Topics

Topics

- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

Objectives

- To introduce the notion of a thread
    - A fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads and Windows thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux
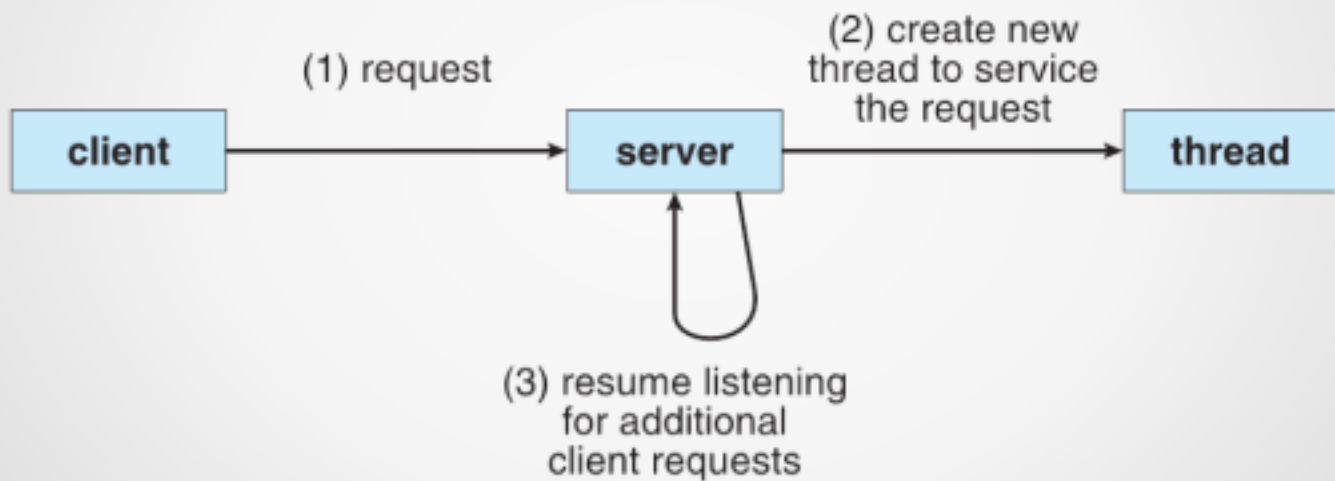- Discuss How this Ties into FreeRTOS tasks

Motivation

Motivation

- Most modern applications are multithreaded

- Threads run within application

- Multiple tasks with the application can be implemented by separate threads

  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

- Process creation is heavy-weight while thread creation is light-weight

- Can simplify code, increase efficiency

- Kernels are generally multithreaded

Multithreaded Server Architecture

Multithreaded Server Architecture

client → (1) request → server → (2) create new thread to service the request → thread

server → (3) resume listening for additional client requests

Benefits

Benefits

- **Responsiveness**
- **Resource Sharing**
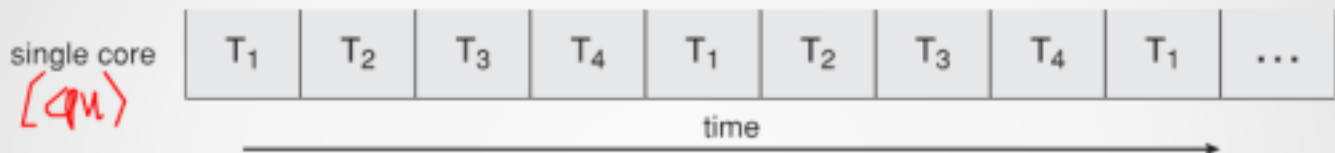- **Economy**
- **Scalability**

Multicore Programming

Multicore Programming

- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
    - Dividing activities
    - Balance
    - Data splitting
    - Data dependency
    - Testing and debugging
- *Parallelism* implies a system can perform more than one task simultaneously
- *Concurrency* supports more than one task making progress
    - Single processor / core, scheduler providing concurrency
- Types of parallelism

    PbM EXEC

    - 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
    - CPUs have cores as well as *hardware threads*
    - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Concurrency vs. Parallelism

Single and Multithreaded Processes

Single and Multithreaded Processes

MEMORY

| code | data | files |

| registers | | stack |

PC, SP, GPRs

thread →

single-threaded process

| code | data | files |

| registers | registers | registers |

| stack | stack | stack |

thread

multithreaded process
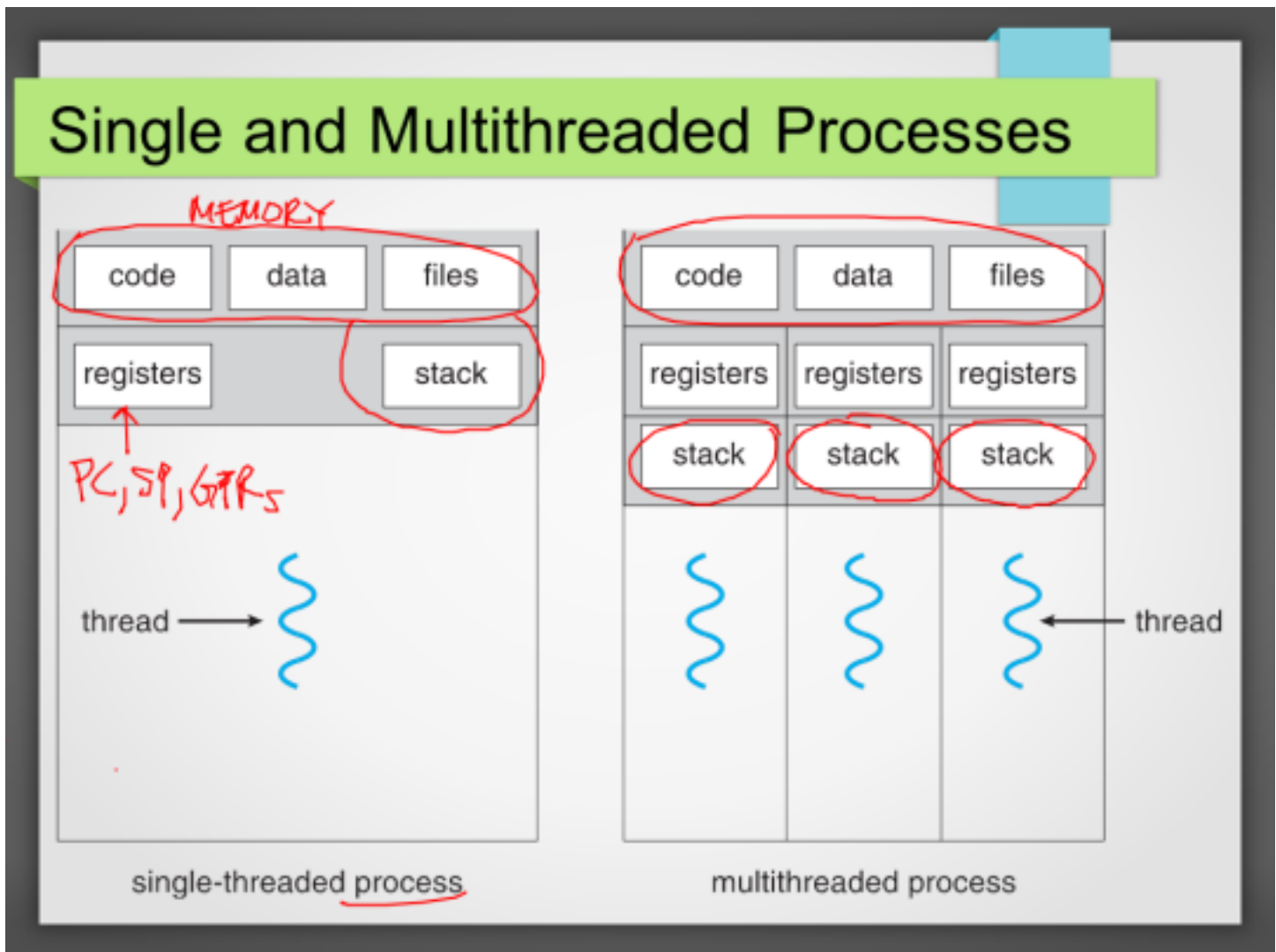
Amdahl??s Law

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

- I.e. 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

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

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

User Threads and Kernel Threads

User Threads and Kernel Threads

- User threads - management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- Kernel threads - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
 - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

Multithreading Models

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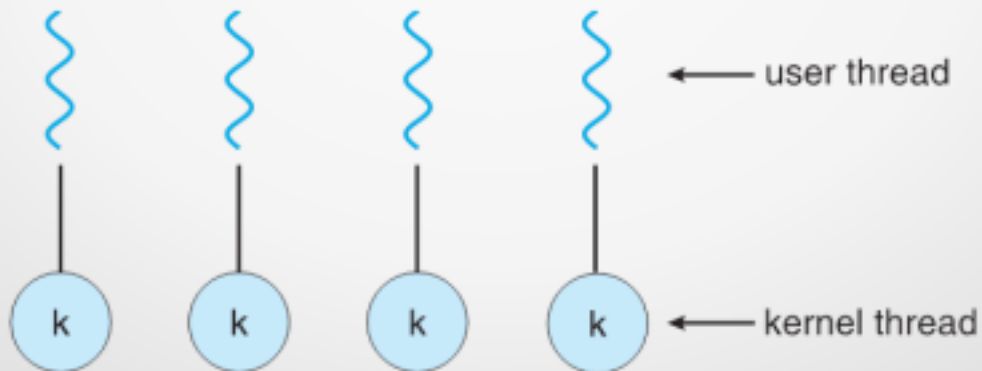- Many-to-Many

Many-to-One

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

user thread
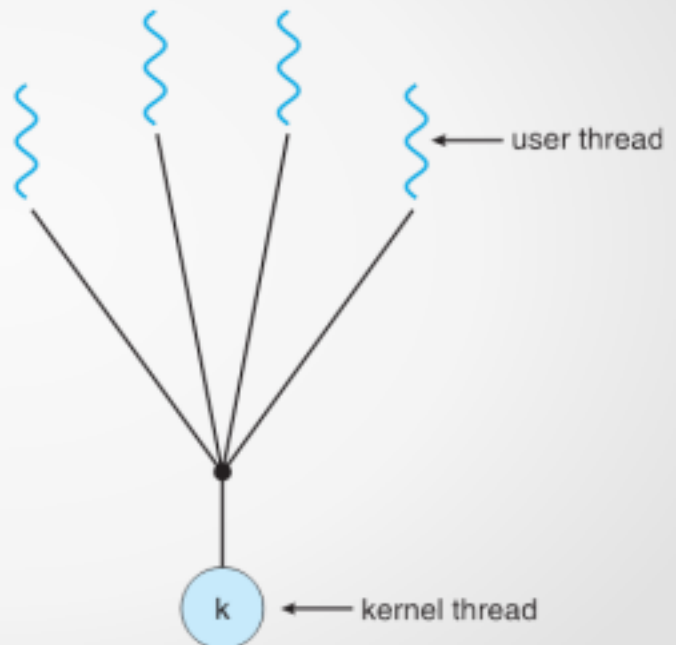
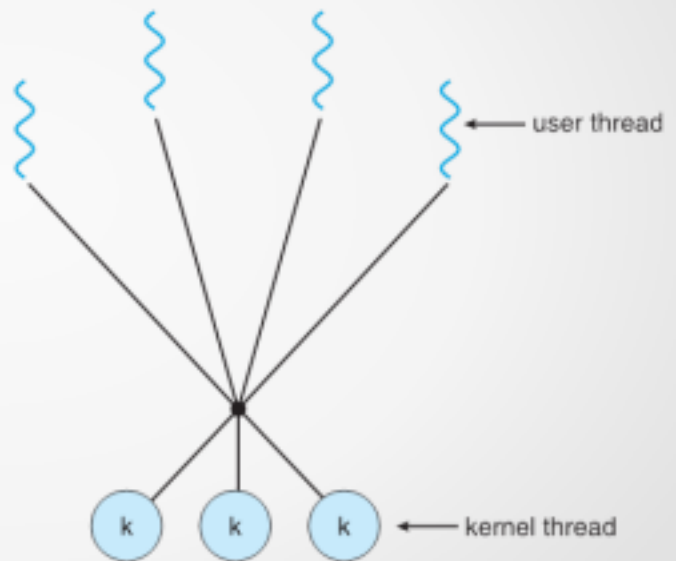kernel thread

One-to-One

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 NT/XP/2000

  - Linux

  - Solaris 9 and later

← user thread

k k k k ← kernel thread
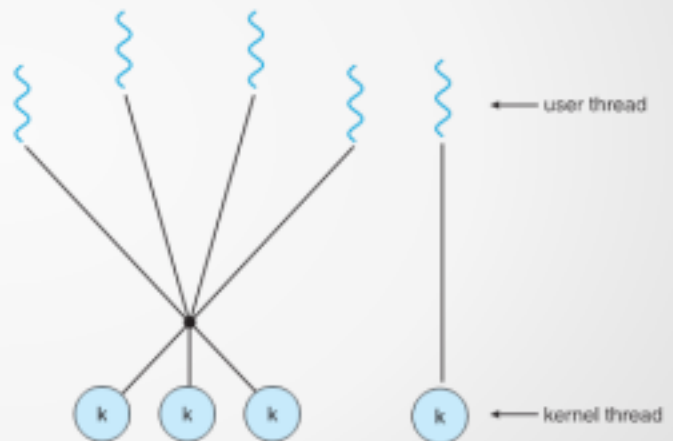
Many-to-Many Model

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

user thread

k   k   k   ← kernel thread

Two-level Model

Thread Libraries

Thread Libraries

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

Pthreads

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

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```

Pthreads Example (Cont.)

Pthreads Example (Cont.)

```
        /* get the default attributes */
        pthread_attr_init(&attr);
        /* create the thread */
        pthread_create(&tid,&attr,runner,argv[1]);
        /* wait for the thread to exit */
        pthread_join(tid,NULL);

        printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Figure 4.9   Multithreaded C program using the Pthreads API.

Pthreads Code for Joining 10 Threads

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.

Win32 API Multithreaded C Program

Win32 API Multithreaded C Program

```c
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   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) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
}
```

Win32 API Multithreaded C Program (Cont.)

Win32 API Multithreaded C Program (Cont.)

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

if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
}
}
```

Implicit Threading

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

- Three methods explored

  - Thread Pools
  - OpenMP
  - Grand Central Dispatch

- Other methods include Microsoft Threading Building Blocks (TBB), java.util.concurrent package

Thread Pools

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:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
     * this function runs as a separate thread.
     */
}
```

OpenMP

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;
}
```

Grand Central Dispatch

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in "^{ }" - `^{ printf("I am a block"); }`
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called main queue
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Threading Issues

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()

Semantics of fork() and exec()

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

Signal Handling

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

  - Signal is generated by particular event

  - Signal is delivered to a process

  - Signal is handled by one of two signal handlers:

    - default
    - 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

- 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

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.)

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

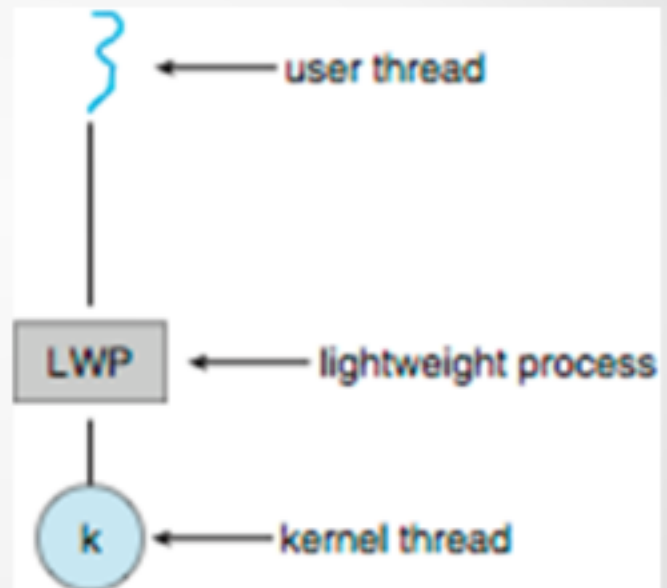- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

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

Operating System Examples

# Operating System Examples

- Windows XP Threads
- Linux Thread

Windows Threads

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

- The primary data structures of a thread include:

  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows XP Threads Data Structures
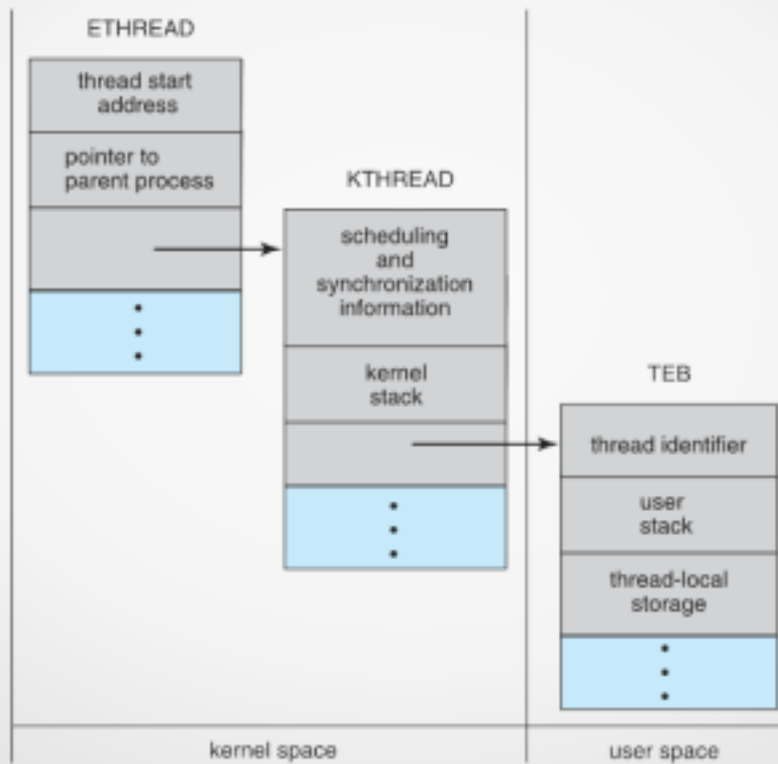
Linux Threads

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)