

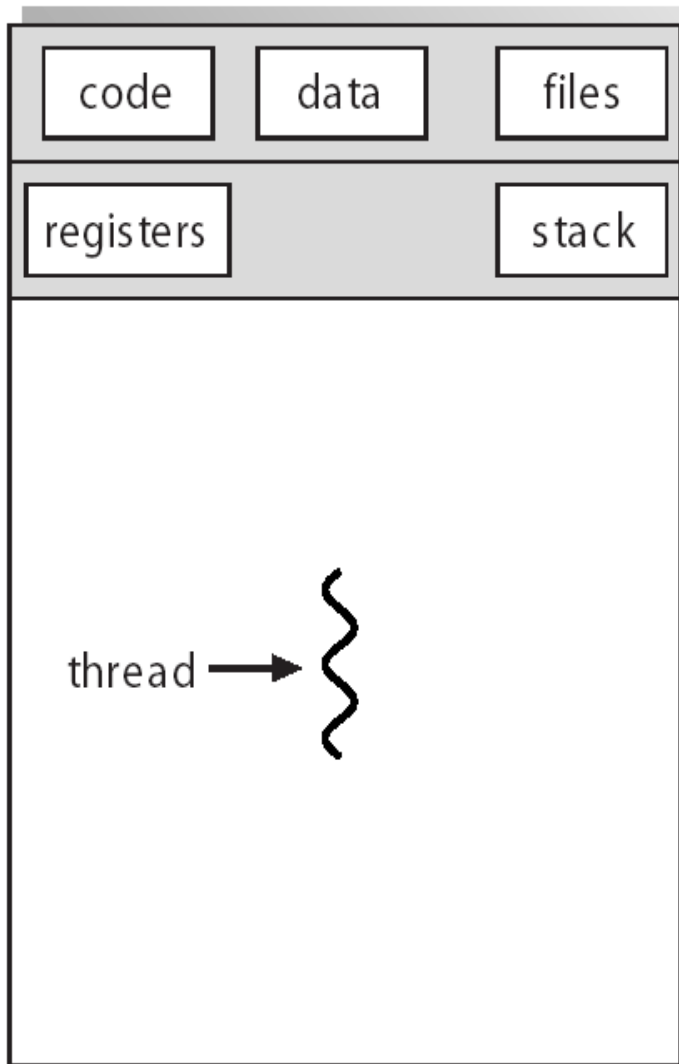


# THREADS

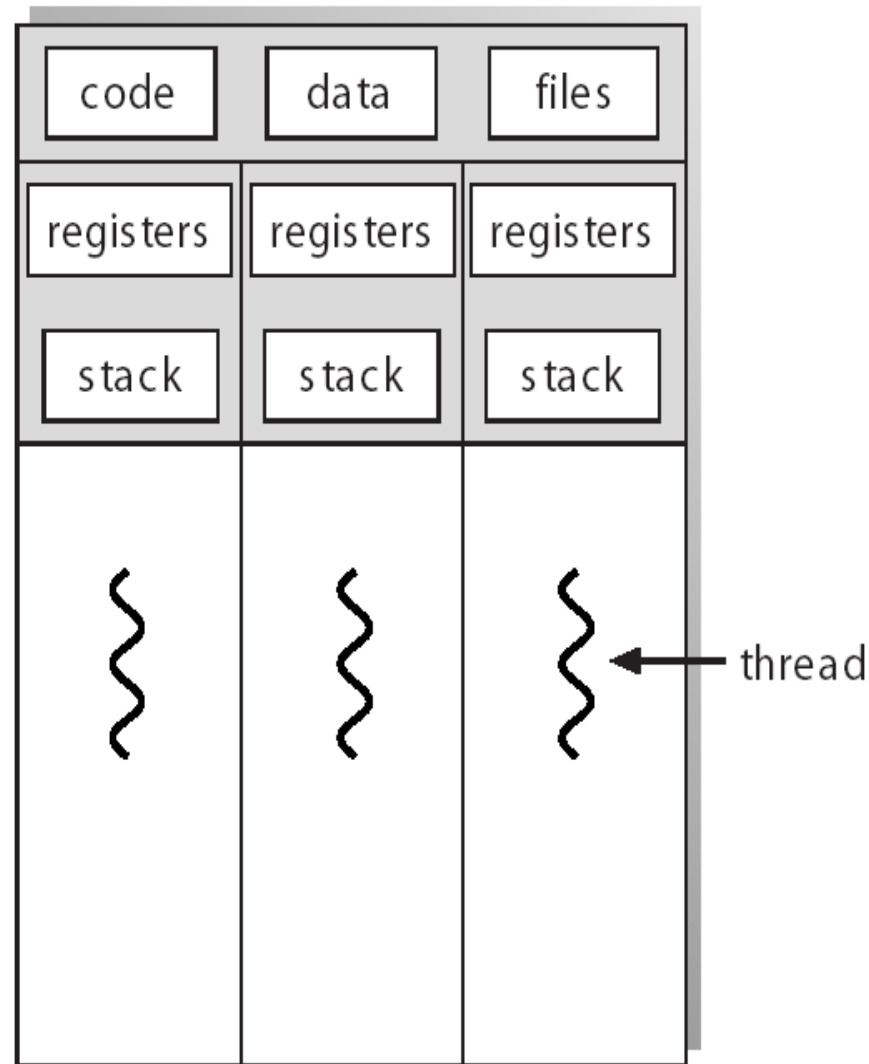
Date

# Threads

- ◆ A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
  - Thread ID
  - program counter
  - register set
  - stack space
- ◆ A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resourcescollectively know as a *task*.
- ◆ A traditional or *heavyweight* process is equal to a task with one thread

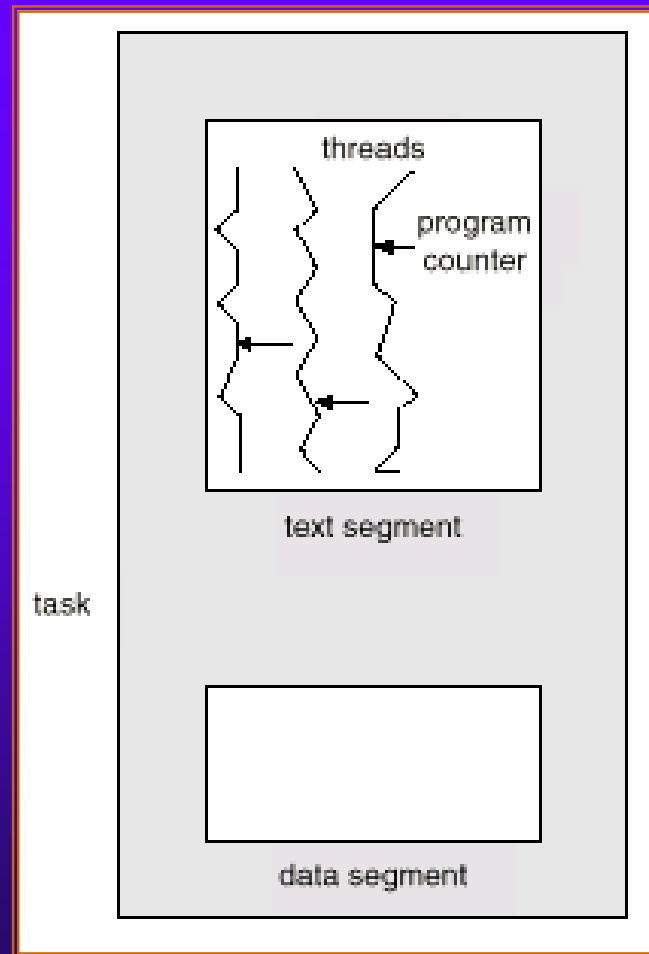


single-threaded process



multithreaded process

# Multiple Threads within a Task



# Benefits

## ◆ Responsiveness

- Even if one thread is blocked other threads can continue execution.

## ◆ Resource sharing

- Sharing memory & other resources of the process it belongs to.

## ◆ Economy

- It is more economical to create & context switch threads.

## ◆ Utilization of multiprocessor architectures

- Increases multi threading ( threads can run parallel)



## ◆ User threads

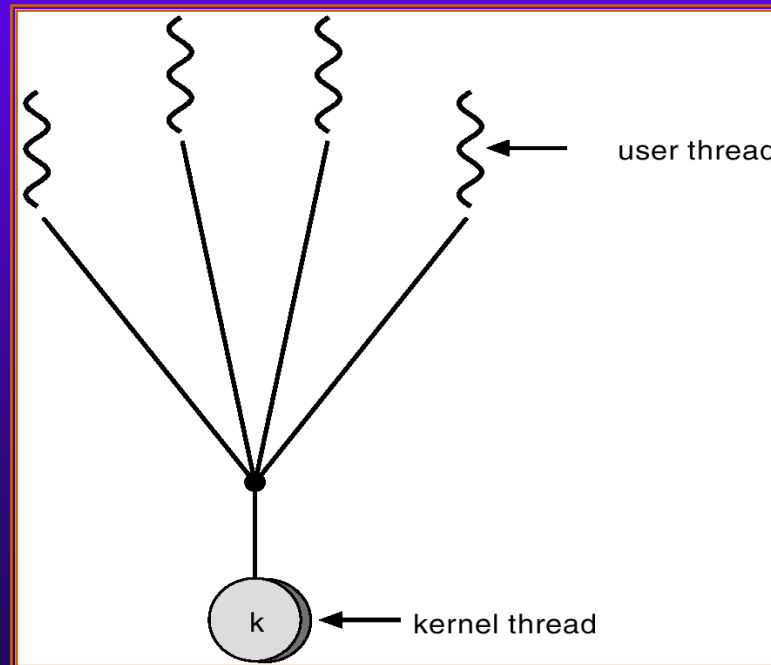
- These threads are supported above the kernel
- Implemented by thread library at the user level
- Fast to create and manage
- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Java threads
  - Win32 threads

## ◆ Kernel threads

- Supported directly by OS
- Slow compared to user threads
- Examples
  - Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X

# Multi threading models

- ◆ Many to one model
  - Many user threads and one kernel thread
  - Thread management is done in user space so it is efficient
  - Because of only one kernel thread the entire process will block if a thread makes a block.
  - Only one thread can access the kernel at a time (unable to run in parallel on multiprocessors)

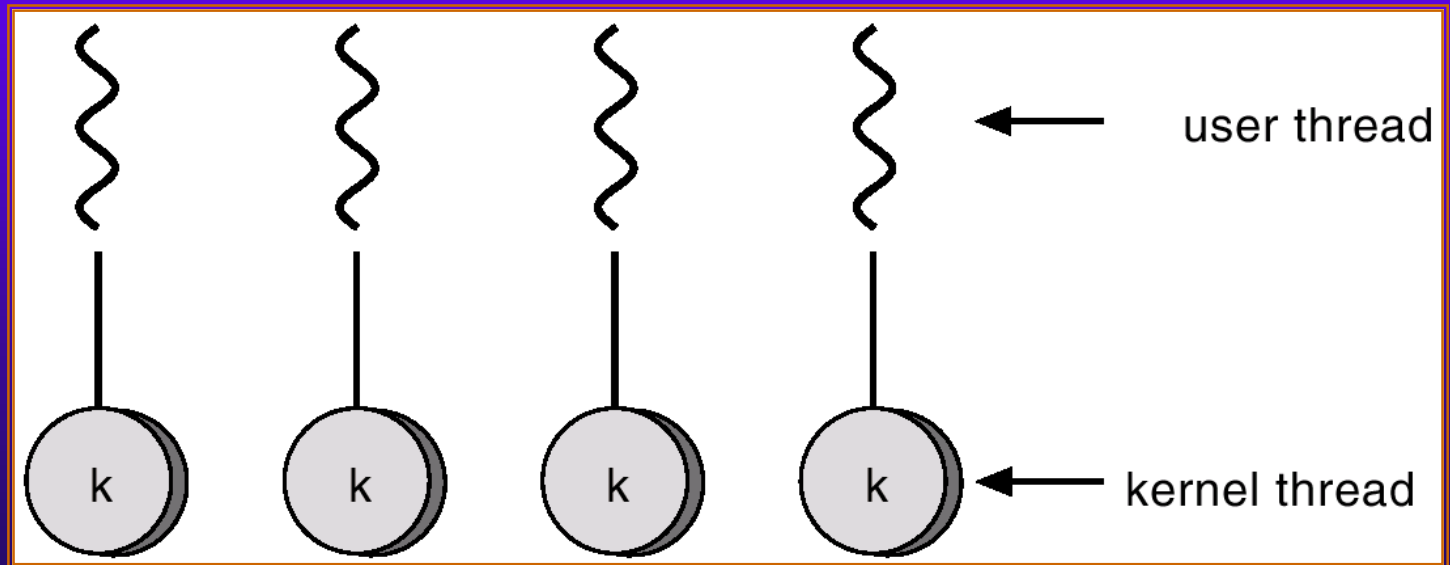


◆ One to one model

- One kernel thread for one user thread
- Provide more concurrency than many to one model
- Allows to run threads in parallel on multiprocessors
- Main drawback is for one user thread we need to create one kernel thread which may degrade the performance.

◆ Examples

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

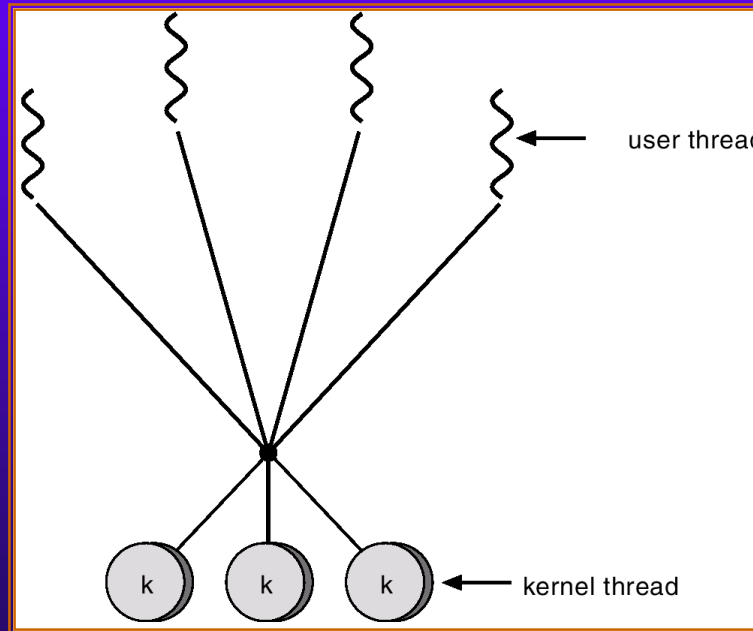


- ◆ Many to many model

- Multiplexes many user threads to a smaller or equal number of kernel threads

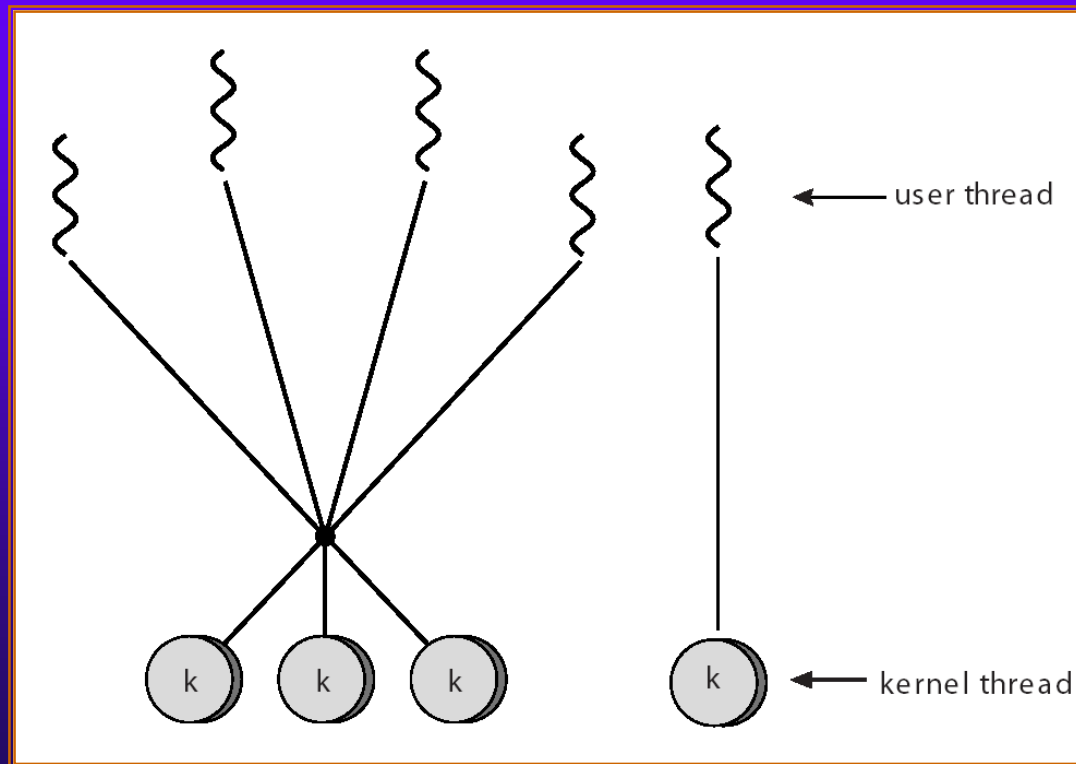
- ◆ Examples

- Solaris prior to version 9, Windows NT/2000 with the *Thread Fiber* package



## ◆ 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



# Pthread example program

```
#include < pthread.h >
#include < stdio.h >

Int sum;          // shared data by the threads

Void *runner ( void *param );    // the thread

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

# Pthread example program cont.....

```
// runner function
```

```
Void *runner ( void *param )  
{  
    int upper = atoi (param);  
    int I;  
    sum=0;  
    if ( upper > 0 )  
    {  
        for ( I=1; I <= upper; I++ )  
        {  
            sum = sum + I;  
        }  
    }  
    pthread_exit ( 0 );  
}
```

# Threading Issues

- The fork and exec system calls
  - If one thread in a system calls `fork()`
    - The new process duplicates all threads
    - The new process duplicates only the calling thread.
- Cancellation
  - Task of terminating the thread before it has completed.
  - Cancellation of target thread (the thread that is to be cancelled) may occur in 2 different scenarios
    - Asynchronous cancellation
    - Deferred cancellation
- Signal handling
  - A signal is generated by the occurrence of particular event
  - A generated signal is delivered to the process
  - Once delivered the signal must be cancelled

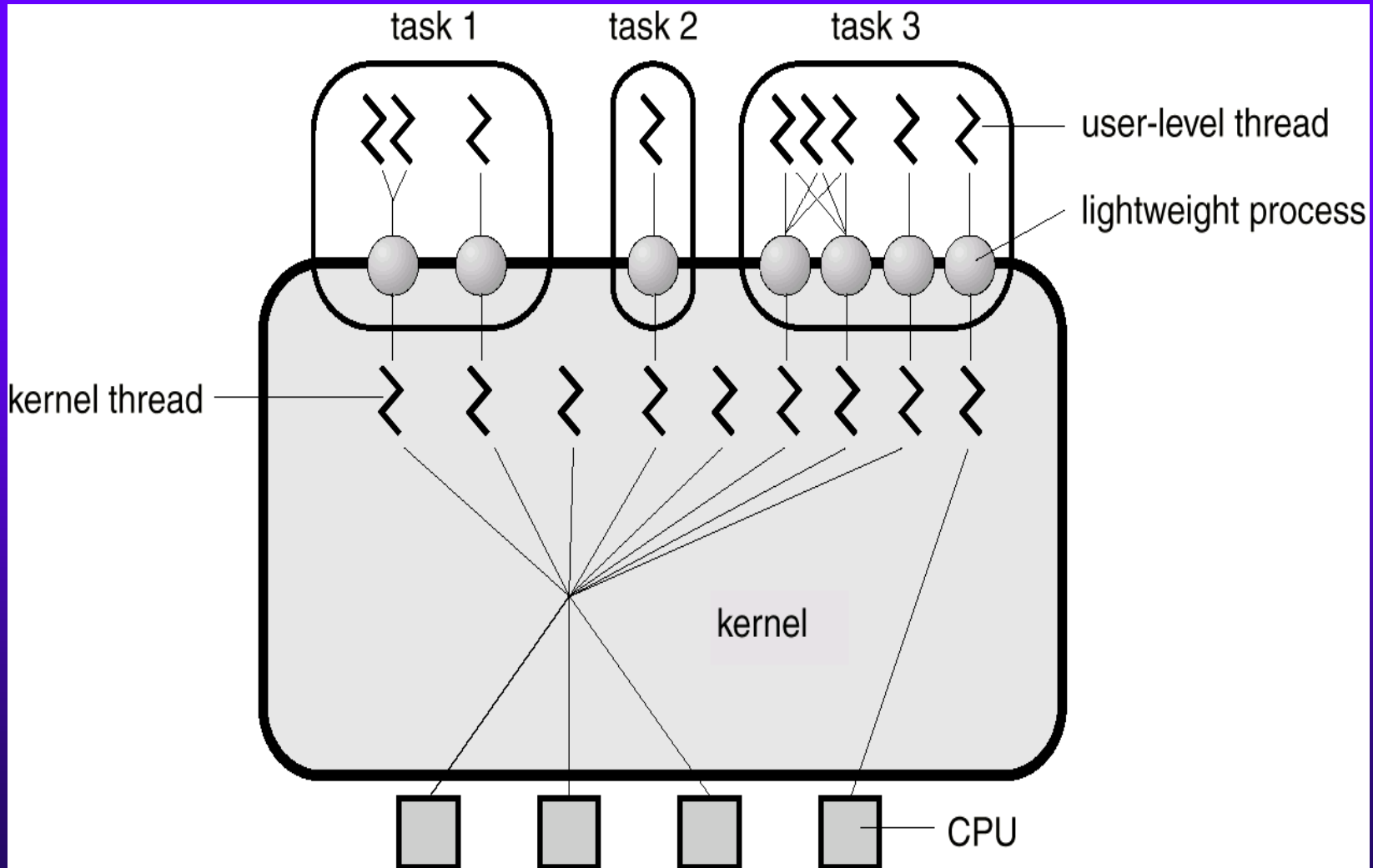


- Signal may be handled by
  - A default signal handler
  - A user defined signal handler
- Delivering signals in multi threaded program
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for process.
- Thread pools
  - Unlimited threads could exhaust system resources and one solution for this is pooling.
  - In this create a number of threads at process startup and place them into a pool. When the server require the thread it is allotted and after completion of task it will be back on pool.
  - This method is usually faster to service a request with an existing thread than waiting to create a thread.
  - A thread pool limit the number of threads that exist at any time
- Thread specific data
  - Each thread may need its own copy of data like local variables

# 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 **up calls** - a communication mechanism from the kernel to the thread library
- ◆ This communication allows an application to maintain the correct number kernel threads

# Solaris 2 threads





- 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.
  - The user level thread can be either Bound (permanently attached to an LWP) or unbound ( not permanently connected to any LWP).threads are unbound by default.
- Resource needs of thread types:
  - Kernel thread: small data structure(copy of the kernel registers, a pointer to the LWP to which it is attached, priority and scheduling information) and a stack; thread switching does not require changing memory access information – relatively fast.
  - LWP: PCB with register data, accounting and memory & accounting information,; switching between LWP s is relatively slow.it is a kernel data structure and it resides in kernel space.
  - User-level thread: only need thread ID,stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWP s that support user-level threads.



# 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)