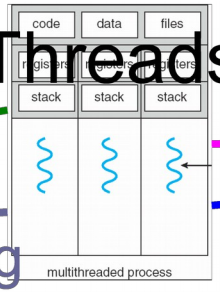
Module 3 - Threads

Reading: Chapter 4

Objective:

- **Understand the concept of the thread and its relationship to the process.**
- **Study and understand the different methods of multithreading used by operating systems to manage and use threads.**
- **Review issues and challenges that threads present.**
- **Review thread library examples and thread implementation examples**

Threads



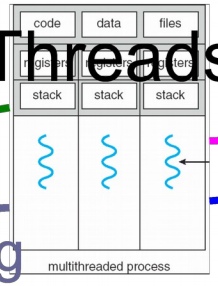
Overview

Mulithreading Models

Threading Issues

Threading Examples

Threads



Overview

Thread of execution

Single thread verus multithread

Motivation

Mulithreading Models

Threading Issues

Threading Examples

Process characteristics

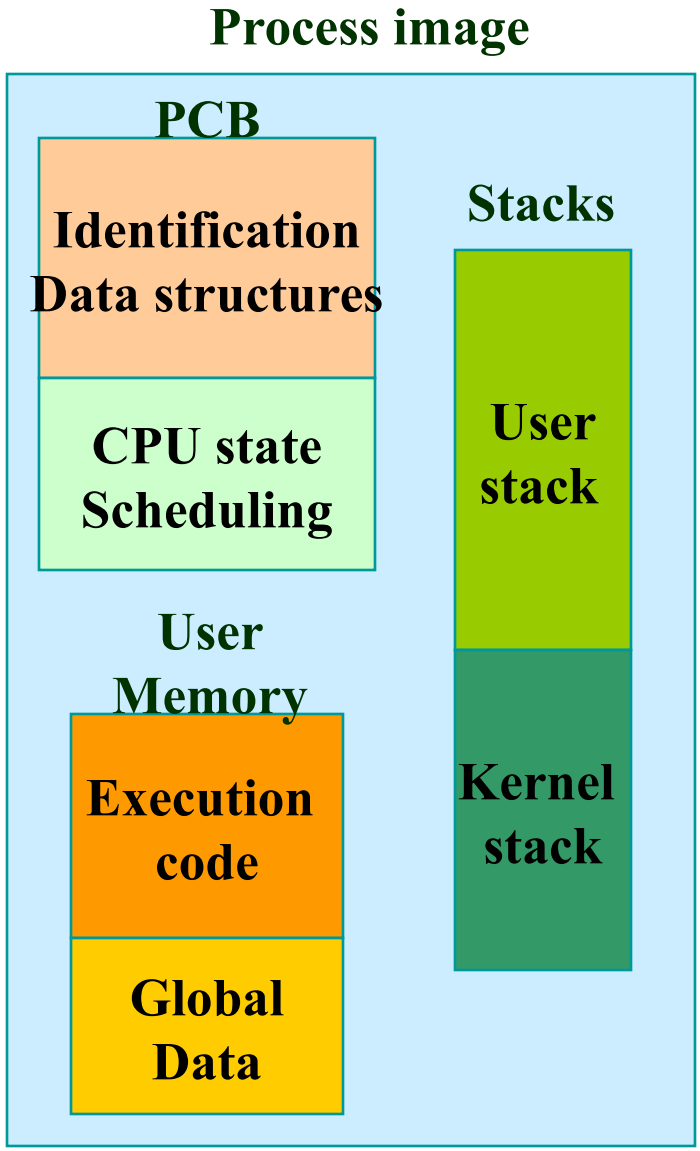
- Resource ownership – a process owns:
  - A virtual address space that contains the image of the process
  - Other resources (files, I/O devices, etc.)
- Execution (scheduling) – a process executes along a path in one or more programs.
  - The execution is interleaved with the execution of other processes.
  - The process has an execution state and priority used for scheduling

Process Characteristics

- These 2 characteristics are most often treated independently by OS's
- Execution is normally designated as execution thread
- Resource ownership is normally designated as process or task

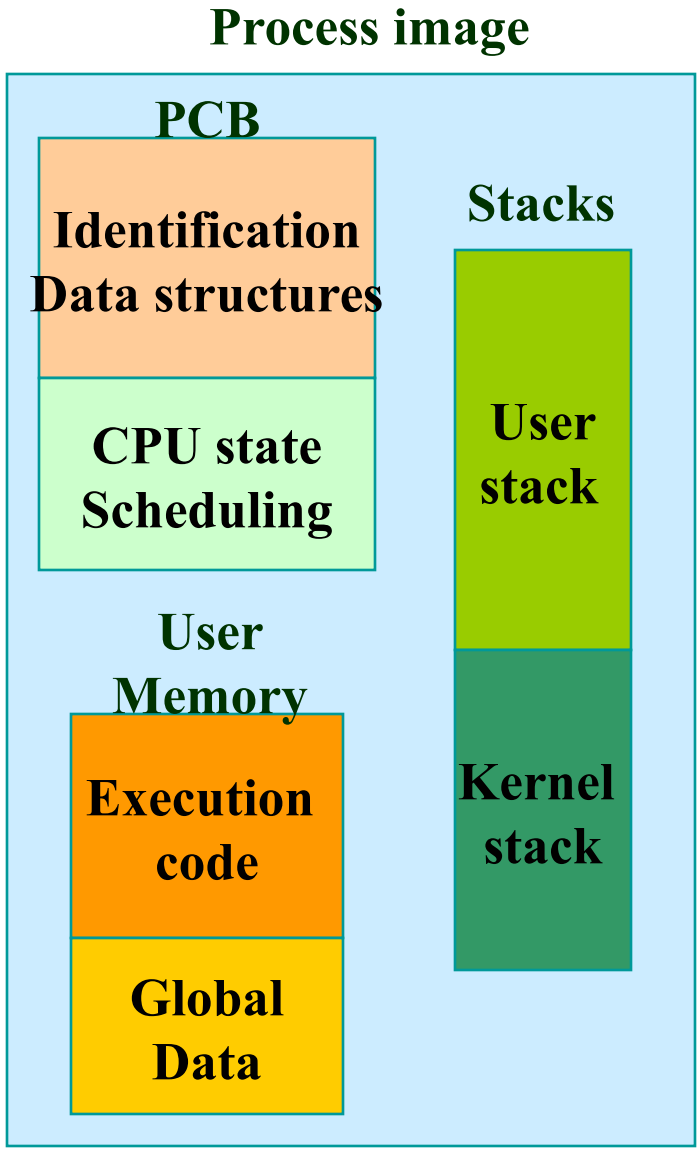
Resource Ownership

- **Related to the following components of the process image**
  - The part of the PCB that contains identification information and data structures
  - Memory containing execution code.
  - Memory containing global data



Execution → the execution thread

- Related to the following components of the process image
  - PCB
    - CPU state
    - Scheduling structures
  - Stacks



Threads vs Processes

Process

- A unit/thread of execution, together with code, data and other resources to support the execution.

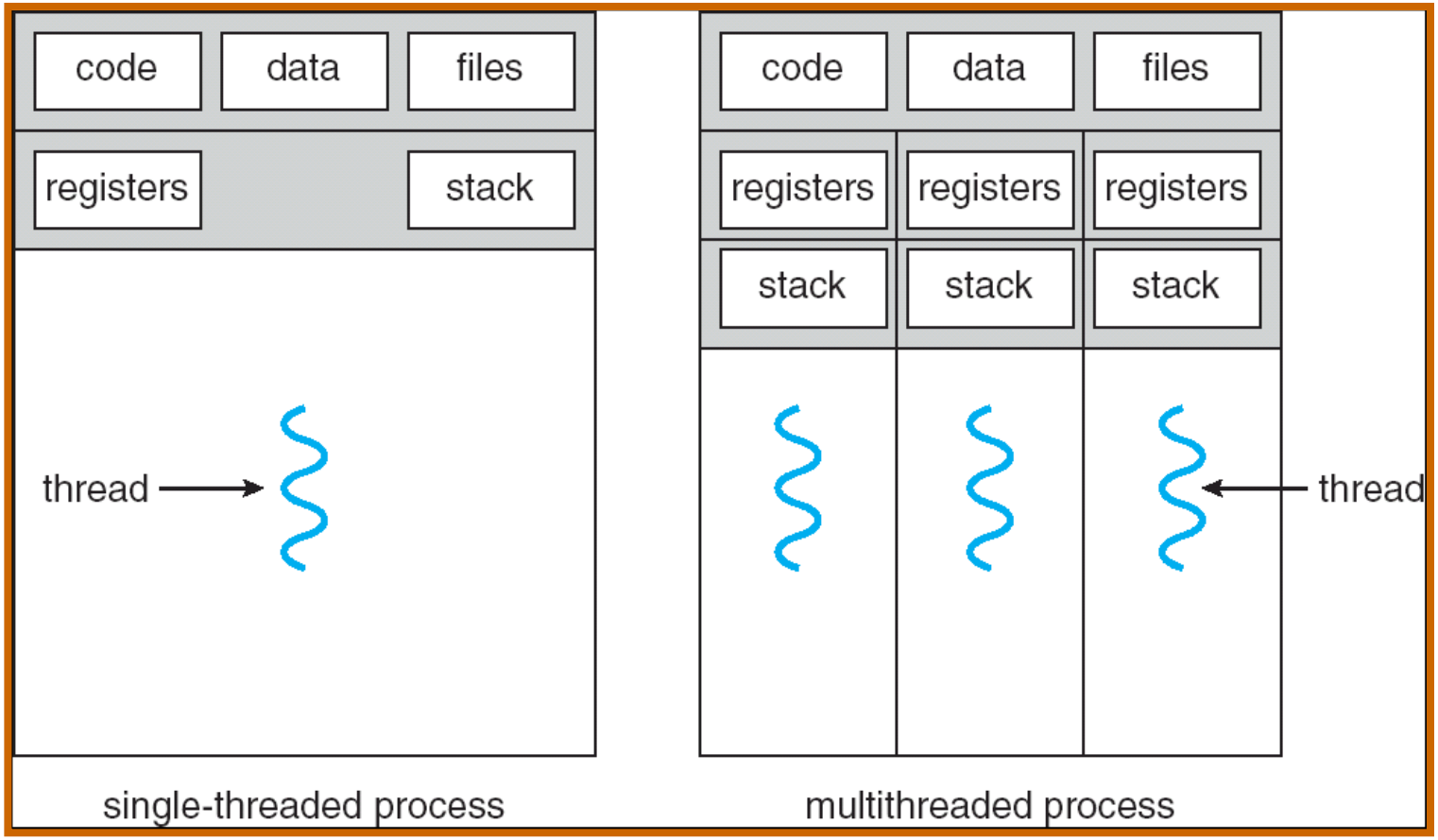
Idea

- Make distinction between the resources and the execution threads
- Could the same resources support several threads of execution?
  - Code, data, .., - yes
  - CPU registers, stack - no

Threads = Lightweight Processes

- A thread is a subdivision of a process
  - A thread of control in the process
- Different threads of a process share the address space and resources of a process.
  - When a thread modifies a global variable (non-local), all other threads sees the modification
  - An open file by a thread is accessible to other threads (of the same process).

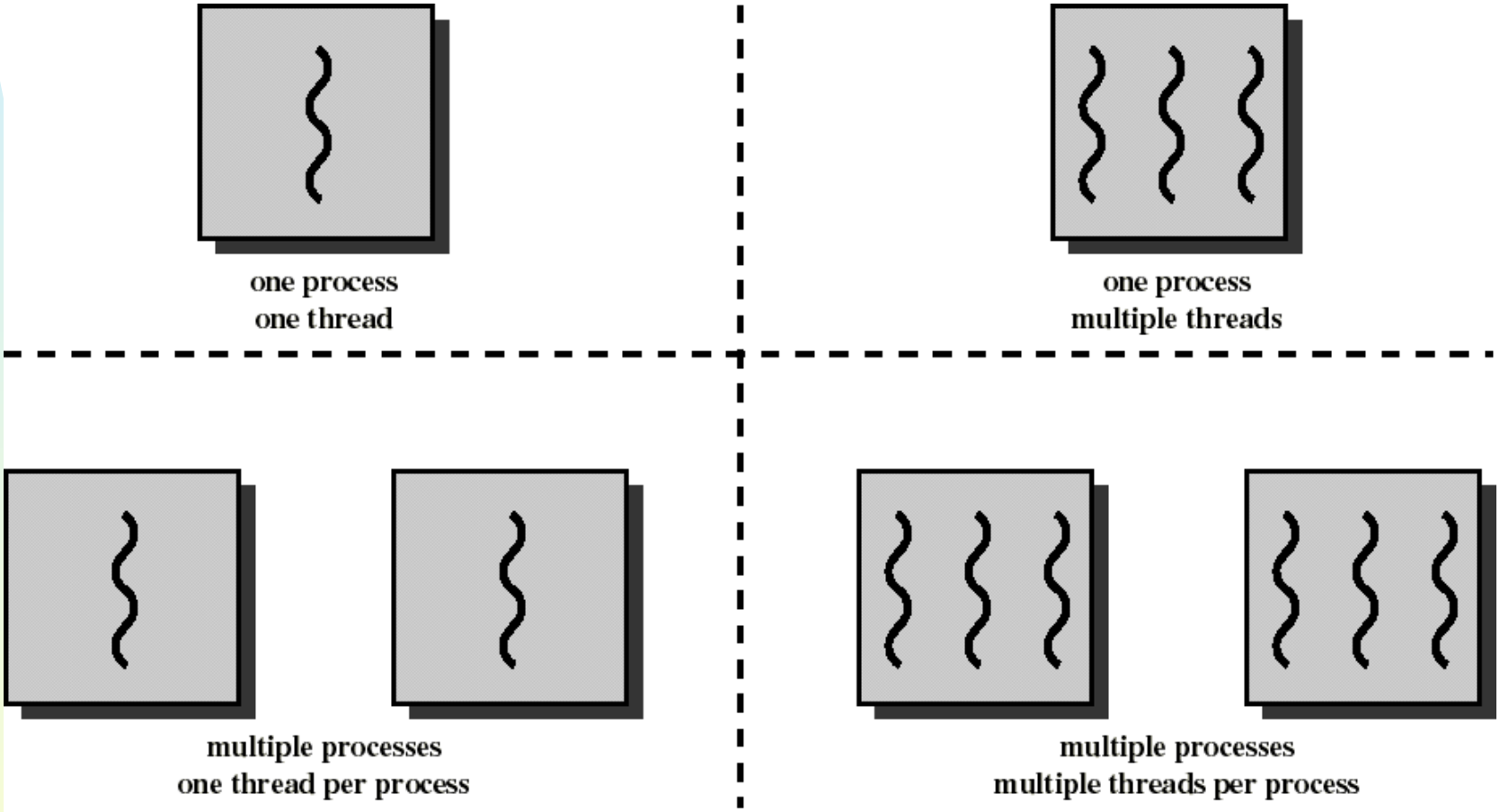
Single vs Multithreaded Processes



Example

- The MS-Word process could involve many threads:
  - Interaction with the keyboard
  - Display of characters on the display page
  - Regularly saving file to disk
  - Controlling spelling and grammar
  - Etc.
- All these threads would share the same document

Threads et processus [Stallings]



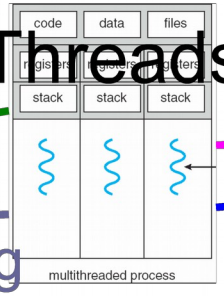
Motivation for Threads

- Responsiveness
  - One thread handles user interaction
  - Another thread does the background work (i.e. load web page)
- Utilization of multiprocessor architectures
  - One process/thread can utilize only one CPU
  - Many threads can execute in parallel on multiple CPUs
- Well, but all this applies to one thread per process as well – why use threads?

Motivation for Threads

- Switching between threads is less expensive than switching between processes
  - A process owns memory, files, and other resources
  - Changing from one process to another may involve dealing with these resources.
  - Switching between threads in the same process is much more simple – implies saving the CPU registers, the stack, and little else.
- Given that threads share memory,
  - Communication between threads within the same process is much more efficient that between processes
- The creation and termination of threads in an existing process is also much less time consuming than for new processes
  - In Solaris, creation of a thread takes about 30 times less time than creating a process

Threads



Overview

Thread of execution

Single thread verus multithread

Motivation

Mulithreading Models

User Thread Kernel Thread

Many to One Model

One to One Model

Many to Many Model

Threading Issues

Threading Examples

Kernel Threads and User Threads

- Where and how to implement threads:
  - In user libraries
    - Controlled at the level of the user program
    - POSIX Pthreads, Java threads, Win32 threads
  - In the OS kernel
    - Threads are managed by the kernel
    - Windows XP/2000, Solaris, Linux, True64 UNIX, Mac OS X
  - Mixed solutions
    - Both the kernel and user libraries are involved in managing threads
    - Solaris 2, Windows 2000/NT

User and Kernel Threads

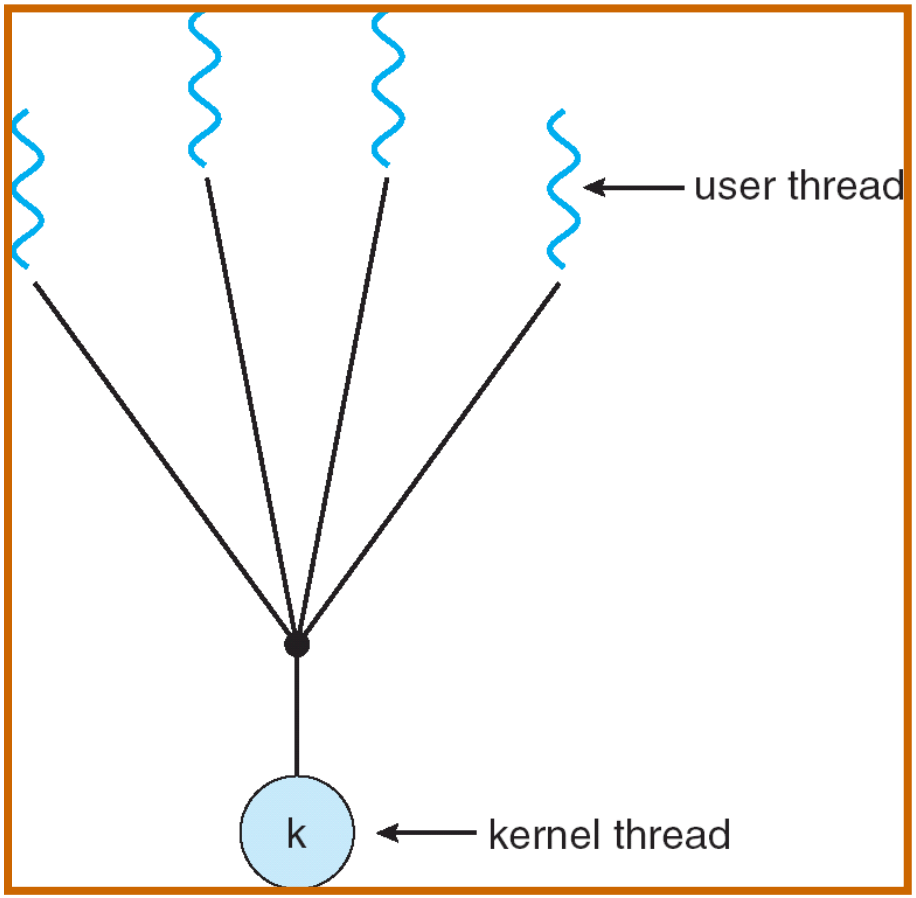
- User threads: supported with the use of user libraries or programming language
  - Efficient since operations on threads do not involve system calls
  - Disadvantage: the kernel cannot distinguish between the state of the process and the state of the threads in the process
    - Thus a blocking thread blocs the whole process
- Kernel threads: supported directly by the OS kernel (WIN NT, Solaris)
  - The kernel is capable of managing directly the states of the threads
  - It can allocate different threads to different CPUs

Multithreading Models

- Different models involve different relationships between kernel and user threads
  - Many to one model
  - One to one model
  - Many to many models (two versions)
- Must take into account the following levels:
  - Process
  - User thread
  - Kernel thread
  - Processor (CPU)

Many to one model

- All code and data structures for thread management in user space
- No system calls involved, no OS support needed
- Many (all) user threads mapped to single kernel thread
- Kernel thread gets allocated to the CPU



Many to One Model

Properties:

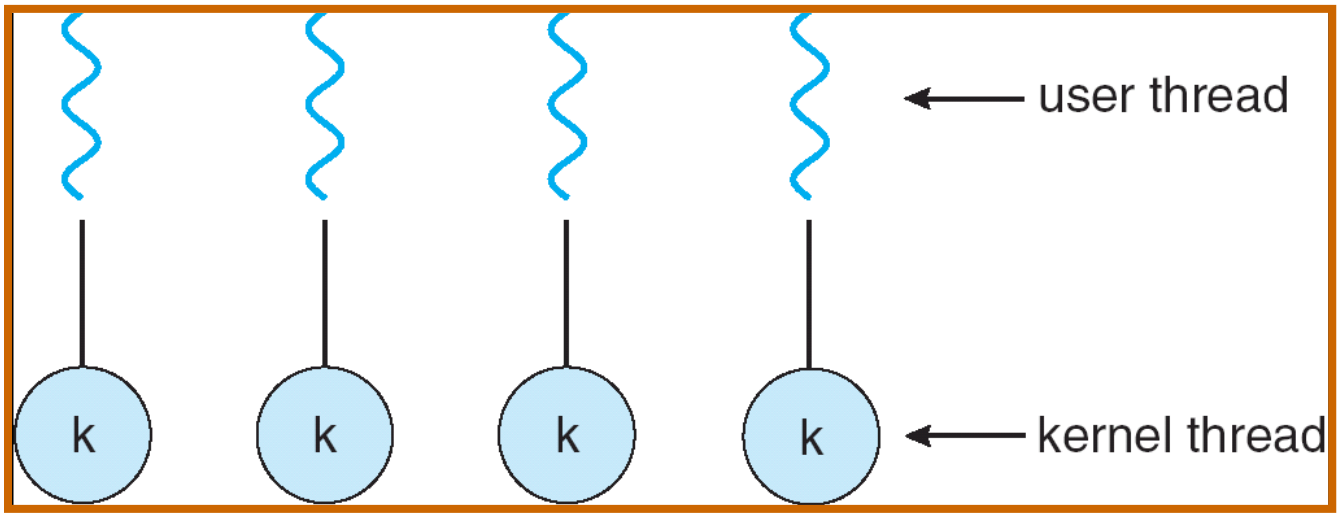
- Cheap/fast, but runs as one process to the OS scheduler
- What happens if one thread blocks on an I/O?
  - All other threads block as well
- How to make use of multiple CPUs?
  - Not possible

Examples

- Solaris Green Threads
- GNU Portable Threads

One to One Model: the kernel controls threads

- Code and data structures in kernel space
- Calling a thread library typically results in system call
- Each user thread is mapped to a kernel thread



One to One Model

Properties

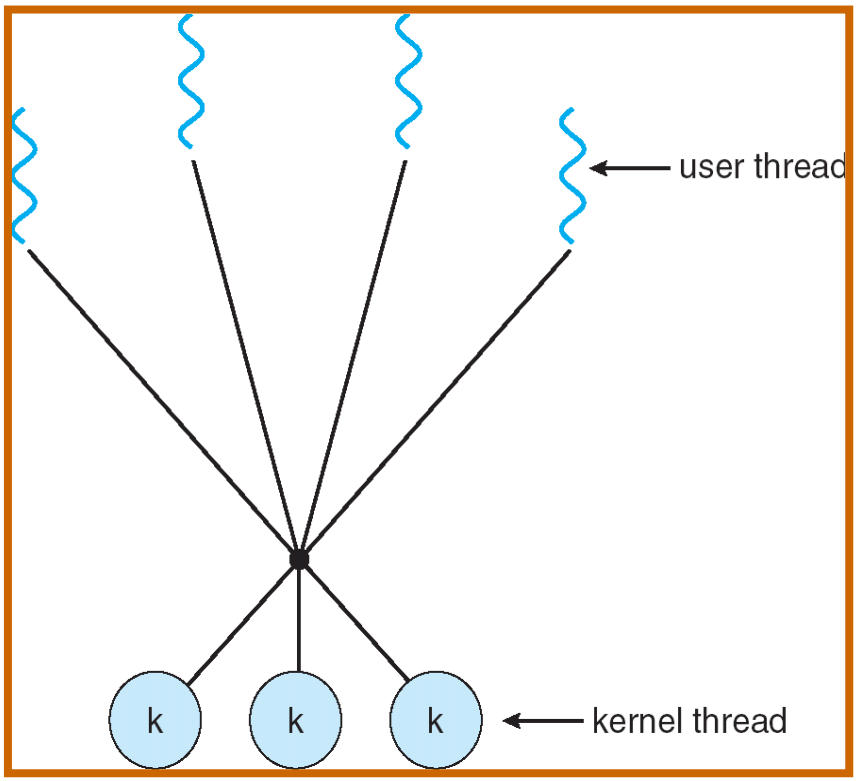
- Usually, limited number of threads
- Thread management relatively costly
- But provides better concurrency of threads

Examples

- Windows NT/XP/2000
- Linux
- Solaris Version 9 and later

Many-to-Many Model

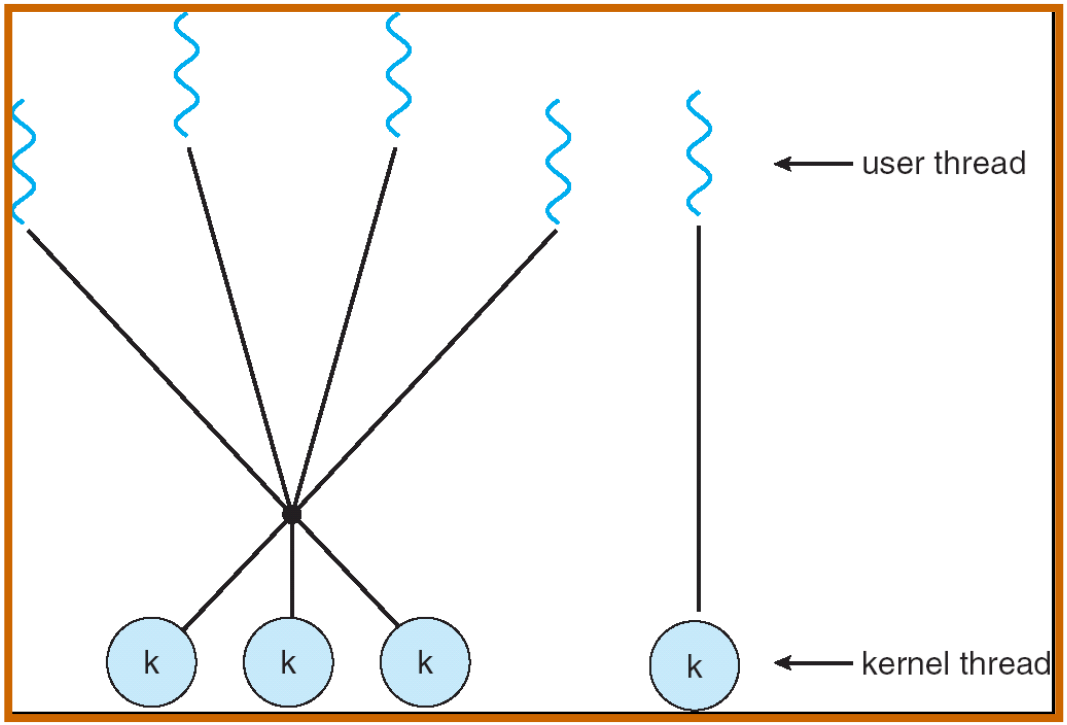
- Allows many user level threads to be mapped to many kernel threads
- The thread library cooperates with the OS to dynamically map user threads to kernel threads
- Intermediate costs and most of the benefits of multithreading
  - If a user thread blocs, its kernel thread can be associated to another user thread
  - If more than one CPU is available, multiple kernel threads can be run concurrently



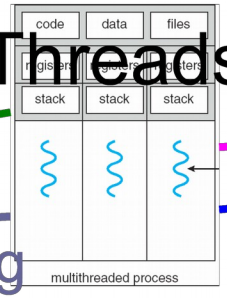
- Examples:
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package

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



Threads



Overview

Mulithreading Models

Threading Issues

Threading Examples

Thread of execution

Single thread verus multithread

Motivation

User Thread
Kernel Thread

Many to One Model

One to One Model

Many to Many Model

Creation and Termination

Fork/exec

Cancellation

Pools

Signal Handling

Thread Specific Data

Scheduler Activations

Threading Issues – Creation/Termination Semantics of fork() and exec()

- **Does** fork() **duplicate only the calling thread or all threads?**
  - **Often two versions provided**
  - **Which one to use?**
- **What does exec() do?**
  - **Well, it replaces the address space, so all threads must go**

Threading Issues – Creation/Termination Thread Cancellation

- **Terminating a thread before it has finished**
- **Two general approaches:**
  - Asynchronous cancellation **terminates the target thread immediately**
    - **Might leave the shared data in corrupt state**
    - **Some resources may not be freed**
  - Deferred cancellation
    - **Set a flag which the target thread periodically checks to see if it should be cancelled**
    - **Allows graceful termination**

Threading Issues – Creation/Termination Thread Pools

- A server process might service requests by creating a thread for each request that needs servicing
  - But thread creation costs are wasted time
  - No control over the number of threads, possibly overwhelming the system
- Solution
  - Create a number of threads in a pool where they await work
  - Advantages:
    - The overhead for creating threads is paid only at the beginning
    - Allows the number of threads in the application(s) to be bound to the size of the pool

Threading Issues - Signal Handling

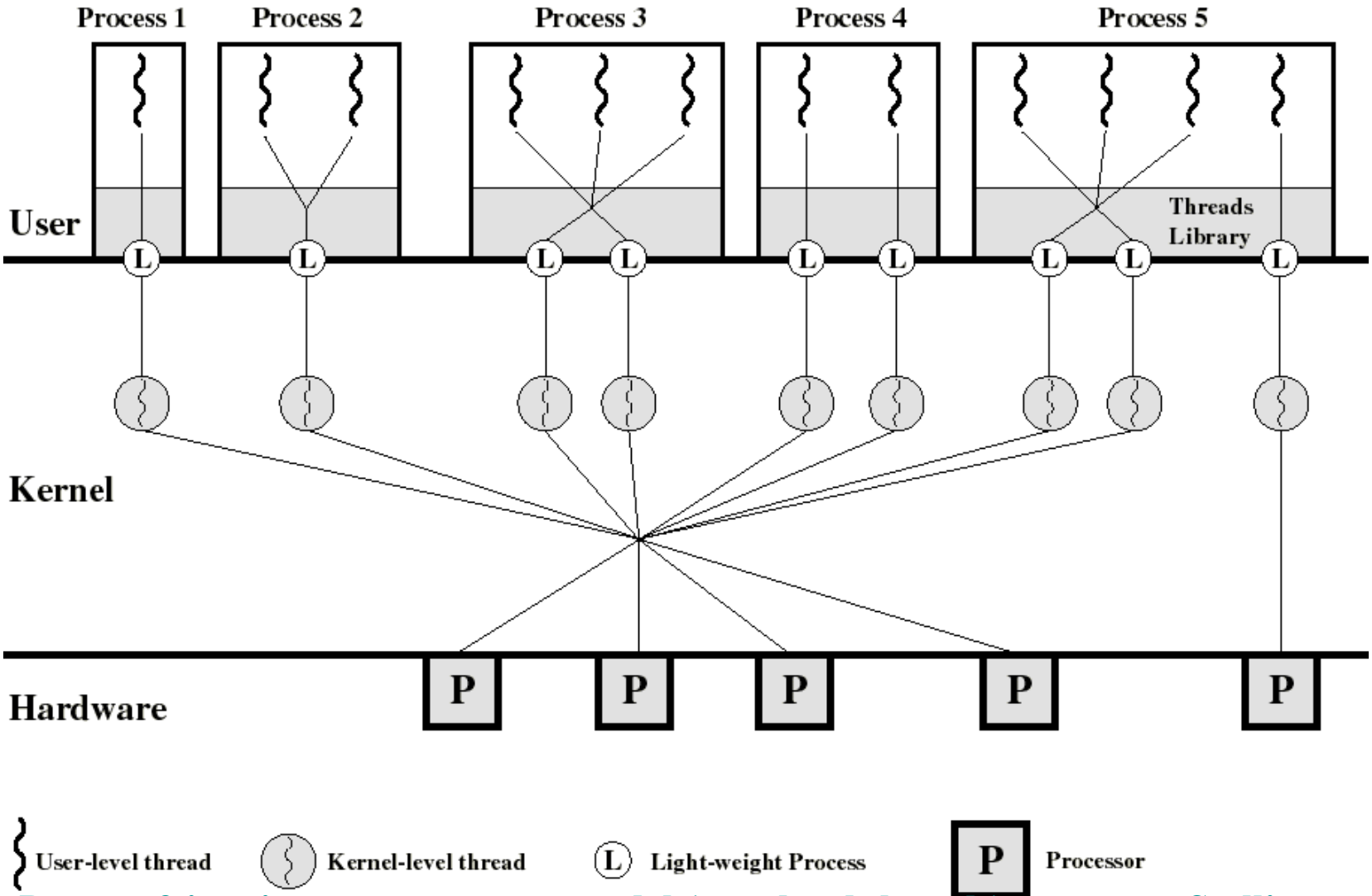
- **Signals are used in UNIX systems to notify a process that a particular event has occurred**
- **Essentially software interrupt**
- **A** signal handler **is used to process signals**
  1. **Signal is generated by particular event**
  2. **Signal is delivered to a process**
  3. **Signal is handled**
- **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 thread to receive all signals for the process**

Threading Issues - 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)

Threading Issues - Scheduler Activations

- **The many to many models (including two level) require communication from the kernel to inform the thread library when a user thread is about to block, and when it again becomes ready for execution**
- **When such event occurs, the kernel makes an** upcall **to the thread library**
- **The thread library's** upcall handler **handles the event (i.e. save the user thread's state and mark it as blocked)**

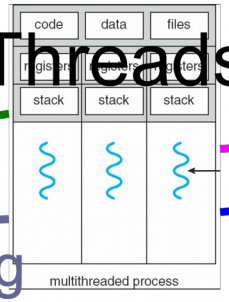


Process 2 is using a many to one model (user level threads) Stallings

Process 4 is using one to one model (kernel level threads)

Level of parallelism can be adjusted for process 3 and process 5 (many to many model)

Threads



Overview

Thread of execution

Single thread verus multithread

Motivation

Mulithreading Models

User Thread
Kernel Thread

Many to One Model

One to One Model

Many to Many Model

Threading Issues

Creation and Termination

Fork/exec

Cancellation

Pools

Signal Handling

Thread Specific Data

Scheduler Activations

Threading Examples

Libraries

PThreads

Win32

Java

Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to the developer of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- Typical functions:
  - pthread_create (&threadid, attr, start_routine, arg)
  - pthread_exit (status)
  - pthread_join (threadid, status)
  - pthread_attr_init (attr)

192.168.57.2 - siteDev* - SSH Secure Shell

File Edit View Window Help

Quick Connect Profiles

```
PTHREAD_CREATE(3)                                        PTHREAD_CREATE(3)

NAME
       pthread_create - create a new thread

SYNOPSIS
       #include <pthread.h>

       int  pthread_create(pthread_t  *  thread, pthread_attr_t * attr, void *
       (*start_routine)(void *), void * arg);

DESCRIPTION
       pthread_create creates a new thread of control  that  executes  concur-
       rently  with  the  calling  thread. The new thread applies the function
       start_routine passing it arg as first argument. The new  thread  termi-
       nates  either explicitly, by calling pthread_exit(3), or implicitly, by
       returning from the start_routine function. The latter case  is  equiva-
       lent  to calling pthread_exit(3) with the result returned by start_rou-
       tine as exit code.

       The attr argument specifies thread attributes to be applied to the  new
       thread.   See  pthread_attr_init(3)  for  a  complete  list  of  thread
       attributes. The attr argument can also be NULL, in which  case  default
:
```

Connected to 192.168.57.2 | SSH2 - aes128-cbc - hmac-md5 - none | 80x24

Thread Programming Exercise

Goal: Write multithreaded matrix multiplication algorithm, in order to make use of several CPUs.

Single threaded algorithm for multiplying n x n matrices A and B :

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++) {
        C[i,j] = 0;
        for(k=0; k<n; k++)
            C[i,j] += A[i,k] * B[k,j];
    }
```

Just to make our life easier:

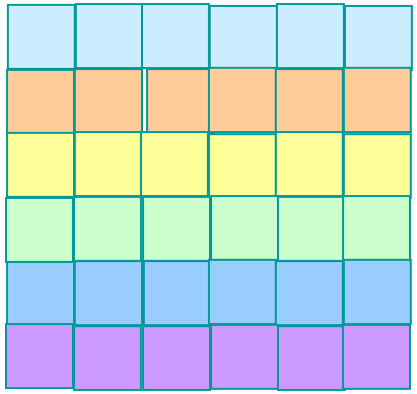
Assume you have 6 CPUs and n is multiple of 6.

How to start? Any ideas?

Multithreaded Matrix Multiplication

Idea:

- create 6 threads
- have each thread compute 1/6 of the matrix C
- wait until everybody finished
- the matrix can be used now



Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5

Let's go!

```
pthread_t tid[6];
pthread_attr_t attr;
int i;

pthread_init_attr(&attr);
for(i=0; i<6; i++)    /* create the working threads */
   pthread_create( &tid[i], &attr, worker, &i);

for(i=0; i<6; i++)  /* now wait until everybody finishes */
   pthread_join(tid[i], NULL);

/* the matrix C can be used now */
…
```

Let's go!

```
void *worker(void *param)
{
   int i,j,k;
   int id = *((int *) param); /* take param to be
                                 pointer to integer */
   int low = id*n/6;
   int high = (id+1)*n/6;

   for(i=low; i<high; i++)
      for(j=0; j<n; j++)
      {
         C[i,j] = 0;
         for(k=0; k<n; k++)
            C[i,j] = A[i,k]*B[k,j];
      }
   pthread_exit(0);
}
```

Let's go!

Would it work?

- do we need to pass A,B,C and n in the parameters?
  - no, they are in the shared memory, we are fine
- did we pass IDs properly?
  - not really, all threads get the same pointer

```
int id[6];
.
.
for(i=0; i<6; i++)    /* create the working threads */
{
    id[i] = i;
    pthread_create( &tid[i], &attr, worker, &id[i]);
}
```

Would it work now?

- should, …

Win32 Thread API

```
// thread creation:
ThreadHandle = CreateThread(
    NULL,         // default security attributes
    0,            // default stack size
    Summation,    // function to execute
    &Param,               // parameter to thread function
    0,            // default creation flags
    &ThreadId);   // returns the thread ID

if (ThreadHandle != NULL) {
    WaitForSingleObject(ThreadHandle, INFINITE);
    CloseHandle(ThreadHandle);
    printf("sum = %d\n",Sum);
}
```

See Silbershatz for simple example

Java Threads

- Java threads are created by calling a start() method of a class that
  - extends Thread class, or
  - implements the Runnable interface:

```
public interface Runnable
{
    public abstract void run();
}
```

- Java threads are inherent and important part of the Java language, with rich API available

Extending the Thread Class

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

public class First
{
    public static void main(String args[]) {
        Worker1 runner = new Worker1();
        runner.start();
        System.out.println("I Am The Main Thread");
    }
}
```

Implementing the Runnable Interface

```
class Worker2 implements Runnable
{
    public void run() {
        System.out.println("I Am a Worker Thread ");
    }
}
public class Second
{
    public static void main(String args[]) {
        Runnable runner = new Worker2();
        Thread thrd = new Thread(runner);
        thrd.start();

        System.out.println("I Am The Main Thread");
    }
}
```

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");
    }
}
```

Thread Cancellation

```
Thread thrd = new Thread (new InterruptibleThread());
thrd.start();

. . .

// now interrupt it
thrd.interrupt();
```

Thread Cancellation

```
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
    }
}
```

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();
   }
}
```

Thread Specific Data

```
class Worker implements Runnable
{
   private static Service provider;

   public void run() {
      provider.transaction();
      System.out.println(provider.getErrorCode());
   }
}


Somewhere in the code:
…
Worker worker1 = new Worker();
Worker worker2 = new Worker();
worker1.start();
worker2.start();
…
```

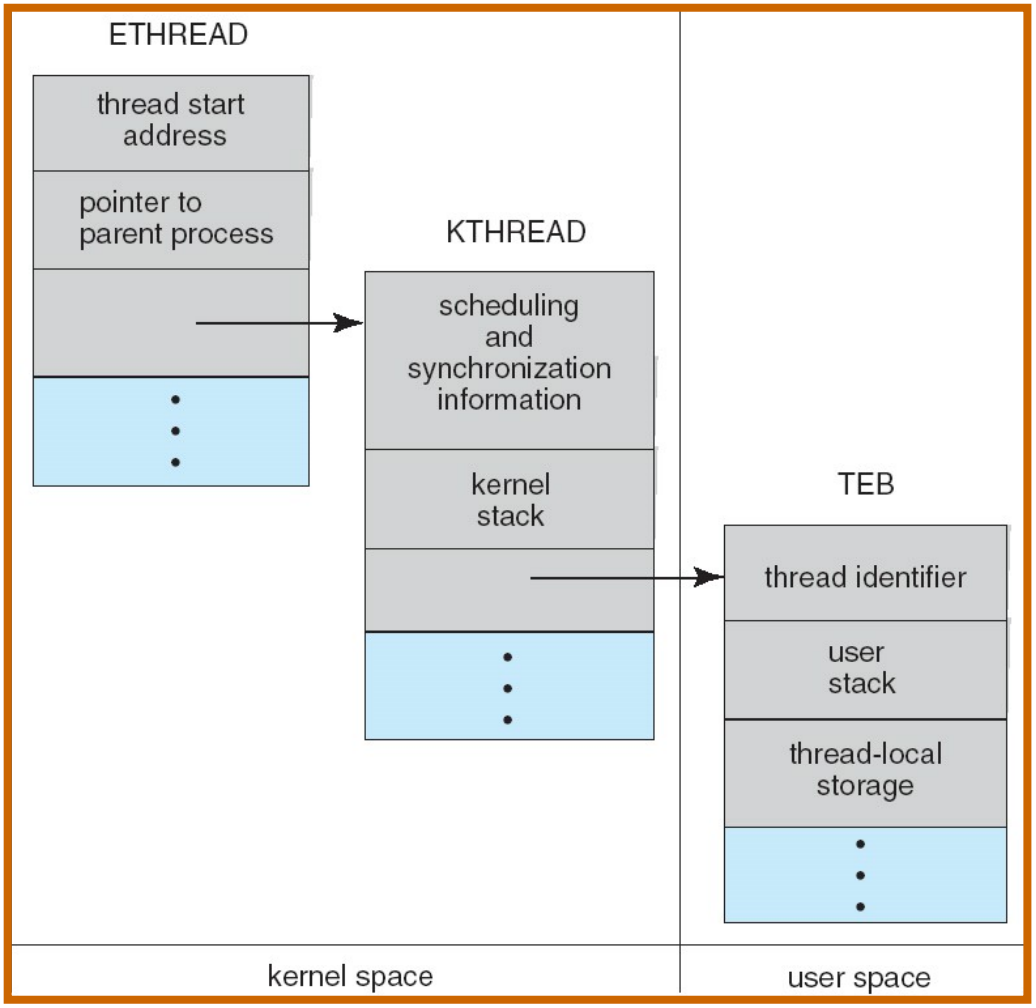
Assume there were different errors in the transactions of both workers, but both transactions finished before any of the prints in the run() methods started.

Would the workers print the same errors or not?

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

Windows XP Threads



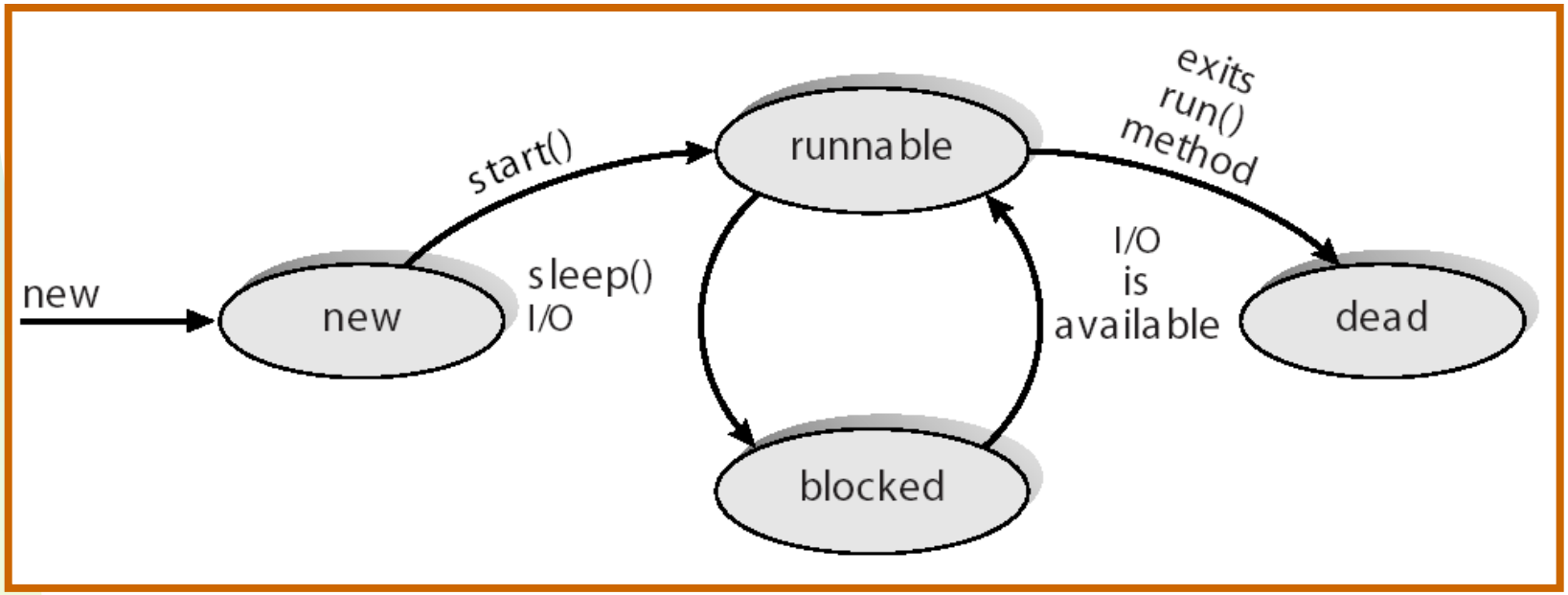
Linux Threads

- **Linux refers to them as *tasks* rather than *threads***
- **Thread creation is done through** clone() **system call**
- **The** clone() **system call allows to specify which resources are shared between the child and the parent**
  - **Full sharing → threads**
  - **Little sharing → like fork()**

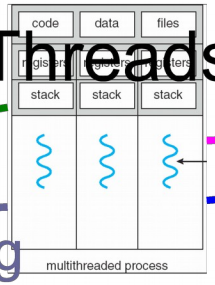
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Java Threads

Java threads are managed by the JVM



Threads



Overview

Thread of execution

Single thread verus multithread

Motivation

Mulithreading Models

User Thread Kernel Thread

Many to One Model

One to One Model

Many to Many Model

Threading Issues

Creation and Termination

Fork/exec

Pools

Cancellation

Signal Handling

Thread Specific Data

Scheduler Activations

Threading Examples

Libraries

PThreads

Win32

Java

Implementations

Windows XP

Linux

Java