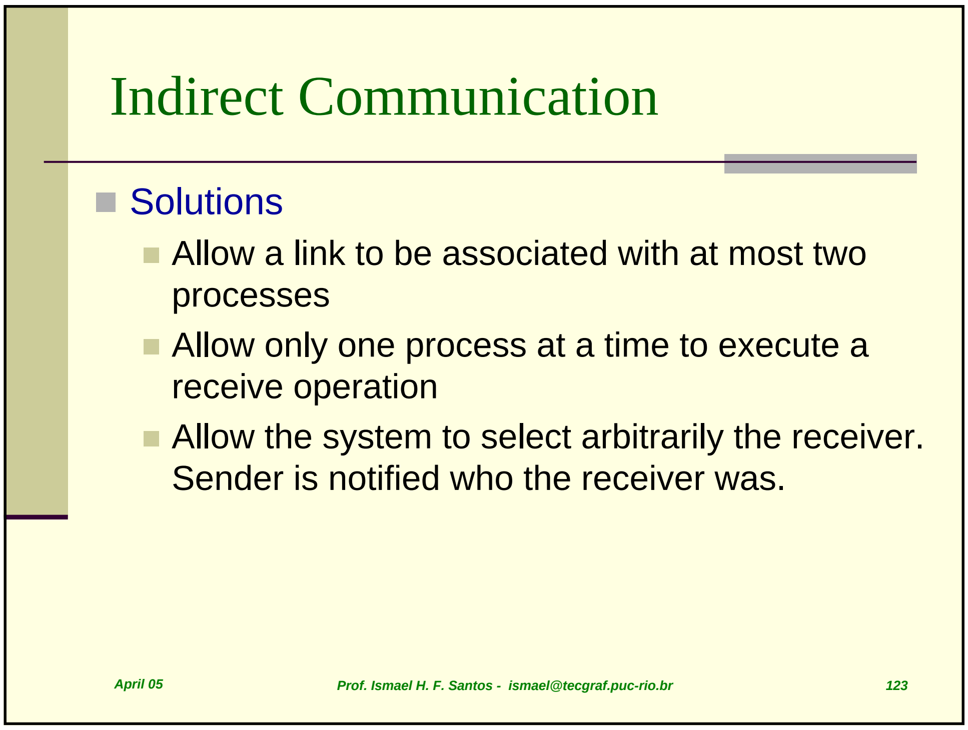
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - **send**(*A, message*) – send a message to mailbox A
  - **receive**(*A, message*) – receive a message from mailbox A

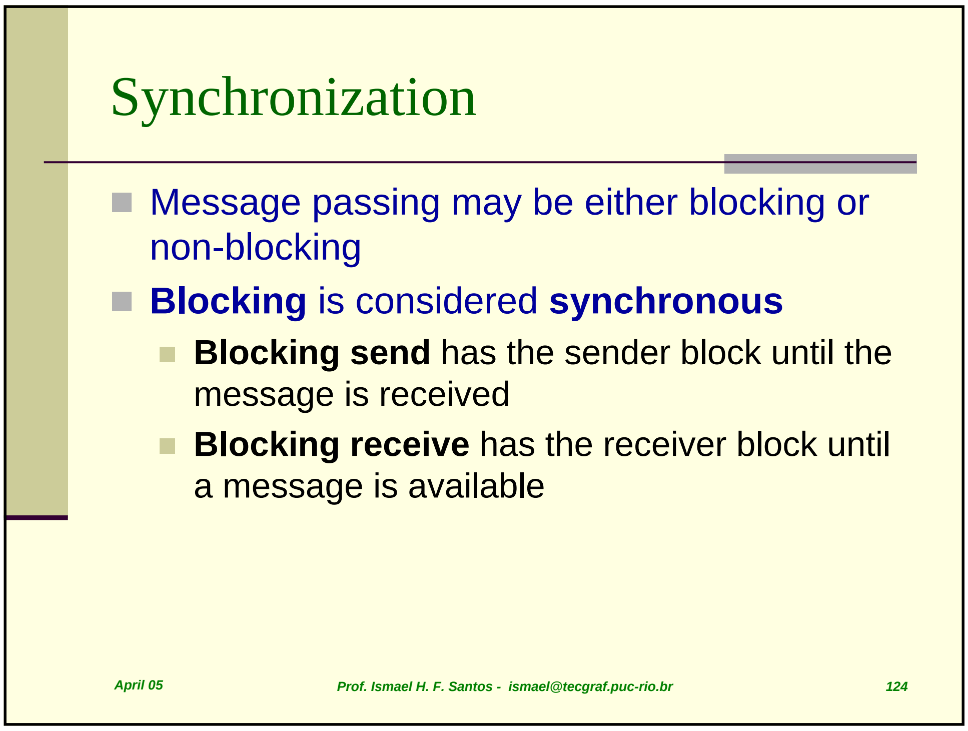
Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?



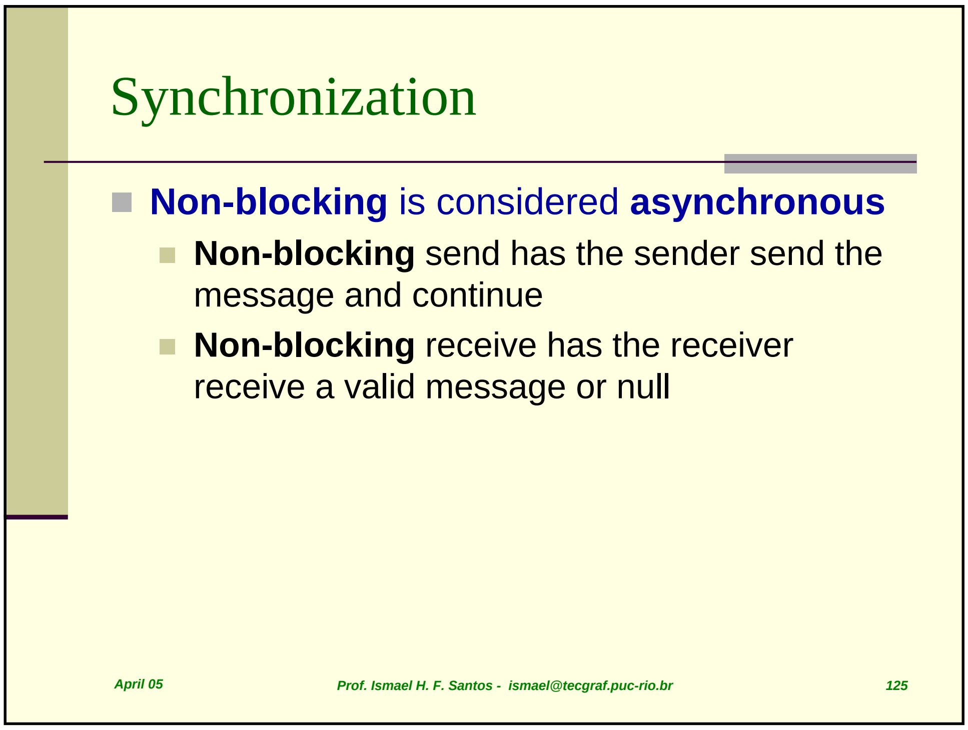
Indirect Communication

- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



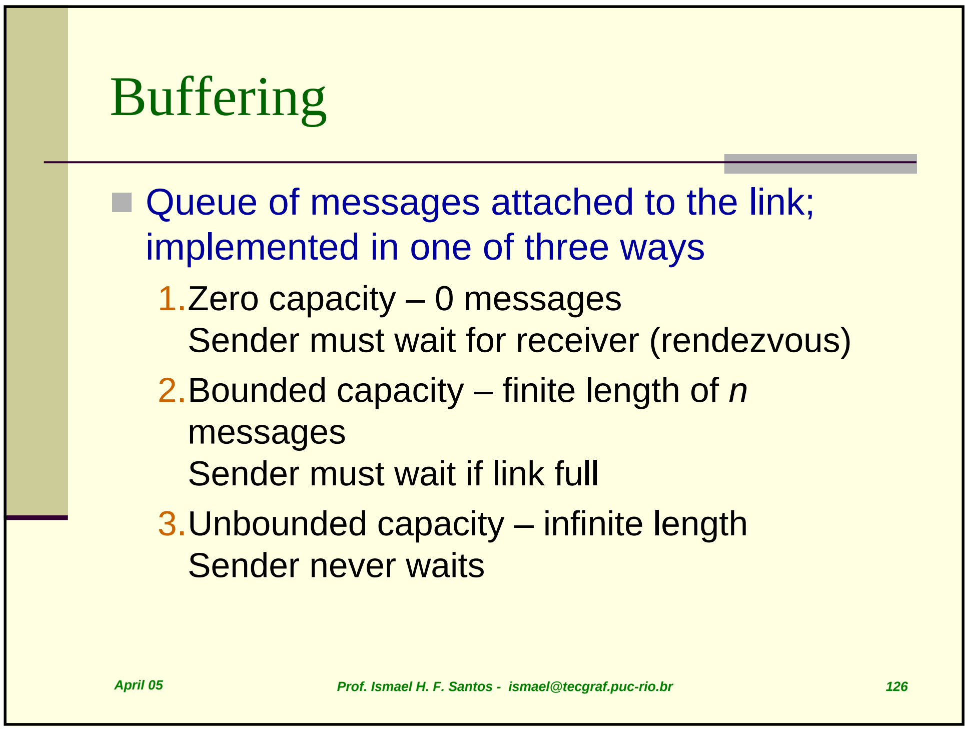
Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available



Synchronization

- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

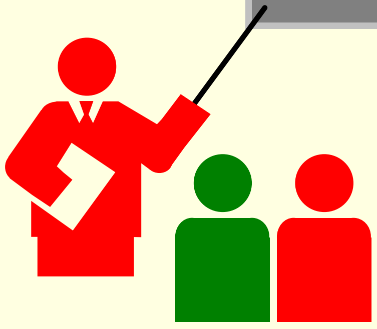


Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

SOP – CO009

Client Server Communications



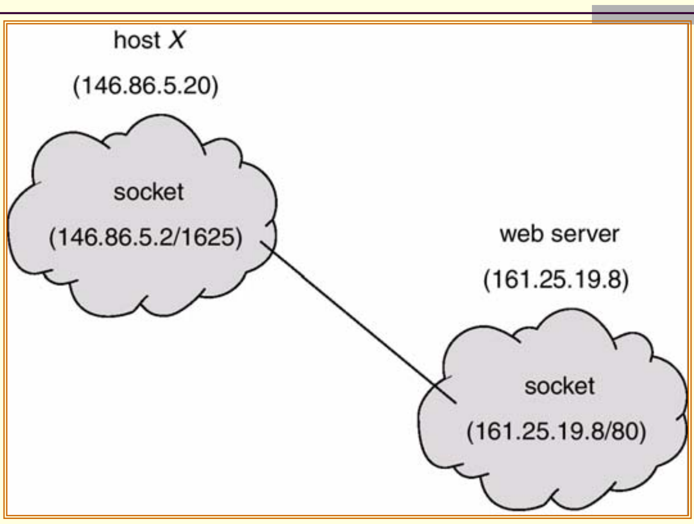
Client-Server Communication

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

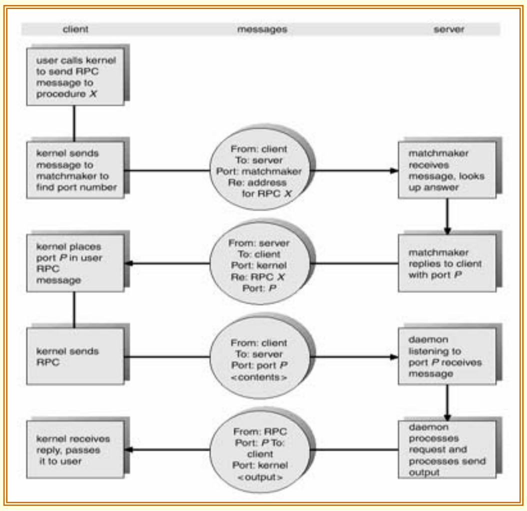
Socket Communication



Remote Procedure Calls

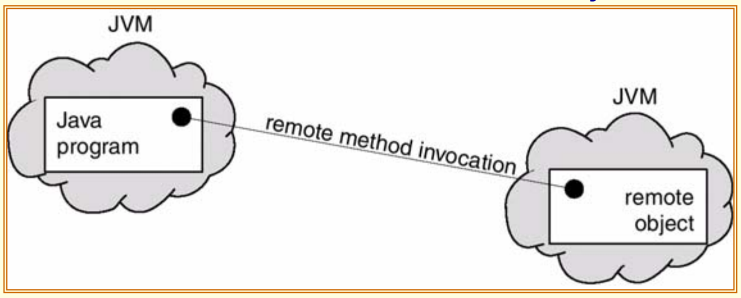
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server.

Execution of RPC

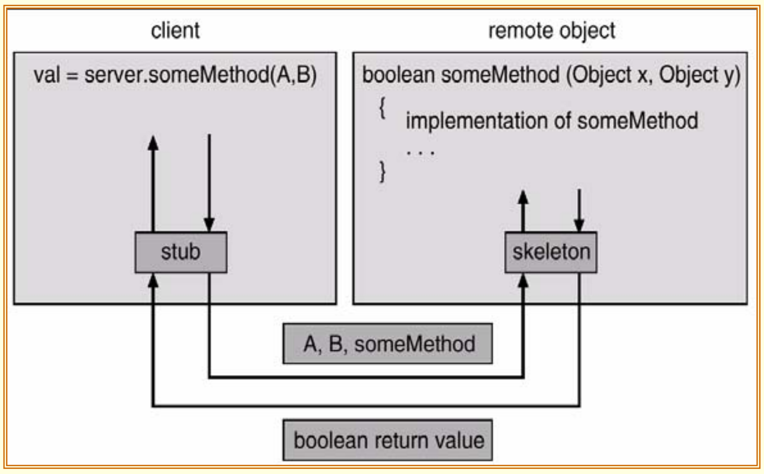

client
messages
server
user calls kernel to send RPC message to procedure X
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC X
matchmaker receives message, looks up answer
kernel places port P in user RPC message
From: server To: client Port: kernel Re: RPC X Port: P
matchmaker replies to client with port P
kernel sends RPC
From: client To: server Port: port P <contents>
daemon listening to port P receives message
kernel receives reply, passes it to user
From: RPC Port: P To: client Port: kernel <output>
daemon processes request and processes send output

Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



Marshalling Parameters


client
val = server.someMethod(A,B)
stub
remote object
boolean someMethod (Object x, Object y)
{
implementation of someMethod
. . .
}
skeleton
A, B, someMethod
boolean return value