Modulo I - Threads
Programação Concorrente

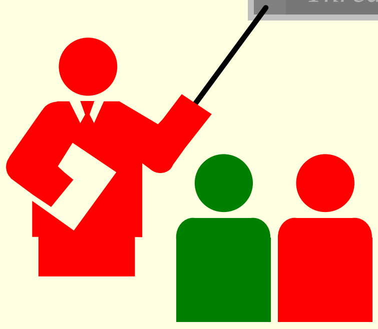
Prof. Ismael H F Santos

Ementa

- Programação Concorrente - Threads
  - Multithreading Models
  - Threading Issues
  - Threads Implementations
    - Pthreads
    - Windows XP Threads
    - Solaris Threads
    - Linux Threads
  - Java Threads
    - Thread States
    - Thread Schedulling
    - Thread Synchronization
    - Timer e TimerTask

SCD – CO023

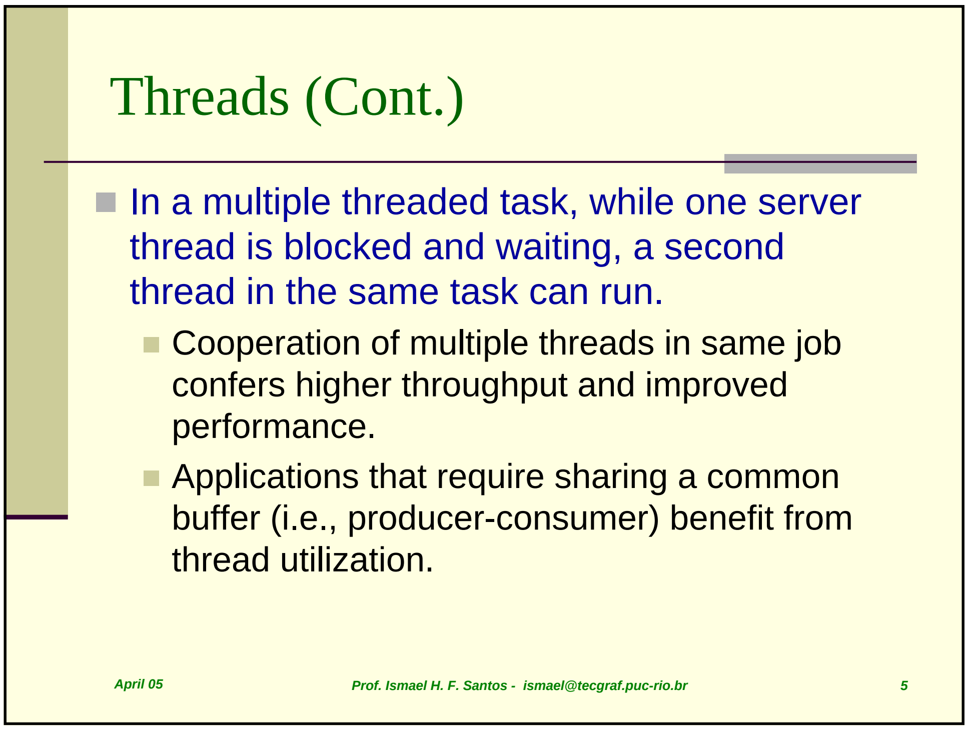
Modelos de Threads



Threads ou Ligth Weighted Process (LWP)

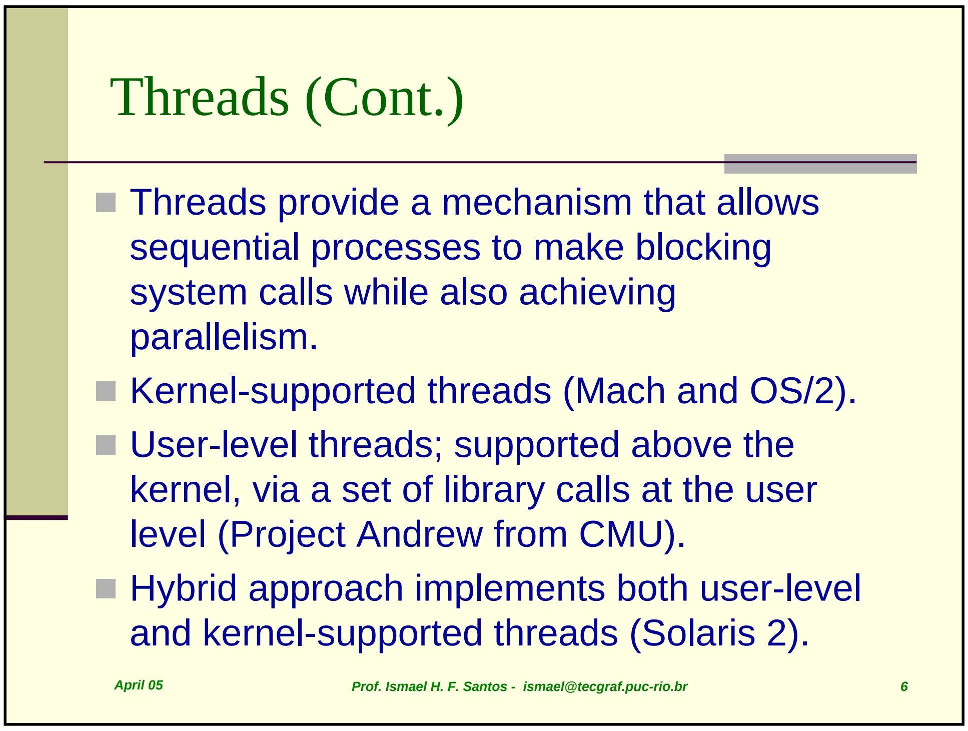
- A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
  - program counter
  - register set
  - stack space
- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources

  collectively know as a *task*.
- A traditional or *heavyweight* process is equal to a task with one thread



Threads (Cont.)

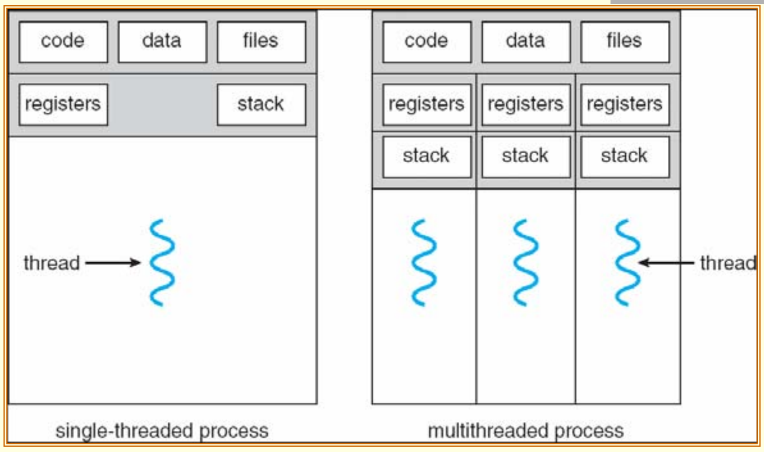
- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
  - Cooperation of multiple threads in same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.



Threads (Cont.)

- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.
- Kernel-supported threads (Mach and OS/2).
- User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

Single and Multithreaded Processes

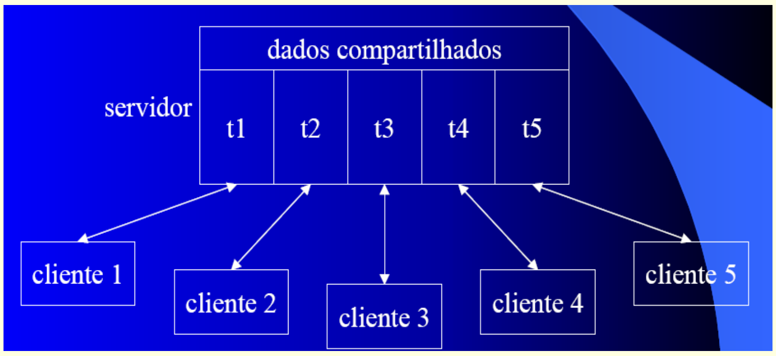


Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

Concorrência em arquitetura cliente cliente-servidor: servidor

- Atendimento simultâneo a vários clientes



Concorrência em arquitetura cliente cliente-servidor: cliente

- Melhor estrutura da aplicação:
  - resposta a eventos de interface e de rede
- Melhor aproveitamento do tempo:
  - disparo de diversas solicitações simultâneas
  - tratamento local de dados enquanto espera resultado de solicitação

User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

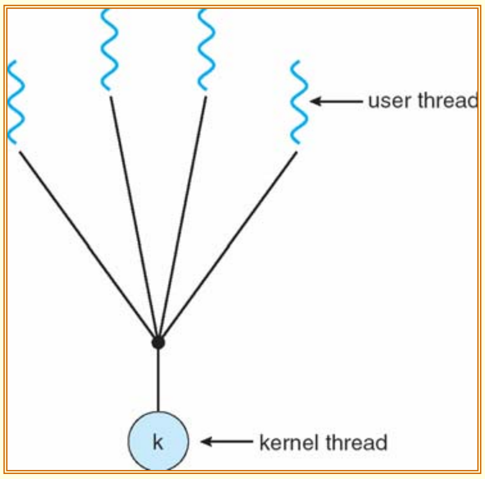
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

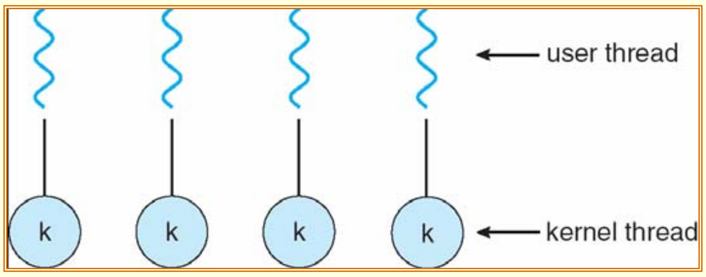
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

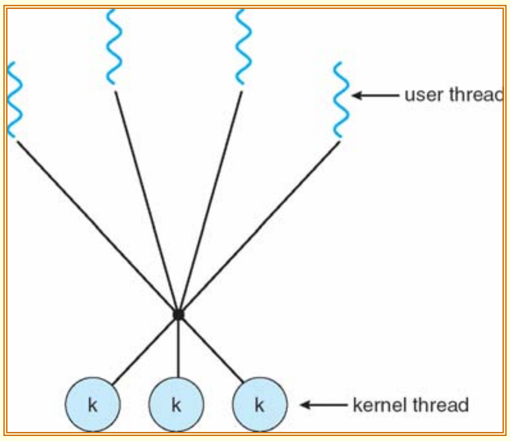
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

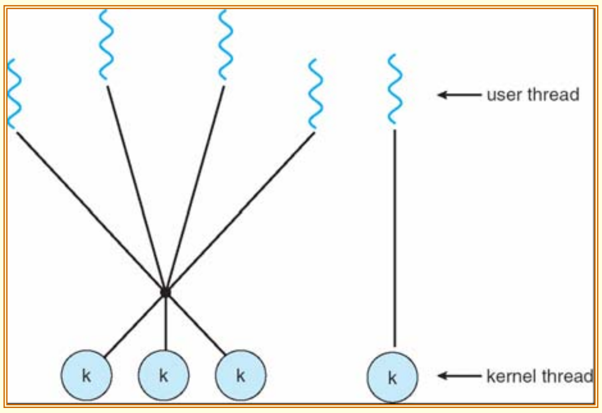
Many-to-Many Model



Two-level Model

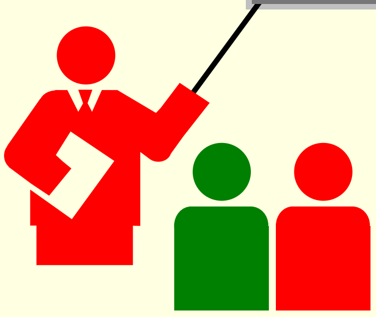
- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

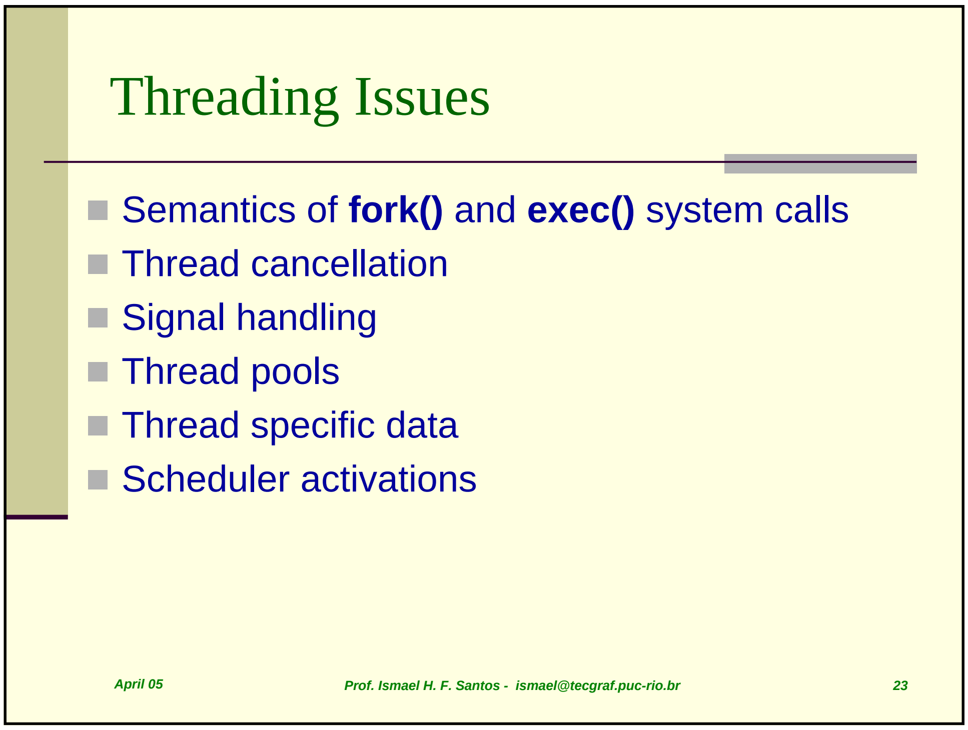
Two-level Model



SCD – CO023

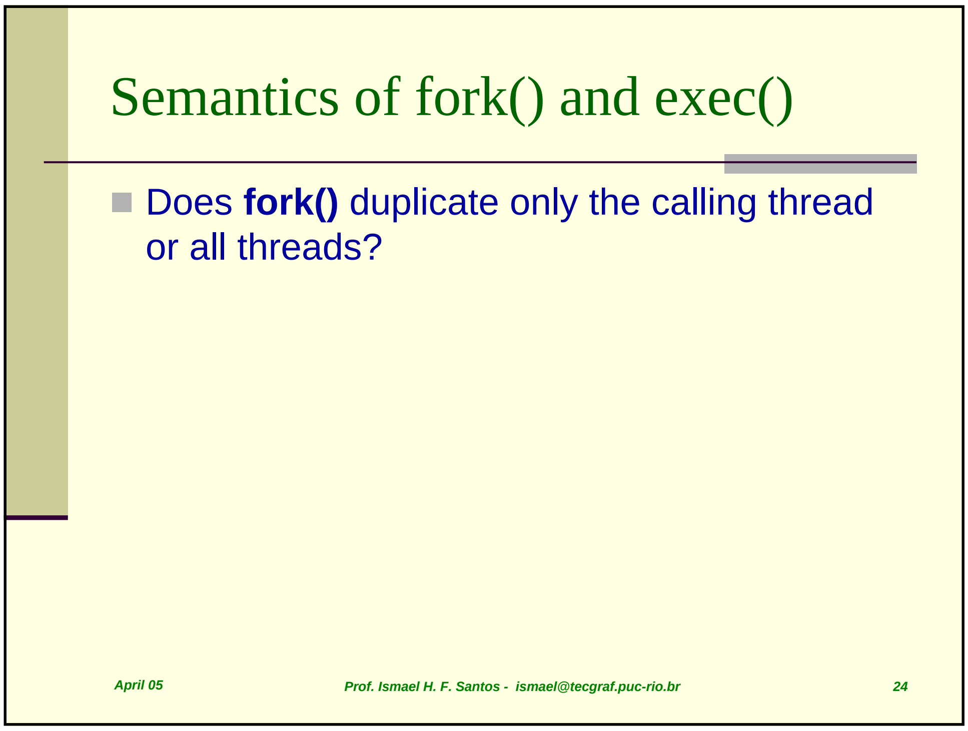
Problemas de Threads





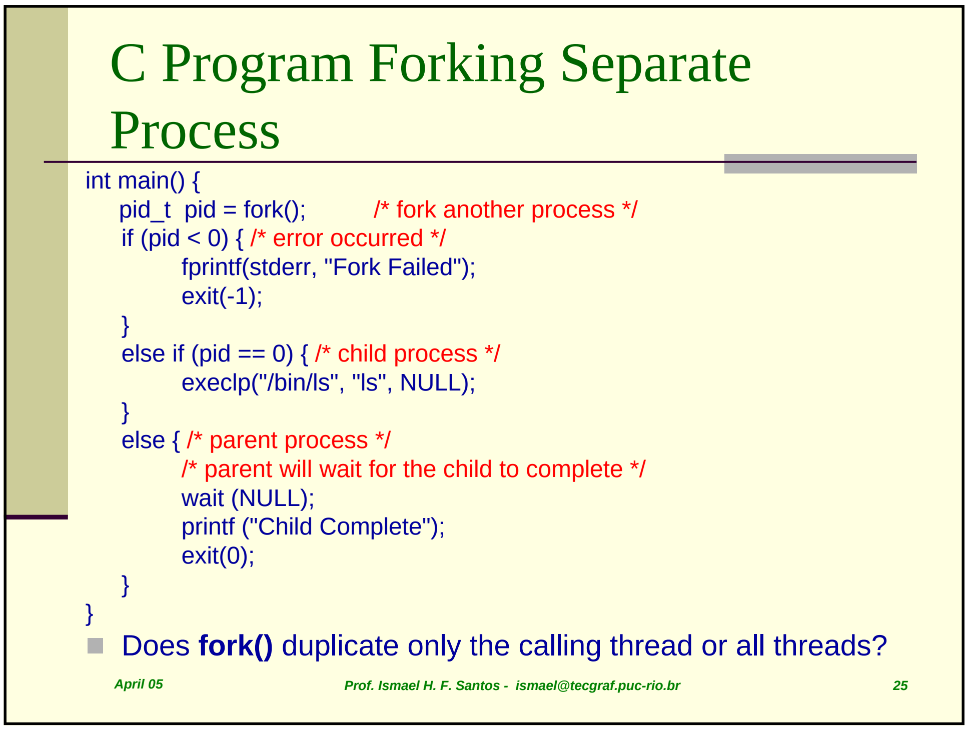
Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations



Semantics of fork() and exec()

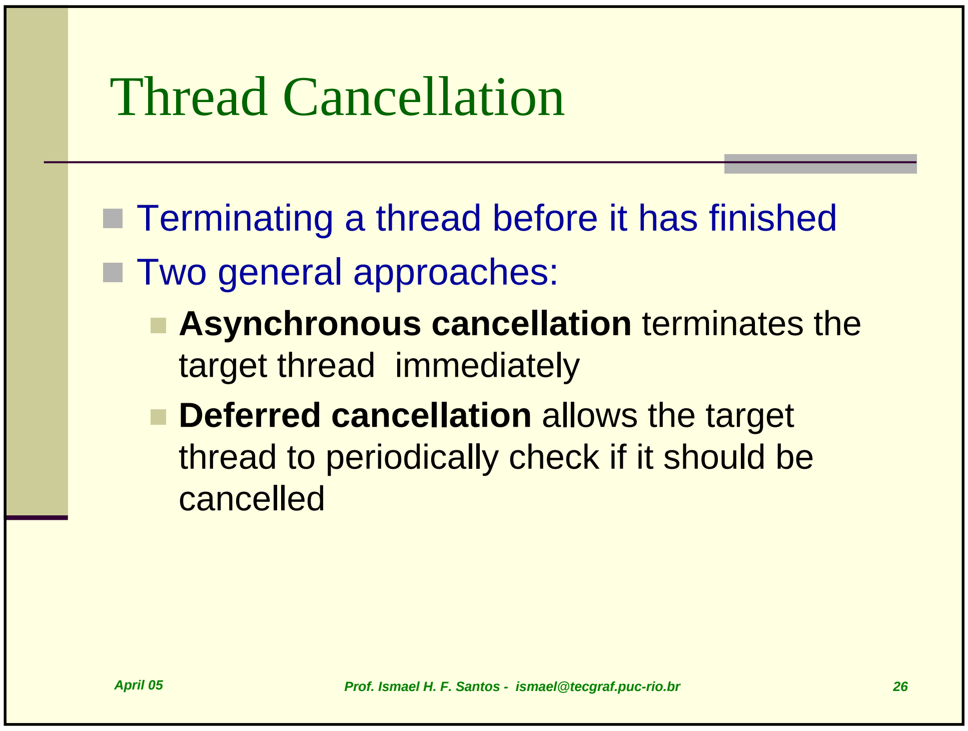
- Does **fork()** duplicate only the calling thread or all threads?



C Program Forking Separate Process

```
int main() {
     pid_t  pid = fork();          /* fork another process */
     if (pid < 0) { /* error occurred */
             fprintf(stderr, "Fork Failed");
             exit(-1);
     }
     else if (pid == 0) { /* child process */
             execlp("/bin/ls", "ls", NULL);
     }
     else { /* parent process */
             /* parent will wait for the child to complete */
             wait (NULL);
             printf ("Child Complete");
             exit(0);
     }
}
```

- Does **fork()** duplicate only the calling thread or all threads?



Thread Cancellation

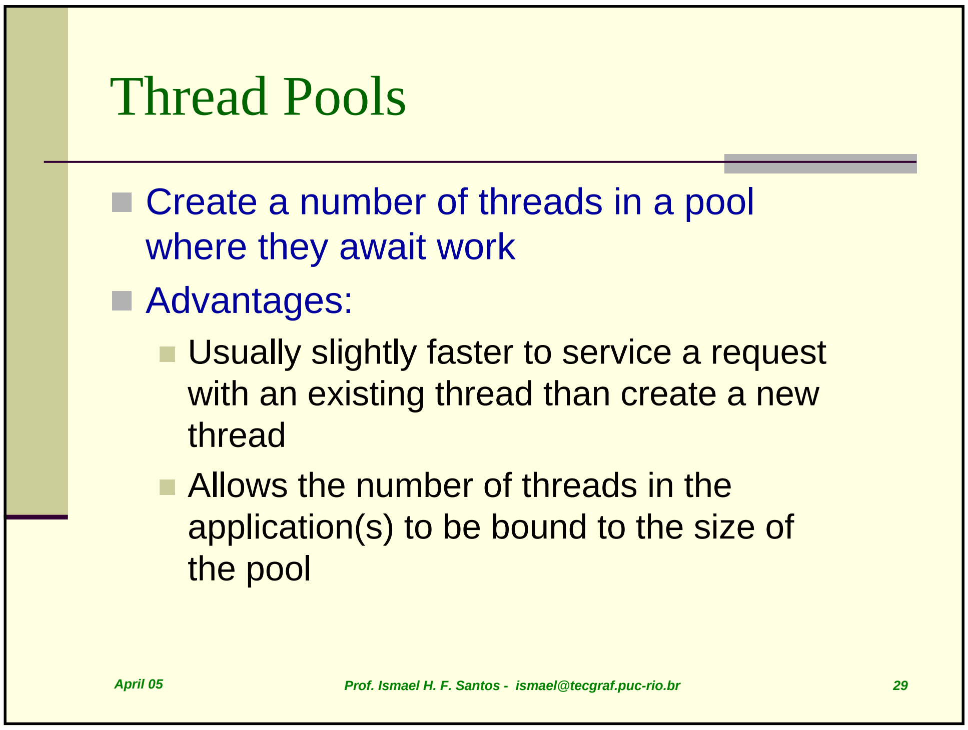
- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled

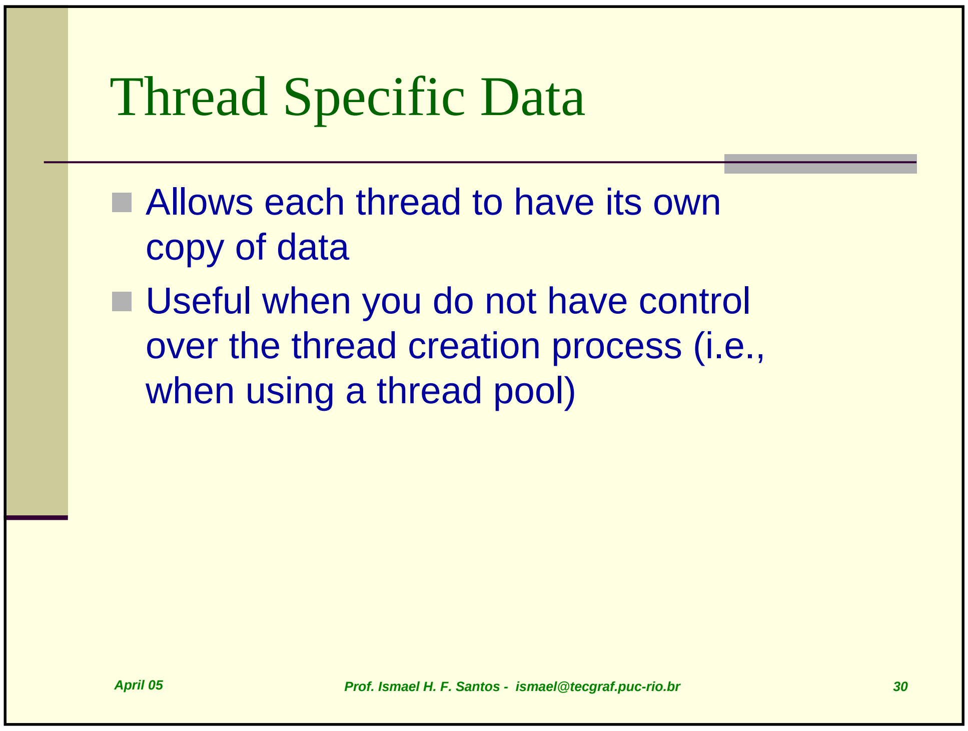
Signal Handling

- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific threa to receive all signals for the process



Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool



Thread Specific Data

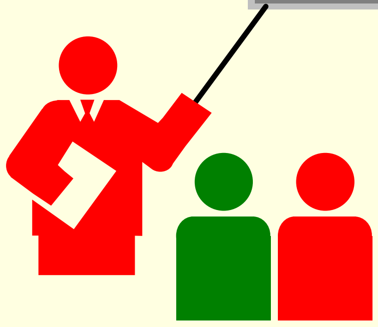
- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

SCD – CO023

Implementações Threads



Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads

```
int sum; /* this data is shared by the thread(s) */

void *runner(void *param) {/* the thread */
   int upper = atoi(param);   int i;
   sum = 0;
   if (upper > 0) {
      for (i = 1; i <= upper; i++)  sum += i;
   }
   pthread_exit(0);
}

main(int argc, char *argv[]) {
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of attributes for the thread */
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* now wait for the thread to exit */
   pthread_join(tid,NULL);
   printf("sum = %d\n",sum);
}
```

Windows XP Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads

Windows XP Threads

- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

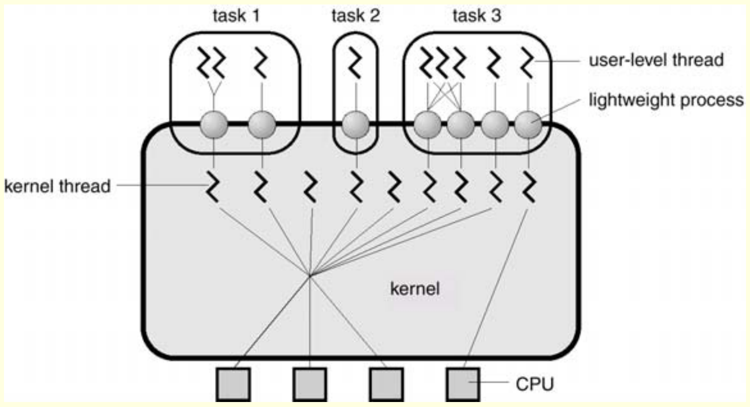
Threads Support in Solaris 2

- Solaris 2 is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.
- LWP – intermediate level between user-level threads and kernel-level threads.

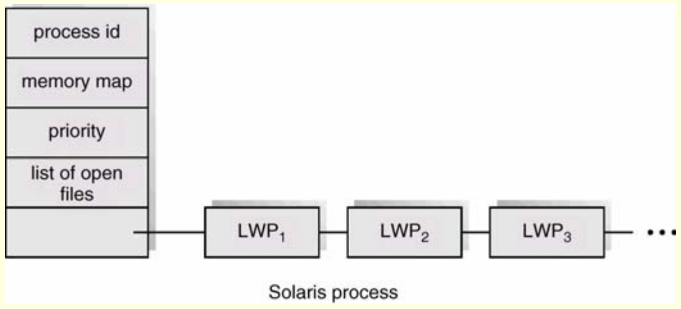
Threads Support in Solaris 2

- Resource needs of thread types:
  - Kernel thread: small data structure and a stack; thread switching does not require changing memory access information – relatively fast.
  - LWP: PCB with register data, accounting and memory information,; switching between LWPs is relatively slow.
  - User-level thread: only need stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level threads.

Solaris 2 Threads



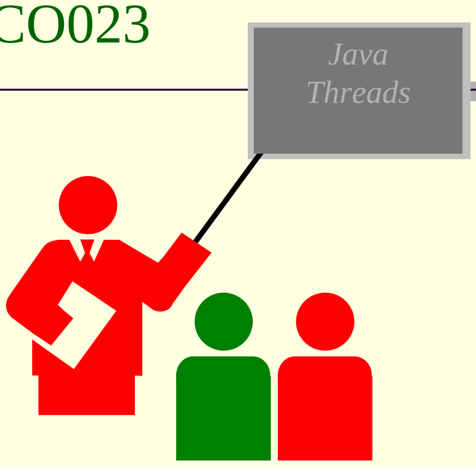
Solaris Process



Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

SCD – CO023



Threads em Java

- Uma classe que estende a classe **Thread** pode ser usada para definição de threads de controle
  - Ao criar um novo objeto dessa classe, cria-se um thread
  - O início de execução ocorre com a chamada ao método **start**
  - O método **run** deve ser redefinido na nova classe para especificar a execução do novo thread

Creating Java Threads

- Java threads may be created by:
  - Extending Thread class

```
public class Worker1 extends Thread {
   public void run() {
     System.out.println(“I am a Worker Thread”); }
}
```

  - Implementing the Runnable interface

```
public interface Runnable {
  public abstract void run();
}
public class Worker2 implements Runnable {
  public void run() {
    System.out.println(“I am a Worker Thread”); }
}
```

Creating the Thread (extends Thread)

```
public class First {
  public static void main(String args[]) {
     Worker runner = new Worker1();
     runner.start();
    System.out.println("I am the main thread");
  }
}
```

Creating the Thread (implements Runnable)

```
public class Second {
  public static void main(String args[]) {
        Runnable runner = new Worker2();
        Thread t = new Thread(runner);
        t.start();
        System.out.println("I am the main thread");
  }
}
```

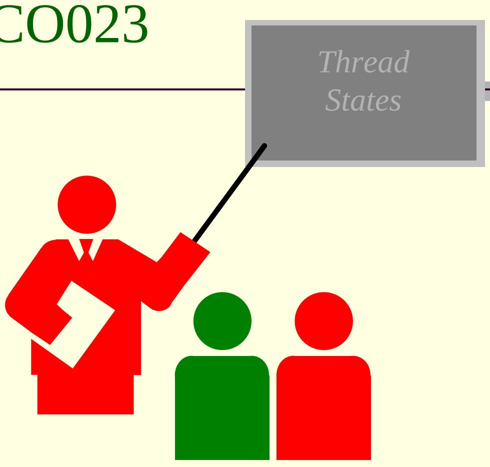
Exemplo: ThreadsDorminhocas

```
public class ThreadsDorminhocas {
   public static void main(String[] args) {
      new ThreadDorminhoca("1");
      new ThreadDorminhoca("2");
      new ThreadDorminhoca("3");
      new ThreadDorminhoca("4");
   }
}
```

ThreadDorminhoca

```
public class ThreadDorminhoca extends Thread {
   int tempo_de_sono;
   public ThreadDorminhoca(String id) {
      super(id);
      tempoSono = (int)(Math.random() * 5000);
      System.out.println("Tempo sono thread "+id+ ":"+ tempoSono+ "ms");
      start();
   }
   public void run() {
      try {
         sleep(tempoSono);
      } catch(InterruptedException exception) {
         System.err.println(exception);
      }
      System.out.println("thread "+getName()+" acordou!");
   }
}
```

SCD – CO023

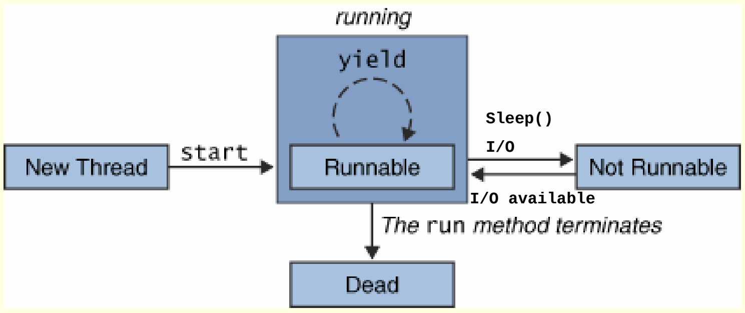


Java Thread States

- A thread can exist in many different states during its "lifetime".
  - New – when the tread is created in memory.
    - Thread t = new Thread ("MyThread");
  - Runnable – when t.start() method is executed JVM:
    - allocate system resources to the thread,
    - schedule the thread for execution,
    - allocate stack space for the thread
    - call the thread's **run()** method
  - The Java scheduler will allocate processor time among the various threads that are in the Runnable state. You can force a thread to relinquish processor time by calling its **yield()** method.

Java Thread States

- When you call **yield()**, the thread will remain in the Runnable state. However, it will allow the Java scheduler to run another thread that might be waiting for its chance.



Java Thread States

- Not Runnable – when a thread is blocking on I/O, or uses the **wait()** method to wait on a condition variable, or call a thread's **sleep()** method.
- Dead – when run() method dies naturally (finish).

- The stop() method in Thread is deprecated and should not be used to stop a thread because it is unsafe.

Joining Threads

```
class JoinableWorker implements Runnable
{
   public void run() {
      System.out.println("Worker working");
   }
}

public class JoinExample {
   public static void main(String[] args) {
      Thread task = new Thread(new JoinableWorker());
      task.start();

      try { task.join(); }
      catch (InterruptedException ie) { }

      System.out.println("Worker done");
   }
}
```

Stopping a Thread

```
public void inicia() {
  if (clockThread == null) {
    clockThread = new Thread(this, "Clock");
    clockThread.start();
  }
}
public void run() {
  Thread myThread = Thread.currentThread();
  while (clockThread == myThread) {
    try { Thread.sleep(1000);
    } catch (InterruptedException e) {
      //JVM doesn't want us to sleep anymore back to work !
    }
  }
}
public void termina() {// terminate thread method
  clockThread = null;
}
```

Thread Cancellation

```
Thread thrd = new Thread (new InterruptibleThread());
thrd.start();
. . .
// now interrupt it
thrd.interrupt();

public class InterruptibleThread implements Runnable {
   public void run() {
      while (true) {
         /**
          * do some work for awhile
          */
         if (Thread.currentThread().isInterrupted()) {
            System.out.println("I'm interrupted!");
            break;
         }
      }
      // clean up and terminate
   }
}
```

Testing Thread State

- JEE 5.0 introduced the Thread.getState() method. One of the following Thread.State values is returned:
  - NEW
  - RUNNABLE
  - BLOCKED
  - WAITING
  - TIMED_WAITING
  - TERMINATED
- The API for the Thread class also includes a method called isAlive(). The isAlive method returns true if the thread has been started and not stopped.

Testing Thread State

- If the isAlive method returns false, you know that the thread either is a New Thread or is Dead. If the isAlive method returns true, you know that the thread is either in a Runnable or Not Runnable state.
- Prior to release 5.0, you couldn't differentiate between a New Thread or a Dead thread; nor could you differentiate between a Runnable thread and a Not Runnable thread.

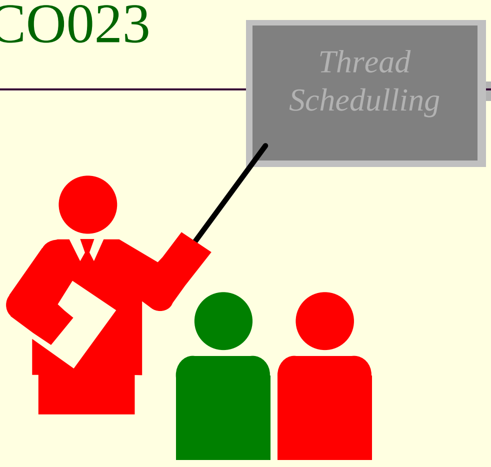
Thread Specific Data

```
class Service {
   private static ThreadLocal errorCode = new ThreadLocal();

   public static void transaction() {
      try {
         // some operation where an error may occur
         ………
      } catch (Exception e) {
         errorCode.set(e);
      }
   }
   // get the error code for this transaction
   public static Object getErrorCode() {
      return errorCode.get();
   }
}
```

```
class Worker implements Runnable {
   private static Service provider;
   public void run() {
      provider.transaction();
      System.out.println(provider.getErrorCode());
   }
}
```

SCD – CO023



Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
- Global Scheduling – How the kernel decides which kernel thread to run next

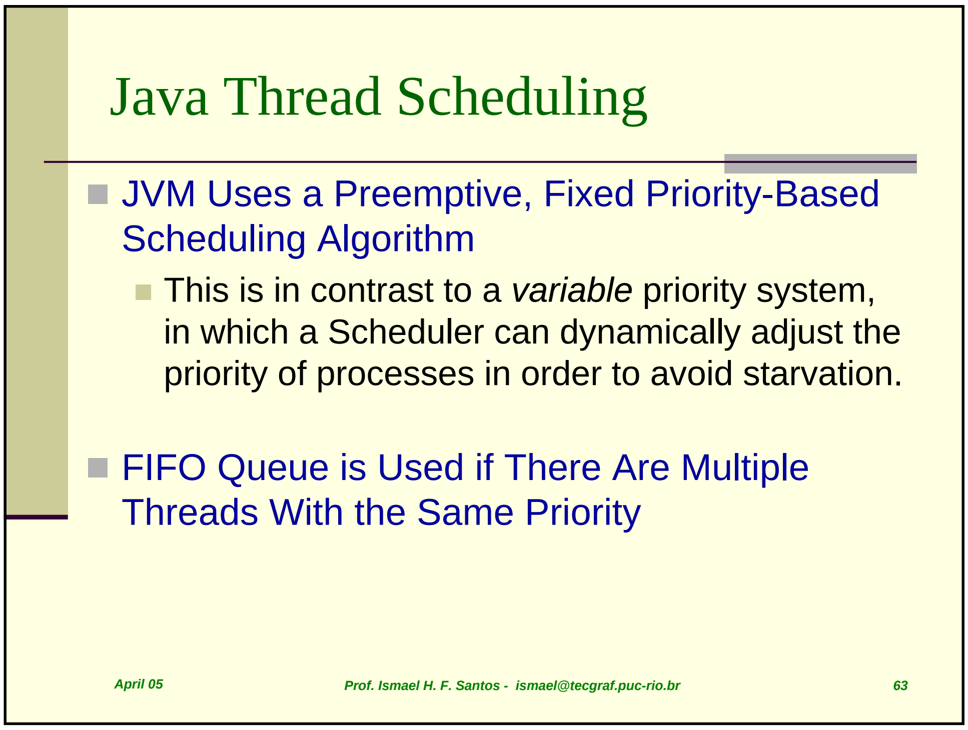
Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[]) {
    int i;
    pthread_t tid[NUM THREADS];
    pthread_attr t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD SCOPE SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED OTHER);
```

Pthread Scheduling API

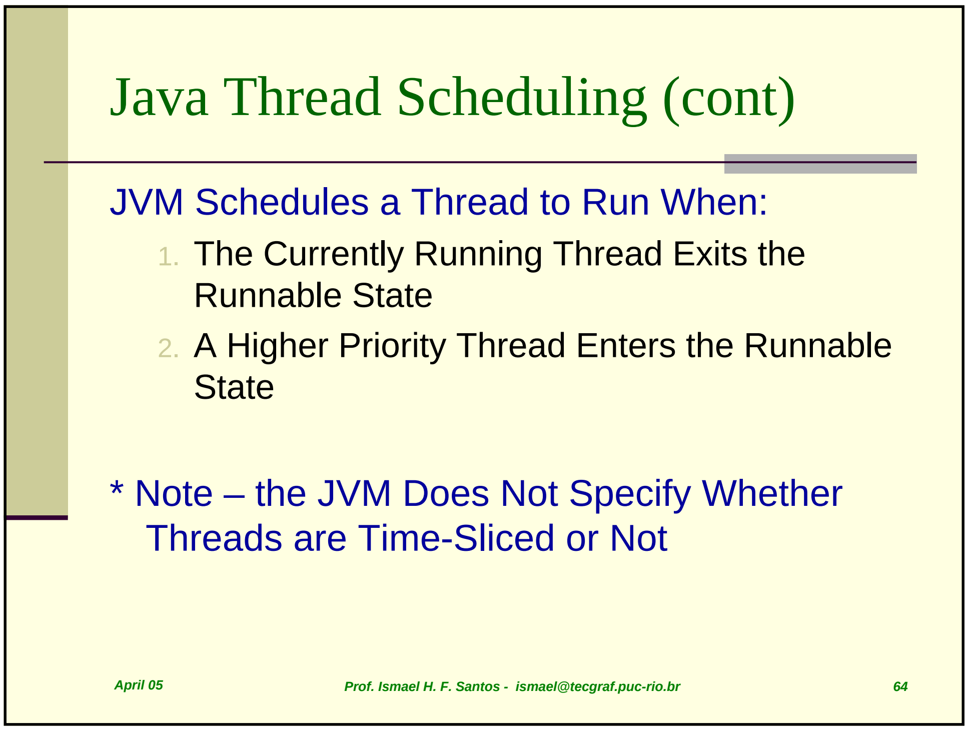
```
   /* create the threads */
   for (i = 0; i < NUM THREADS; i++)
      pthread_create(&tid[i], &attr, runner, NULL);
    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
      pthread_join(tid[i], NULL);
}

 /* Each thread will begin control in this function */
void *runner(void *param) {
    printf("I am a thread\n");
    pthread exit(0);
}
```



Java Thread Scheduling

- JVM Uses a Preemptive, Fixed Priority-Based Scheduling Algorithm
  - This is in contrast to a *variable* priority system, in which a Scheduler can dynamically adjust the priority of processes in order to avoid starvation.
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority



Java Thread Scheduling (cont)

JVM Schedules a Thread to Run When:

1. The Currently Running Thread Exits the Runnable State
2. A Higher Priority Thread Enters the Runnable State

* Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not

Java Thread Scheduling (cont)

- JVM Schedules a Thread to Run when:
  - The Currently Running Thread exits the Runnable State, executing yield() or when its run() method exits.
  - A Higher-Priority Thread becomes Runnable.
  - On systems that support time-slicing, the thread's time allotment has expired.

* **Rule of Thumb:** At any given time, the highest-priority thread is running. However, this is not guaranteed. The thread scheduler may choose to run a lower-priority thread to avoid starvation. For this reason, use thread priority only to affect scheduling policy for efficiency purposes; do *not* rely on it for algorithm correctness !

Time-Slicing

- Since the JVM Doesn't Ensure Time-Slicing, the yield() method may Be Used:

```
while (true) {
    // perform CPU-intensive task
      . . .
    Thread.yield();
  }
```

- This Yields Control to Another Thread of Equal Priority

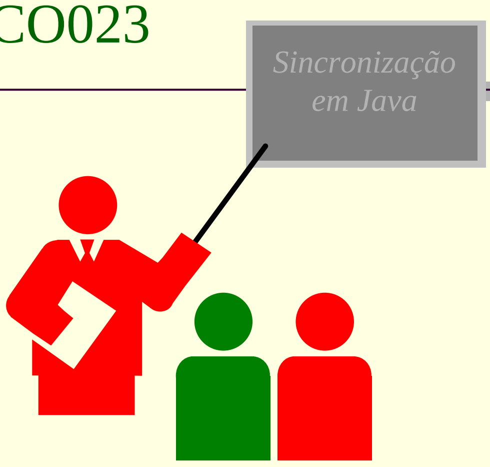
Thread Priorities

Priority	**Comment**
Thread.MIN_PRIORITY	Minimum Thread Priority
Thread.MAX_PRIORITY	Maximum Thread Priority
Thread.NORM_PRIORITY	Default Thread Priority

Priorities May Be Set Using setPriority() method:

```
setPriority(Thread.NORM_PRIORITY + 2);
```

SCD – CO023



Bounded-Buffer – Shared Memory Solution

```
import java.util.*;
public class BoundedBuffer implements Buffer {
        private static final int BUFFER SIZE = 5;
        private int count; // number of items in the buffer
        private int in; // points to the next free position
        private int out; // points to the next full position
        private Object[] buffer;
        public BoundedBuffer()  {
                // buffer is initially empty
                count = 0;  in = 0; out = 0;
                buffer = new Object[BUFFER SIZE];
         }
        // producers calls this method
        public void insert(Object item) {    // next slide  }

        // consumers calls this method
        public Object remove() {             // next slide   }
}
```

Bounded-Buffer – insert() / remove()

```
public void insert(Object item) {
      while (count == BUFFER SIZE)
               ; // do nothing -- no free buffers
                 // add an item to the buffer
      ++count;
      buffer[in] = item;
      in = (in + 1) % BUFFER SIZE;
}

public Object remove() {
      Object item;
      while (count == 0)
               ; // do nothing -- nothing to consume
      // remove an item from the buffer
      --count;
      item = buffer[out];
      out = (out + 1) % BUFFER SIZE;
      return item;
}
```

Producer Consumer Problem

```
public class Server {
   public Server() {
     MessageQueue mailBox = new MessageQueue();

      Producer   pThread = new Producer(mailBox);
      Consumer cThread = new Consumer(mailBox);

      pThread.start();
      cThread.start();
  }
  public static void main(String args[])   {
      Server server = new Server();
  }
}
```

Producer Thread

```
public class Producer extends Thread {
   public Producer(MessageQueue m) {
            mbox = m;
   }

   public void run() {
      while (true) {
        //produce an item & enter it into the buffer
        Date message = new Date();
        System.out.println("Producer produced " + message);
        mbox.send(message);
      }
  }
  private  MessageQueue mbox;
}
```

Consumer Thread

```
public class Consumer extends Thread {
   public Consumer(MessageQueue m) {
     mbox = m;
   }
   public void run() {
     while (true) {
     Date message = (Date)mbox.receive();
      if (message != null)
           // consume the message
           System.out.println("Consumer consumed " + message);
     }
  }
  private MessageQueue mbox;
}
```

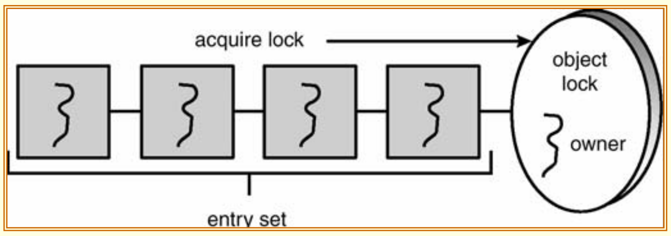
Java Synchronization

- Bounded Buffer solution using synchronized, wait(), notify() statements
- Multiple Notifications
- Block Synchronization
- Java Semaphores
- Java Monitors

synchronized Statement

- Every object has a lock associated with it.
- Calling a synchronized method requires “owning” the lock.
- If a calling thread does not own the lock (another thread already owns it), the calling thread is placed in the wait set for the object’s lock.
- The lock is released when a thread exits the synchronized method.

Entry Set



synchronized insert() Method

```
public synchronized void insert(Object item) {
   while (count == BUFFER SIZE)
                  Thread.yield();
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
}
```

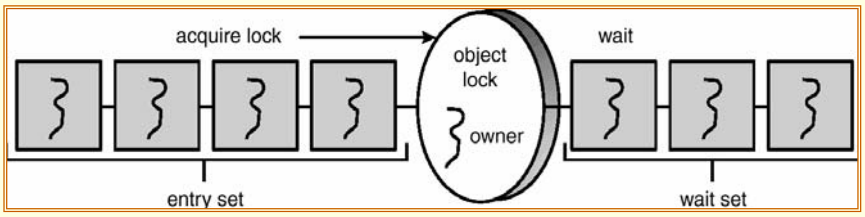
synchronized remove() Method

```
public synchronized Object remove() {
  Object  item;
  while (count == 0)
        Thread.yield();
  --count;
  item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  return item;
}
```

The wait() Method

- When a thread calls wait(), the following occurs:
  - the thread releases the object lock.
  - thread state is set to blocked.
  - thread is placed in the wait set.

Entry and Wait Sets



The notify() Method

When a thread calls notify(), the following occurs:

1. selects an arbitrary thread *T* from the wait set
2. moves *T* to the entry set
3. sets *T* to Runnable

T can now compete for the object's lock again

insert() with wait/notify Methods

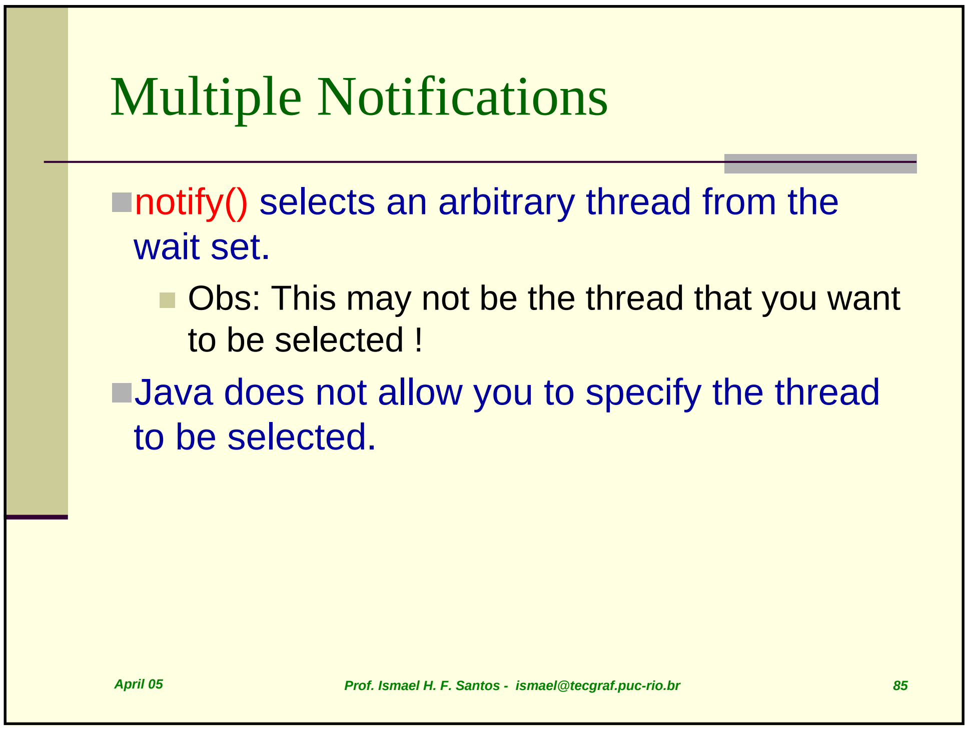
```
public synchronized void insert(Object item) {
  while (count == BUFFER_SIZE)
        try {
                wait();
        } catch (InterruptedException e) { }
  }
  ++count;
  buffer[in] = item;
  in = (in + 1) % BUFFER_SIZE;
  notify();
}
```

remove() with wait/notify Methods

```
public synchronized Object remove() {
  Object  item;
  while (count == 0)
        try {
                wait();
        } catch (InterruptedException e) { }
  --count;
  item = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  notify();
  return item;
}
```

Complete Bounded Buffer using Java Synchronization

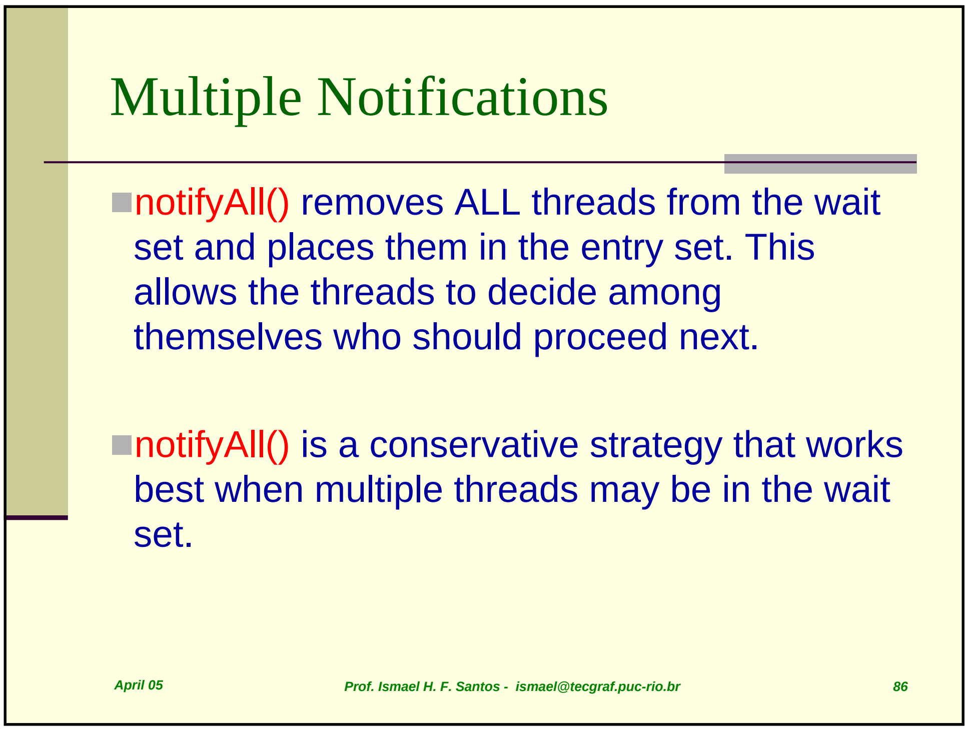
```
public class BoundedBuffer implements Buffer {
   private static final int BUFFER SIZE = 5;
   private int count, in, out;
   private Object[] buffer;
   public BoundedBuffer() { // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;
        buffer = new Object[BUFFER SIZE];
   }
   public synchronized void insert(Object item) { // See previous slides
   }
   public synchronized Object remove() { // See previous slides
   }
}
```



Multiple Notifications

- notify() selects an arbitrary thread from the wait set.
  - Obs: This may not be the thread that you want to be selected !
- Java does not allow you to specify the thread to be selected.

April 05 Prof. Ismael H. F. Santos - ismael@tecgraf.puc-rio.br 85



Multiple Notifications

- notifyAll() removes ALL threads from the wait set and places them in the entry set. This allows the threads to decide among themselves who should proceed next.
- notifyAll() is a conservative strategy that works best when multiple threads may be in the wait set.

April 05 Prof. Ismael H. F. Santos - ismael@tecgraf.puc-rio.br 86

Block Synchronization

- Scope of lock is time between lock acquire and release
- Blocks of code – rather than entire methods – may be declared as synchronized.
- This yields a lock scope that is typically smaller than a synchronized method.

Block Synchronization (cont)

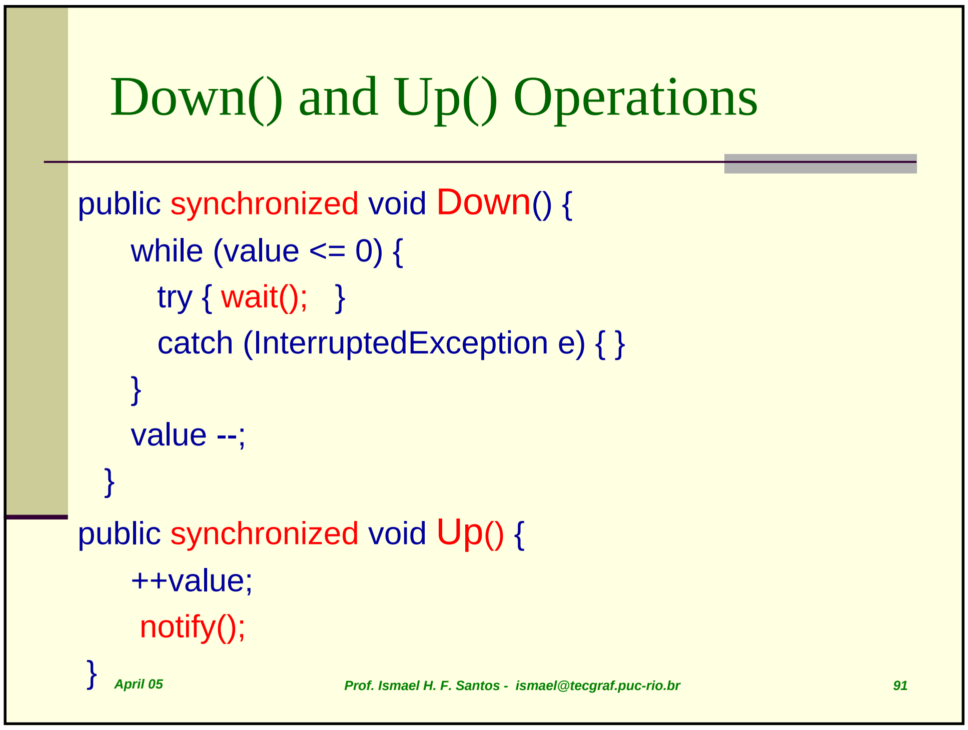
```
Object mutexLock = new Object();
. . .
public void someMethod() {
  // non-critical section
  synchronized(mutexLock) { // protocolo entrada
        // critical  section
  }                                  // protocolo saída
  // non-critical section
}
```

Java Semaphores

- Java does not provide a semaphore, but a basic semaphore can be constructed using Java synchronization mechanism.

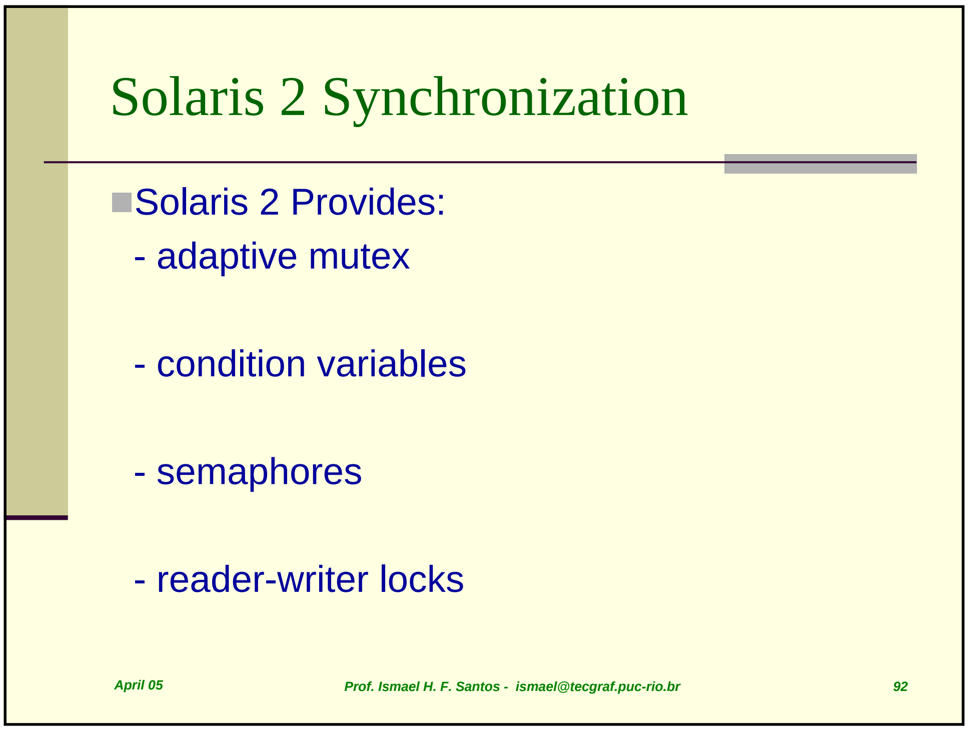
Java Semaphore Class

```
public class Semaphore {
   public Semaphore() {
         value = 0;
   }
   public Semaphore(int v) {
         value = v;
   }
   public synchronized void Down() { /* see next slide */ }
   public synchronized void Up() { /* see next slide */ }
   private int value;
}
```



Down() and Up() Operations

```
public synchronized void Down() {
      while (value <= 0) {
         try { wait();   }
         catch (InterruptedException e) { }
      }
      value --;
   }
public synchronized void Up() {
      ++value;
       notify();
 }
```



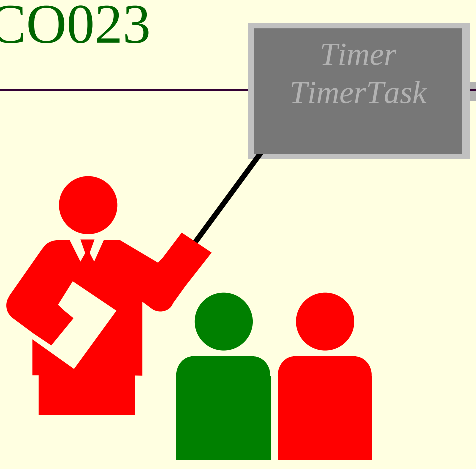
Solaris 2 Synchronization

- Solaris 2 Provides:
  - adaptive mutex
  - condition variables
  - semaphores
  - reader-writer locks

Windows NT Synchronization

- Windows NT Provides:
  - mutex
  - critical sections
  - semaphores
  - event objects

SCD – CO023



Class Timer and TimerTask

- Timer
  - Timer is a facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.
- TimerTask
  - Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially. Timer tasks should complete quickly. If a timer task takes excessive time to complete, it "hogs" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch up" and execute in rapid succession when (and if) the offending task finally completes.

Class Timer and TimerTask

```
import java.util.Timer;
import java.util.TimerTask;
// Schedule a task to execute once 5 seconds have passed.
public class Reminder {
  Timer timer;
  public Reminder(int secs) {
    t = new Timer(); t.schedule(new RemindTask(),secs*1000);
  }
  class RemindTask extends TimerTask {
    public void run() {
      System.out.format("Time's up!%n");
      t.cancel();    //Terminate the timer thread }
  }
  public static void main(String args[]) {
   new Reminder(5); System.out.format("Task scheduled.%n");}
}
```

Another way of schedulling

- Another way of scheduling a task is to specify the time when the task should execute.
  - For example, the following code schedules a task for execution at 11:01 P.M.

```
Calendar calendar = Calendar.getInstance();
calendar.set(Calendar.HOUR_OF_DAY, 23);
calendar.set(Calendar.MINUTE, 1);
calendar.set(Calendar.SECOND, 0);
Date time = calendar.getTime();

timer = new Timer();
timer.schedule(new RemindTask(), time);
```

Stopping Timer Threads

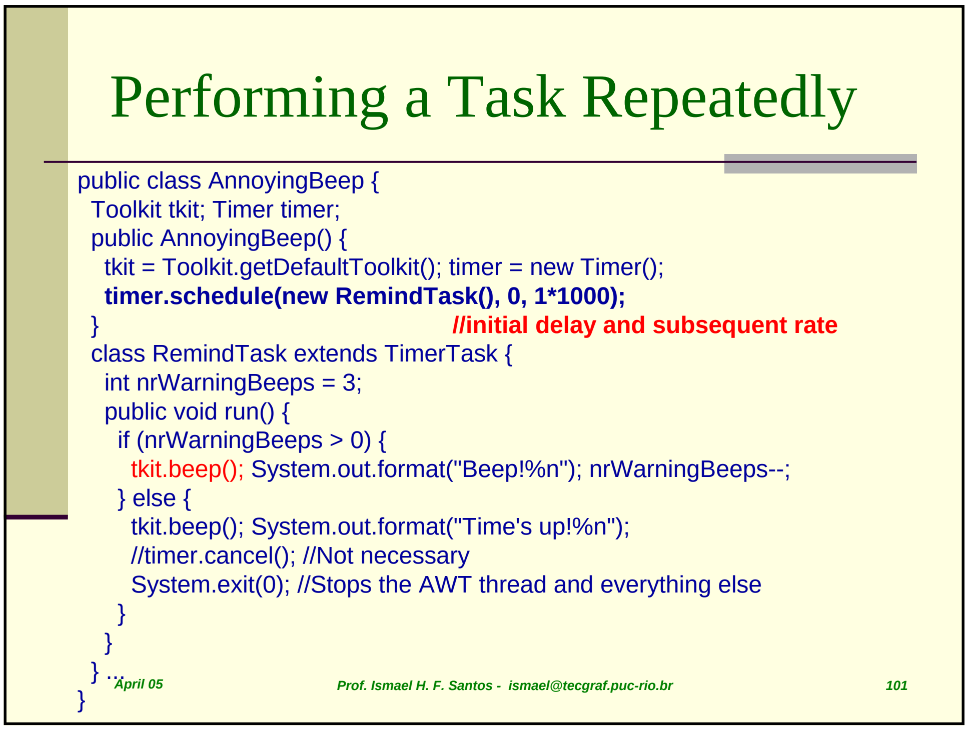
- By default, a program keeps running as long as its timer threads are running. You can terminate a timer thread in four ways:
  - Invoke cancel on the timer. You can do this from anywhere in the program, such as from a timer task's run method.
  - Make the timer's thread a "daemon" by creating the timer like this: new Timer(true). If the only threads left in the program are daemon threads, the program exits.
  - After all the timer's scheduled tasks have finished executing, remove all references to the Timer object. Eventually, the timer's thread will terminate.
  - Invoke the System.exit method, which makes the entire program (and all its threads) exit.

Stopping Timer Threads

- The Reminder example uses the first scheme, invoking the cancel method from the timer task's run method.
- Sometimes, timer threads aren't the only threads that can prevent a program from exiting when expected. For example, if you use the AWT at all (even if only to make beeps), it automatically creates a nondaemon thread that keeps the program alive.

Stopping Timer Threads (cont.)

```
public class ReminderBeep {
  ...
  public ReminderBeep(int secs) {
    toolkit = Toolkit.getDefaultToolkit(); // AWT method !!
    t = new Timer(); t.schedule(new RemindTask(),secs*1000);
  }
  class RemindTask extends TimerTask {
    public void run() {
      System.out.format("Time's up!%n"); toolkit.beep();
      //timer.cancel(); //Not necessary !
      System.exit(0); //Stops AWT thread and everything else
    }
  }
...
}
```



Performing a Task Repeatedly

```
public class AnnoyingBeep {
  Toolkit tkit; Timer timer;
  public AnnoyingBeep() {
    tkit = Toolkit.getDefaultToolkit(); timer = new Timer();
    timer.schedule(new RemindTask(), 0, 1*1000);
  }                                  //initial delay and subsequent rate
  class RemindTask extends TimerTask {
    int nrWarningBeeps = 3;
    public void run() {
      if (nrWarningBeeps > 0) {
        tkit.beep(); System.out.format("Beep!%n"); nrWarningBeeps--;
      } else {
        tkit.beep(); System.out.format("Time's up!%n");
        //timer.cancel(); //Not necessary
        System.exit(0); //Stops the AWT thread and everything else
      }
    }
  } …
}
```