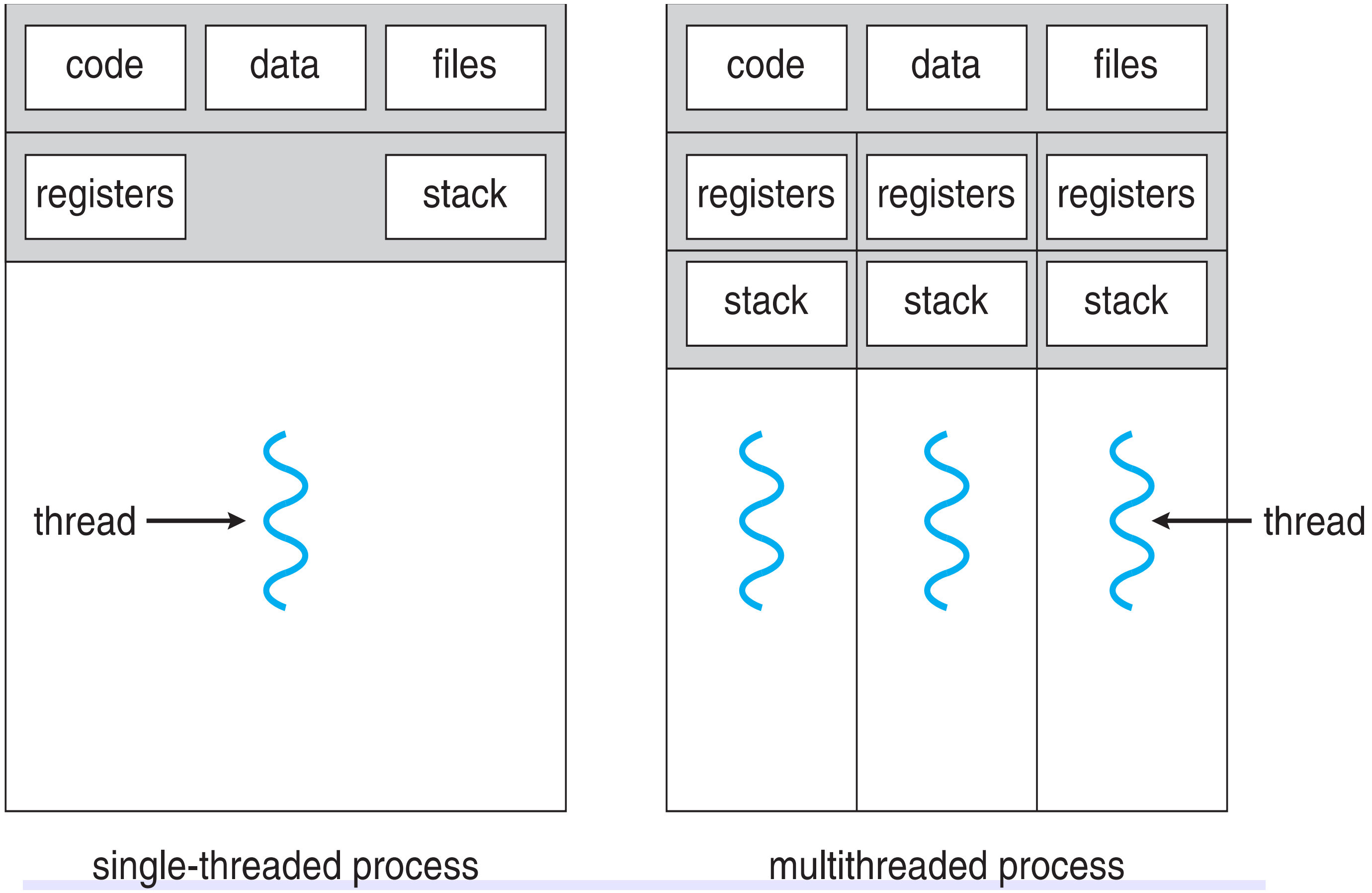
Threads

- Thread concept
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues

Basics

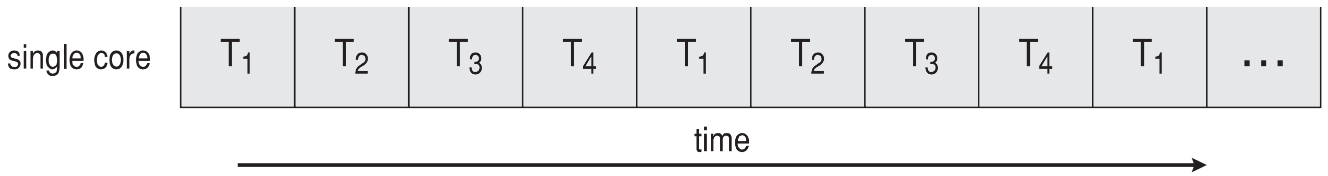
Each process has to get access to system resources necessary to perform its mission. To accomplish the mission the process can use one or more threads of execution. The threads share access to process-available resources, but can also have privately allocated resources.



Benefits of multi-threading

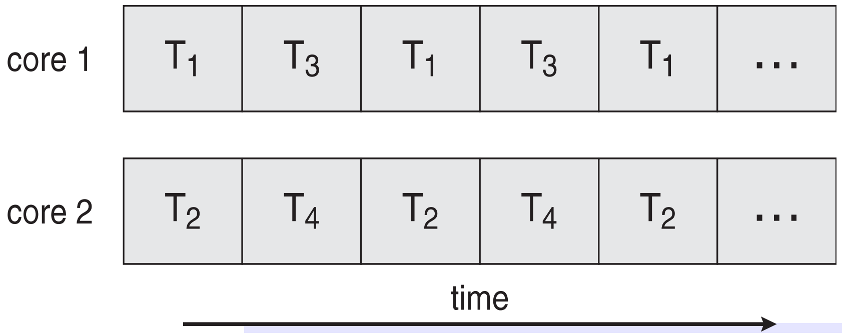
- **Responsiveness** – may allow continued execution if part of process is **blocked**; especially important for user interfaces
- **Process resource sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures to perform parallel thread activities **concurrently**

Concurrent execution on single-core system:



Concurrency supports more than one task making progress

Parallelism on a multi-core system:



Parallelism implies a system can perform more than one task **simultaneously**

Amdahl’s law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

Multicore programming

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading. E.g. CPUs have cores as well as **hardware threads** (Oracle SPARC T4: 8 cores, and 8 hardware threads per core)
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library. Examples:
  - POSIX **threads (Pthreads)**
  - Windows threads
  - Java threads

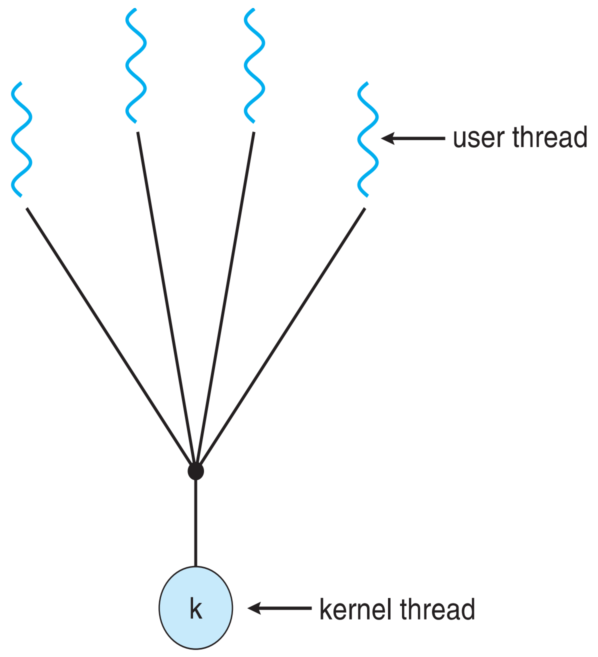
- **Kernel threads** - virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

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

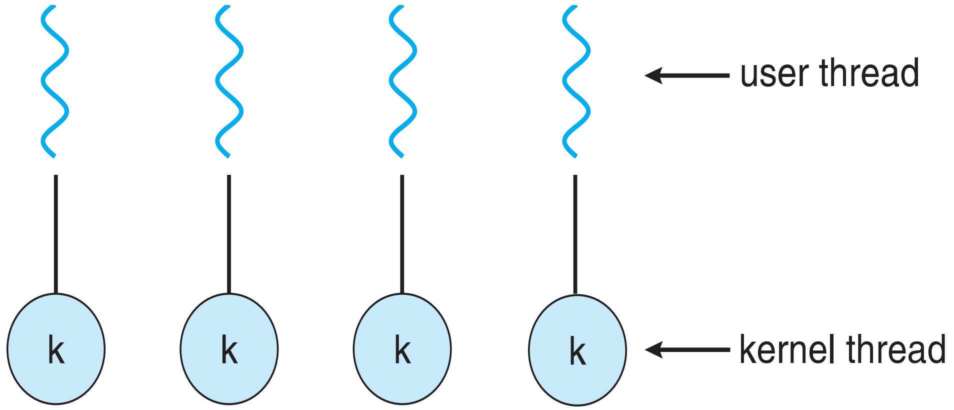
Many-to-One

- Many user-level threads mapped to single kernel thread
- 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**



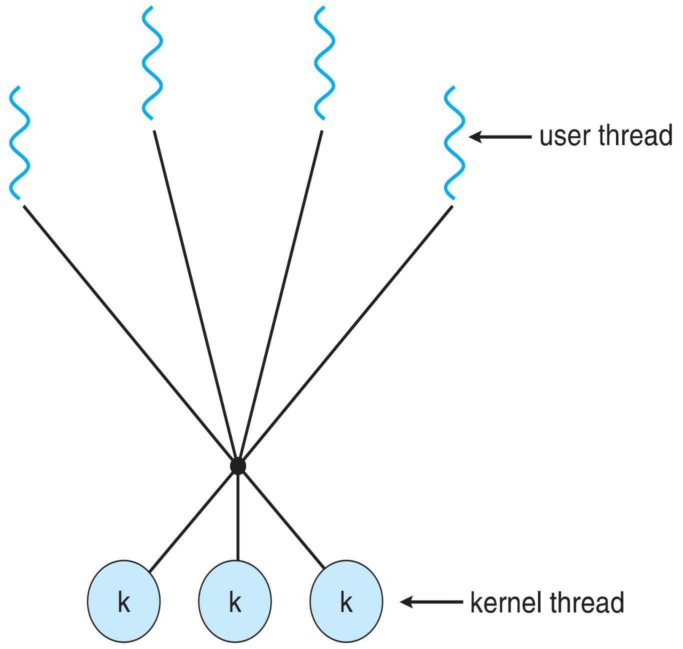
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



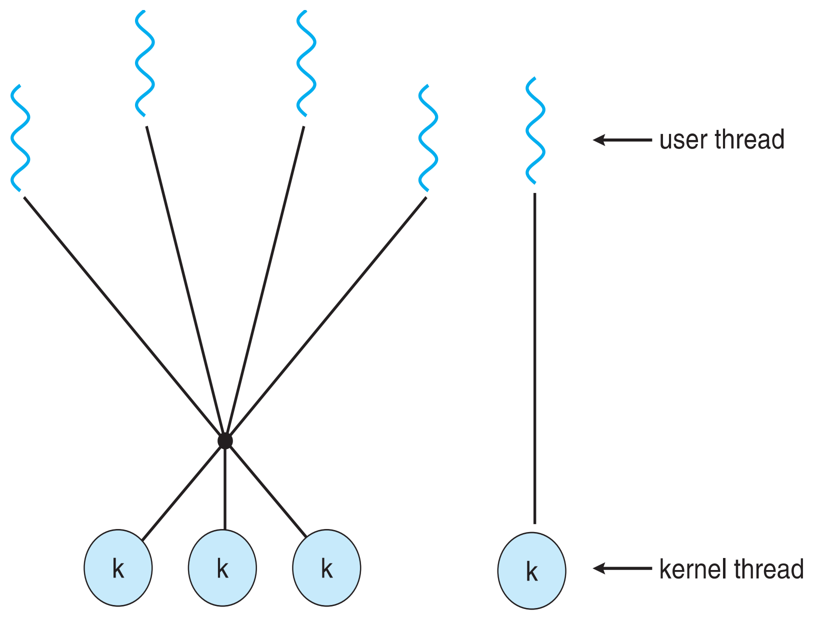
Many-to-Many Model

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



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing thread library
  - Library entirely in user space
  - Kernel-level library supported by the OS

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)

Pthreads - example

```
#include <pthread.h>
#include <stdio.h>
int sum; // shared variable
void *runner(void *param) {// worker function
int i, upper = *((int *)(param));
   sum = 0;
   if (upper > 0) {
        for (i = 1; i <= upper; i++) sum += i;
   }
   pthread_exit(0);
}
int main(int argc, char *argv[]){
pthread_t tid;   // thread identifier (TID)
pthread_attr_t attr; // thread attributes
int ret, Param;
   if (argc != 2) {
        fprintf(stderr,"Usage: %s arg",argv[0]); exit(1);
   }
   Param=atoi(argv[1]);
   pthread_attr_init(&attr); // setting (def.) thread attributes
   if((ret=pthread_create(&tid,&attr,runner,&Param)!=0){//create thread
        perror("create error"); exit(3);
   }
   if((ret=pthread_join(tid,NULL))!=0){ // wait for thread termination
        perror("join"); exit(4);
   }
   printf("sum = %d\n",sum); return 0;
}
```

Microsoft Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

MsWin threads - example

```
#include <stdio.h>
#include <windows.h>
DWORD Sum; // shared variable
DWORD WINAPI Summation(PVOID Param) {// worker function
   DWORD Upper = *(DWORD *)Param;
   for (DWORD i = 0; i <= Upper; i++)      Sum += i;
   return 0;
}
int main(int argc, char *argv[]){
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   if (argc != 2) {// Argument control
        fprintf(stderr, "Integer argument required\n"); return -1;
   }
   Param = atoi(argv[1]);

   // thread creation
   ThreadHandle = CreateThread(NULL, 0, Summation, &Param, 0, &ThreadId);
   if (ThreadHandle != NULL) {    // waiting for
        WaitForSingleObject(ThreadHandle, INFINITE);// thread termination
        CloseHandle(ThreadHandle);
        printf("sum = %d\n",Sum);
   }
   return 0;
}
```

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Example implicit threading solutions
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch (Mac OS X and iOS operating systems)
  - Microsoft Threading Building Blocks (TBB),
  - **`java.util.concurrent`** package

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
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.Tasks could be scheduled to run periodically
- Windows API supports thread pools.

OpenMP

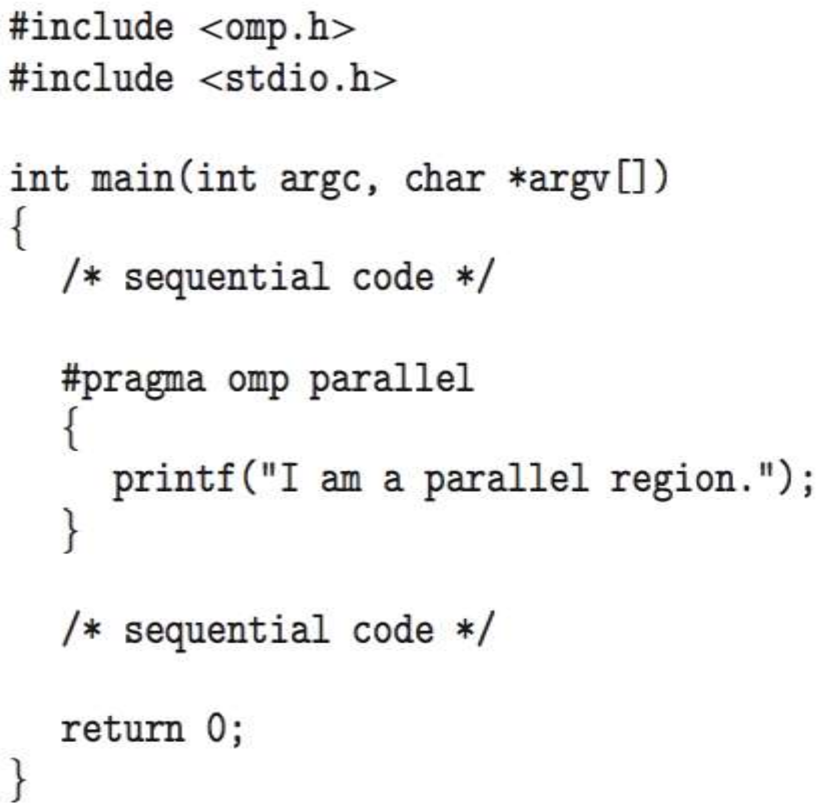
- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
  for(i=0;i<N;i++) {
     c[i] = a[i] + b[i];
}
```

Run for loop in parallel



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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

Threading Issues

- 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()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
  - POSIX requires that only the thread which called **`fork()`** is duplicated
- **`exec()`** usually works as normal – replace the running process including all threads

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 by one of two signal handlers:
     a) default
     b) user-defined
- Every signal has **default handler** that kernel runs when handling signal

  User-defined signal handler can override default
  - For single-threaded, signal delivered to process

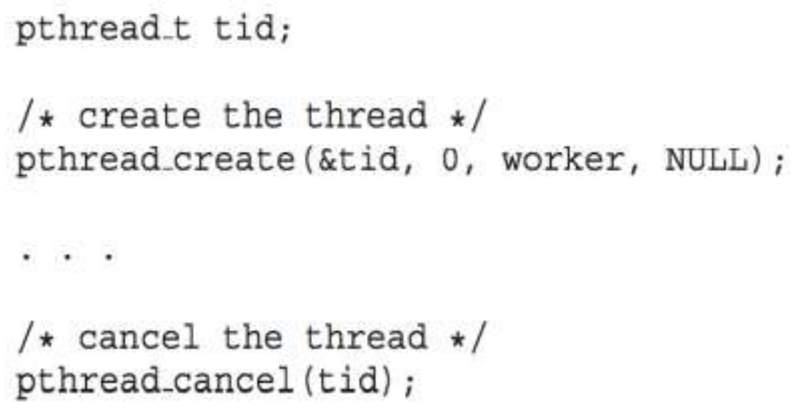
Signal Handling (Cont.)

Where should a signal be delivered for multi-threaded?

- 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

Thread Cancellation

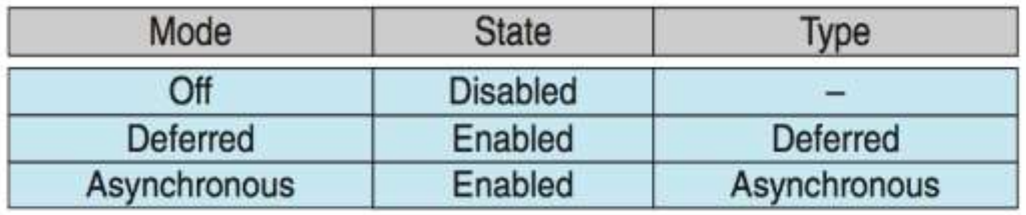
- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread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



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

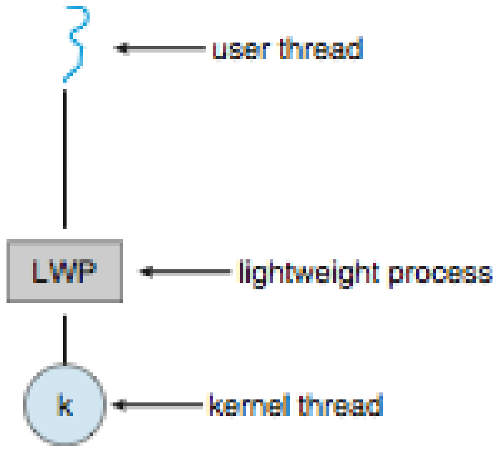
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - i.e. pthread_testcancel()
    - then cleanup handler is invoked

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 of kernel threads


user thread
LWP
lightweight process
k
kernel thread