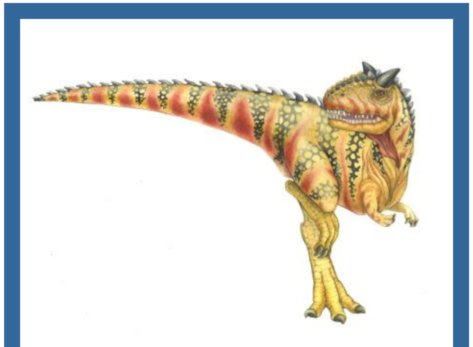
Threads

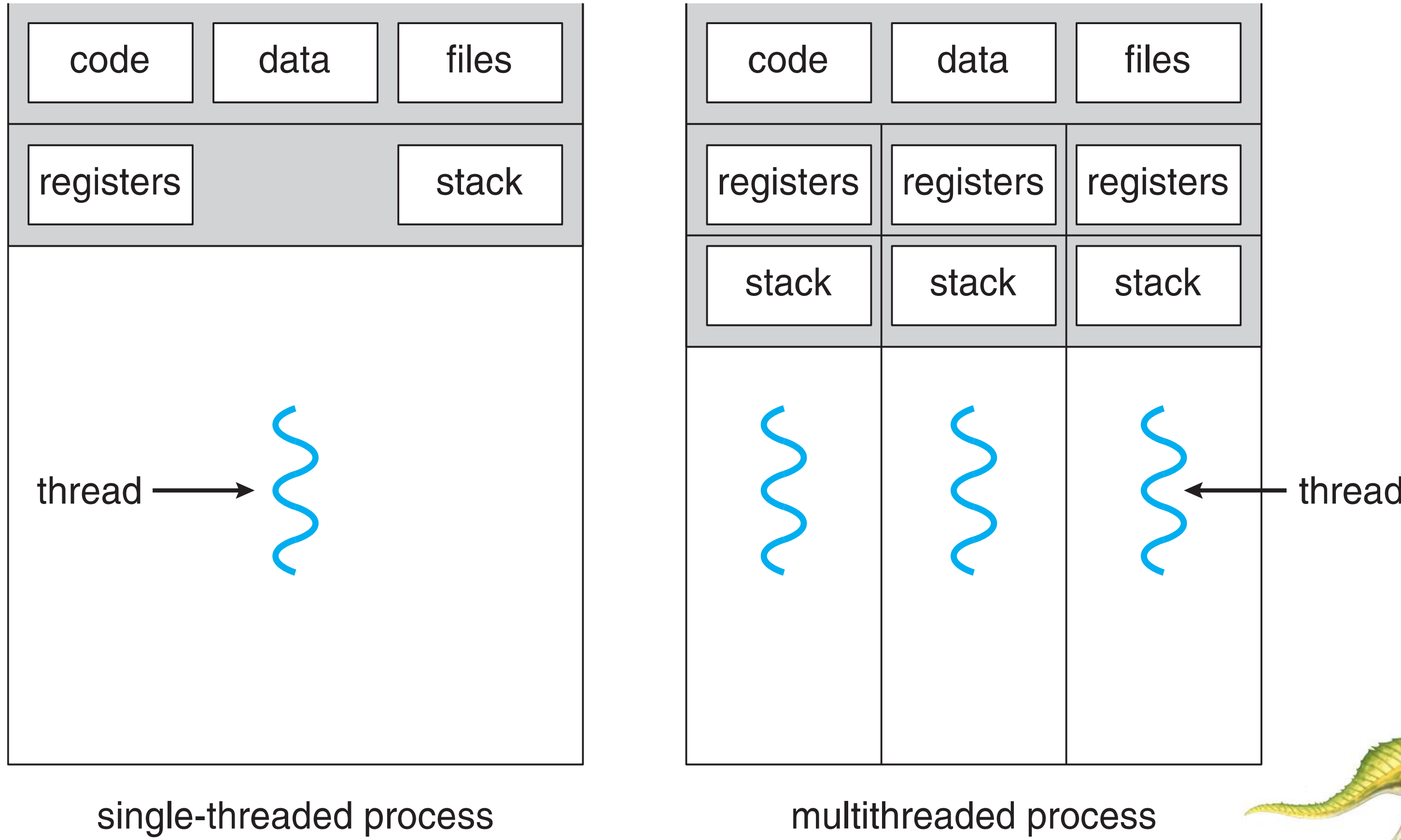


Threads

- A thread is a basic unit of CPU utilization;
  - it comprises
    - a **thread ID,**
    - **a program counter,**
    - **a register set, and**
    - **a stack.**
  - It shares with other threads belonging to the same process its **code section, data section, and other operating-system resources**, such as open files and signals.

Single and Multithreaded Processes

- A traditional (or ***heavy weight***) process has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time.



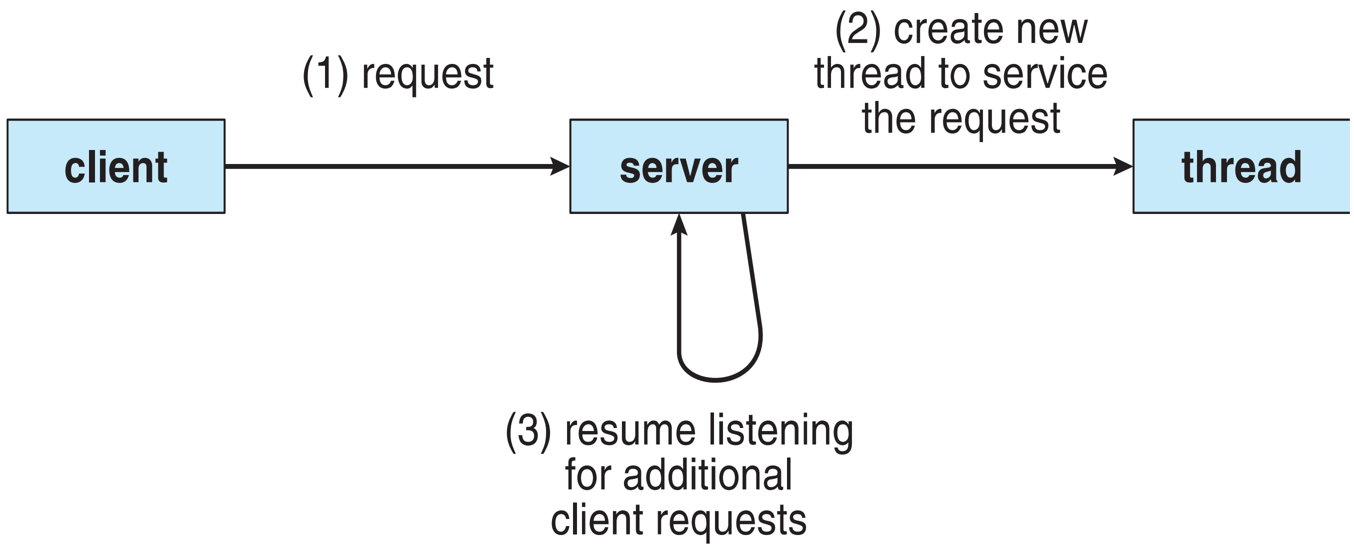
Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display-a thread for displaying graphics
  - Fetch data-another thread for responding to keystrokes from the user
  - Spell checking-a third thread for performing spelling and grammar checking in the background
- Applications can also be designed to leverage processing capabilities on multicore systems.
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Simple Thread Program

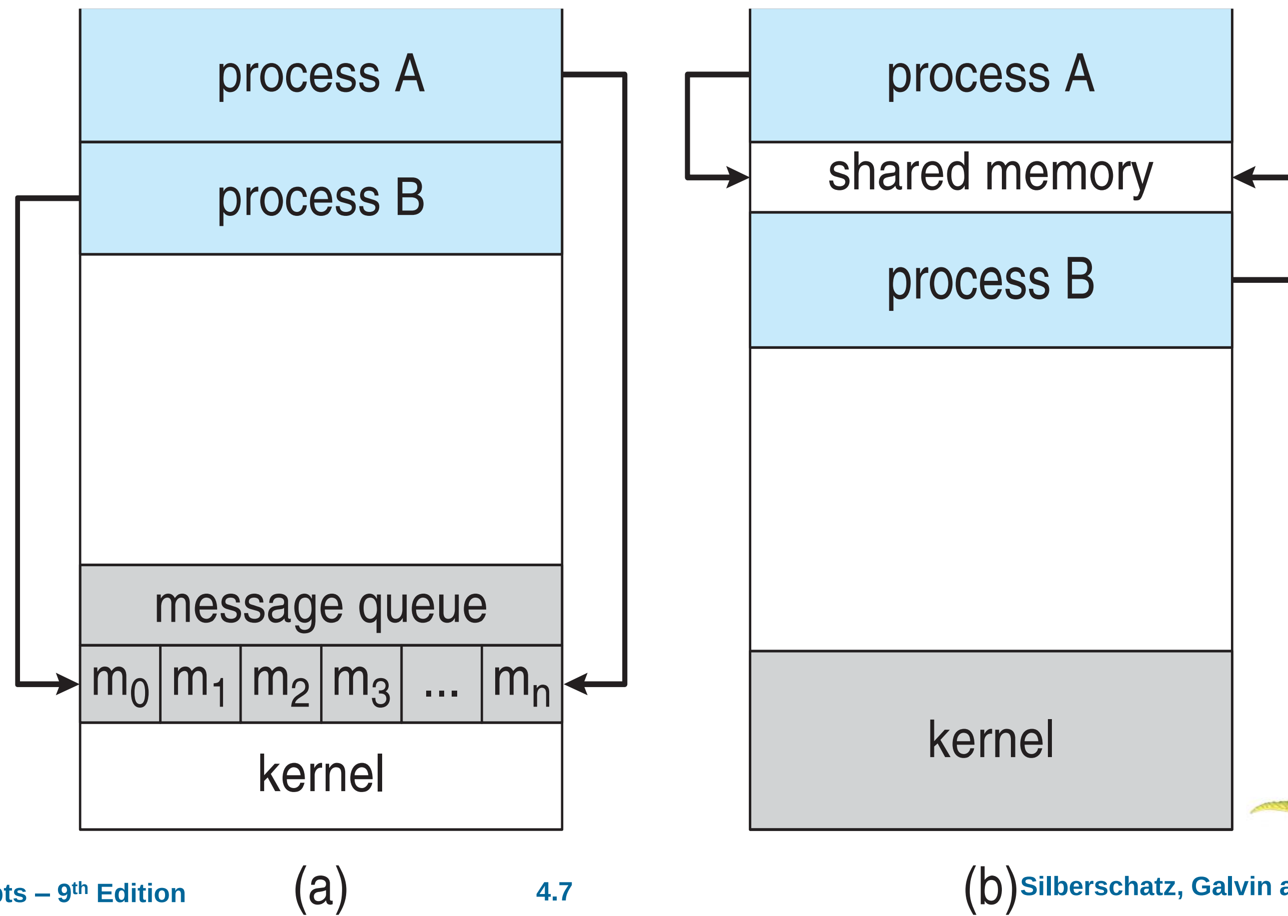
```c
#include <pthread.h>
#include <stdio.h>

void* thread_code( void * param )
{
        printf( "In thread code\n" );
}
int main()
{
        pthread_t thread;
        pthread_create(&thread, 0, &thread_code, 0 );
        printf("In main thread\n" );
}
```

Multithreaded Server Architecture



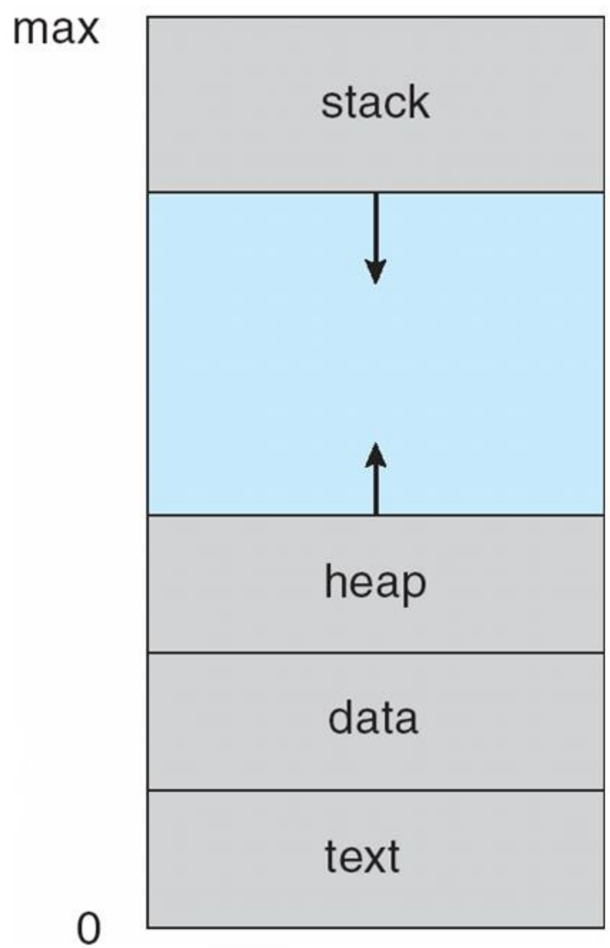
Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing



Benefits

- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures



User Threads and Kernel Threads

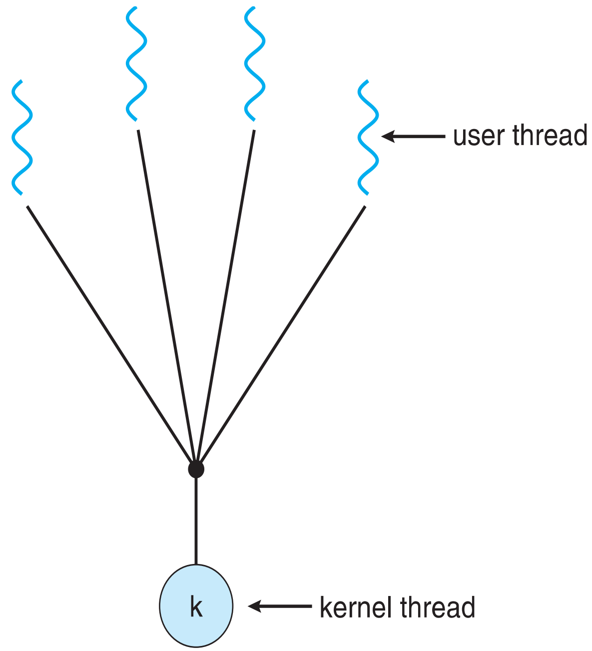
- **User threads** - management done by user-level threads library
  - managed without kernel support
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
  - managed directly by the operating system
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X
- **A relationship must exist between user threads and kernel threads.**

Multithreading Models

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

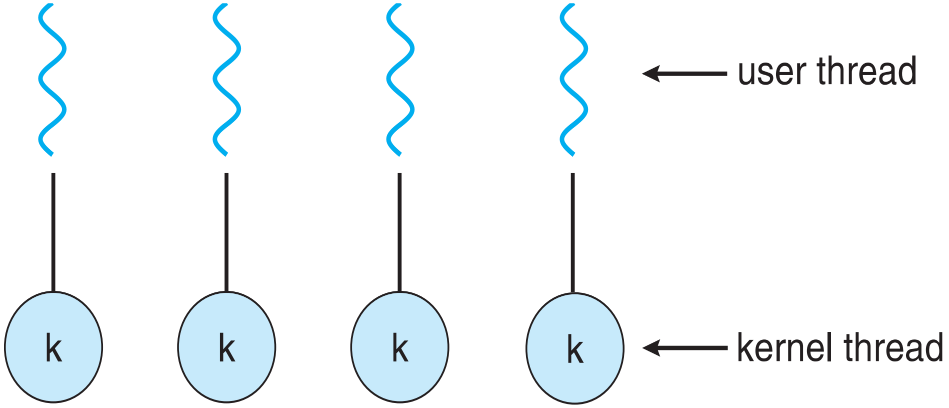
Many-to-One

- Many user-level threads mapped to single kernel thread
- Thread management is done by the thread library in user space, **so it is efficient**
- **One thread blocking causes all to block**
- **Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time**
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**
- very few systems continue to use the model because of its **inability to take advantage of multiple processing cores**



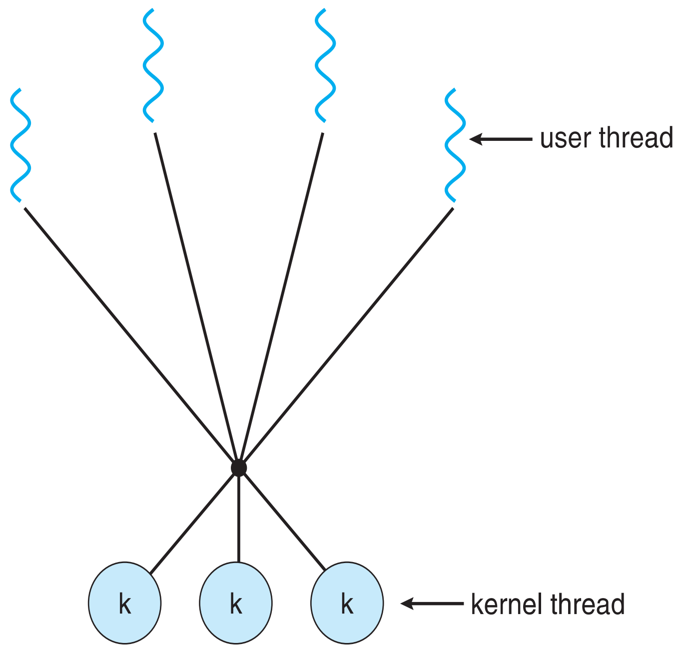
One-to-One

- Each user-level thread maps to a kernel thread
- Creating a user-level thread creates a kernel thread
- **More concurrency than many-to-one**: by allowing another thread to run when a thread makes a blocking system call
- It also allows multiple threads to run in parallel on multiprocessors
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads (a smaller or equal number of kernel threads)
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Effect of Multithreaded Models on Concurrency

- The many-to-one model allows the developer to create as many user threads as he/she wishes, it does not result in true concurrency, because **the kernel can schedule only one thread at a time.**
- The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application.
- The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

```
Microsoft Windows [Version 10.0.19044.1889]
(c) Microsoft Corporation. All rights reserved.

C:\Users\MAHE>wmic
wmic:root\cli>CPU Get NumberOfCores
NumberOfCores
4

wmic:root\cli>CPU Get NumberOfCores, NumberOfLogicalProcessors
NumberOfCores  NumberOfLogicalProcessors
4              8

wmic:root\cli>
```

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing a thread libraries
  - Library entirely in user space with no kernel support
    - All code and data structures for the library exist in user space.
    - This means that invoking a function in the library results in a local function call in user space and not a system call
  - Kernel-level library supported by the OS
    - Code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel
- Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java.

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- 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)

Windows, and Java Thread

- Windows thread library is a kernel-level library available on Windows systems.
- The Java thread API allows threads to be created and managed directly in Java programs.
- However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system.
- This means that on Windows systems, Java threads are typically implemented using the Windows API;
- UNIX and Linux systems often use Pthreads.

Threading Issues

- Some of the issues to consider in designing multithreaded programs
- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- The semantics of the fork() and exec() system calls change in a multithreaded program
- Issue
  - If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?
    - Does **`fork()`** duplicate only the calling thread or all threads?
- Solution
  - Some UNIX systems have chosen to have two versions of fork(),
    - one that duplicates all threads and
    - another that duplicates only the thread that invoked the fork() system call.
- **`But which version of fork () to use and when?`**

Semantics of fork() and exec()

- If a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.
- Depends on the application
  - If exec() is called immediately after forking,
    - Then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate.
  - If the separate process does not call exec() after forking,
    - Then separate process should duplicate all threads.

Signal Handling

Signals are used in UNIX systems **to notify a process that a particular event has occurred.**

- A signal may be received either **synchronously or asynchronously** depending on the **source** of and the **reason for the event** being signaled.
- All signals, whether synchronous or asynchronous, follow the same pattern:
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Once delivered, the signal must be handled
  4. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined

Signal Handling (Cont.)

Every signal has **default handler** that kernel runs when handling that signal

- **User-defined signal handler** can override default action

- Signals are handled in different ways.
  - Some signals (such as changing the size of a window) are simply **ignored**; others (such as an illegal memory access) are handled by **terminating the program**.
  - Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?
  - 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

Signal Handling (Cont.)

- The **method for delivering** a signal depends on the **type of signal generated.**
  - Synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process.
  - Asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (<control><C>, for example)—should be sent to all threads.
- The standard UNIX function for delivering a signal is
  - kill(pid_t pid, int signal)
  - Pthread_kill(pthread_t tid, int signal)

Thread Cancellation

- Terminating a thread before it has finished
  - For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
  - Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are canceled.
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation**: One thread terminates the target thread immediately
  - **Deferred cancellation:** allows the target thread to periodically check if it should be cancelled

Thread Cancellation

- The difficulty with cancellation occurs in situations where **resources have been allocated to a canceled thread** or where a thread is canceled while **in the midst of updating data it is sharing with other threads**. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously **may not free a necessary system-wide resource**.
- With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely.
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests, cancellation, but actual cancellation depends on thread state
- Pthreads supports three cancellation modes. Each mode is defined as a state and a type

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

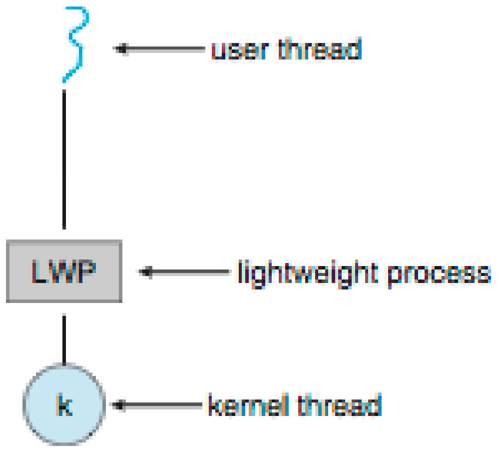
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - i.e. **`pthread_testcancel()`**
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

- **Thread-local storage** (**TLS**) 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)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads



Operating System Examples

- Windows Threads
- Linux 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)
  - Flags control behavior

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.

- **`struct task_struct`** points to process data structures (shared or unique)