CENG 2034 - Operating Systems

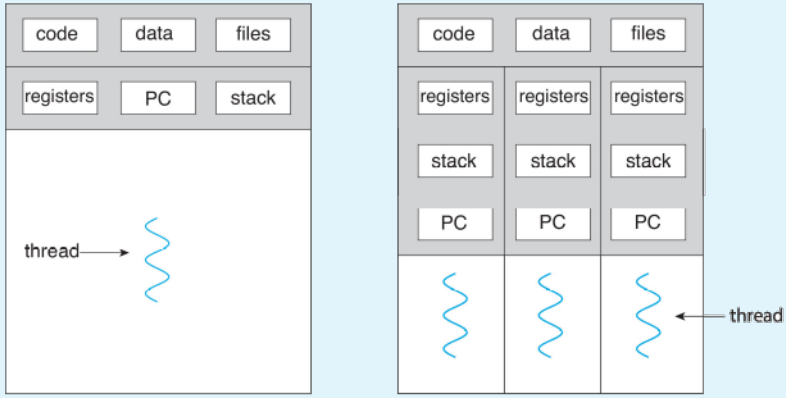
Week 6: Threads & Concurrency

Burak Ekici

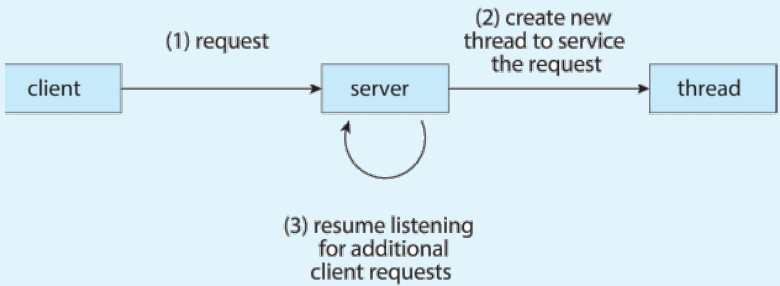
April 14, 2023

Outline

Single and Multithreaded Processes



Multithreaded Server Architecture



Benefits

- Responsiveness – may allow continued execution if part of process is blocked, especially important for user interfaces

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

- 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
- Economy – cheaper than process creation, thread switching lower overhead than context switching

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

Multicore Programming

- Multicore or multiprocessor systems puts pressure on programmers, challenges include:

Multicore Programming

- Multicore or multiprocessor systems puts pressure on programmers, challenges include:
  - Dividing activities

Multicore Programming

- Multicore or multiprocessor systems puts pressure on programmers, challenges include:
  - Dividing activities
  - Balance

Multicore Programming

- Multicore or multiprocessor systems puts pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting

Multicore Programming

- Multicore or multiprocessor systems puts pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency

Multicore Programming

- Multicore or multiprocessor systems puts pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging

Multicore Programming

- Multicore or multiprocessor systems puts 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

Multicore Programming

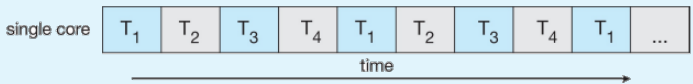
- Multicore or multiprocessor systems puts pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress

Multicore Programming

- Multicore or multiprocessor systems puts 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

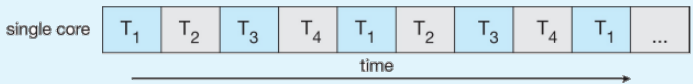
Concurrency vs. Parallelism

- Concurrent execution on single-core system:

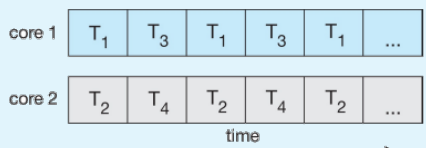


Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



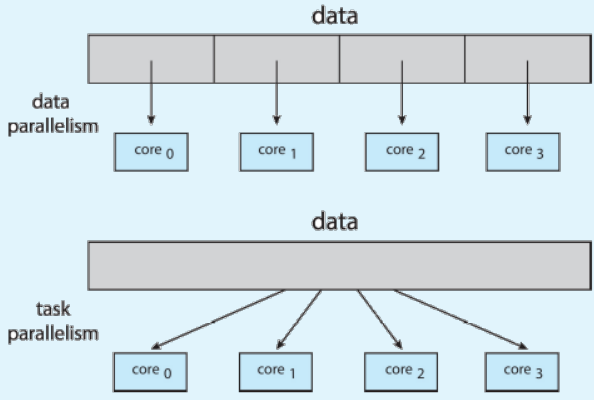
Parallelism

- Data parallelism – distributes subsets of the same data across multiple cores, same operation on each

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

Data and Task Parallelism



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components

Amdahl's Law

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

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

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

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

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

- If application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times

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

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

- 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

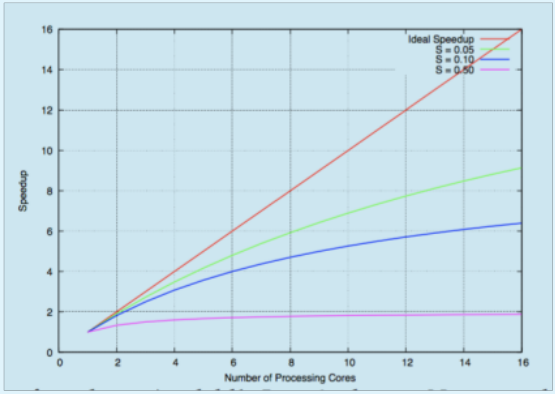
Amdahl's Law

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

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

- 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 has disproportionate effect on performance gained by adding additional cores

Amdahl's Law

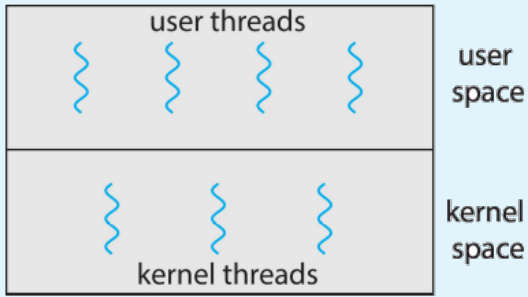

Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

User Threads and Kernel Threads

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

Outline

User Threads and Kernel Threads



Multithreading Models

- Many-to-One

Multithreading Models

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

Multithreading Models

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

Many-to-One

- Many user-level threads mapped to single kernel thread

Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block

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 multicore system because only one may be in kernel at a time

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 multicore system because only one may be in kernel at a time
- Few systems currently use this model

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 multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:

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 multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads

Many-to-One

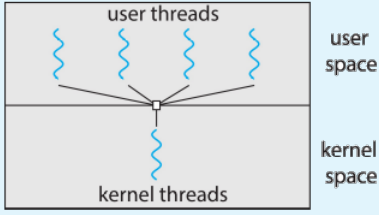
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

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 multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

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 multicore 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

One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread

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

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

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:

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

One-to-One

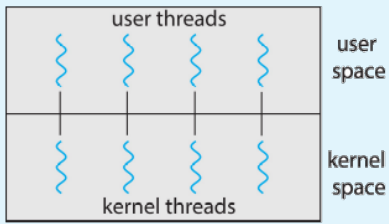
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples:
  - Windows
  - Linux

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

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



Many-to-Many

- Allows many user level threads to be mapped to many kernel threads

Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads

Many-to-Many

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

Many-to-Many

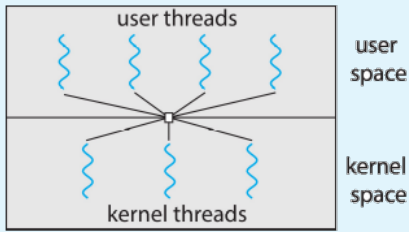
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common

Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common

Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common

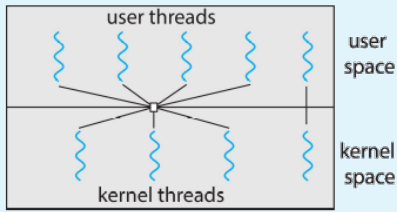


Two-level Model

Similar to M:M, except that it allows a user thread to be bound to kernel thread

Two-level Model

Similar to M:M, except that it allows a user thread to be bound to kernel thread



Outline

Thread Libraries

- Thread library provides programmer with API for creating and managing threads

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space

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

- May be provided either as user-level or kernel-level

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

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

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

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 (Linux & Mac OS X)

Example (Pthreads – Producer-Consumer)

```
pthread_t prod, cons;

pthread_create(&prod, NULL, (void *) addCA, p);
pthread_create(&cons, NULL, (void *) dropCA, p);

pthread_join(prod, NULL);
pthread_join(cons, NULL);
```

Example (Pthreads – Producer-Consumer (cont'd))

```
void addCA(CA *a)
{
    int next;
    while (true)
    {
        sem_wait(&(*(a)).empty);
        next = (rand() % 8) + 1;
        (*(a)).buffer[(*(a)).in] = next;
        (*(a)).in = ((*(a)).in + 1) % size;
        (*(a)).counter++;
        printCA('p', a);
        sem_post(&(*(a)).full);
    }

}
```

```
void dropCA(CA *a)
{
    int prev;

    while (true)
    {
        sleep(1);
        sem_wait(&(*(a)).full);
        prev = (*(a)).buffer[(*(a)).out];
        (*(a)).buffer[(*(a)).out] = 0;
        (*(a)).out = ((*(a)).out + 1) % size
     ;
        (*(a)).counter--;
        printCA('c', a);
        sem_post(&(*(a)).empty);
    }

}
```

Outline

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads

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

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
- Some methods:

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
- Some methods:
  - Thread Pools

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
- Some methods:
  - Thread Pools
  - Fork-Join

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
- Some methods:
  - Thread Pools
  - Fork-Join
  - OpenMP

Thread Pools

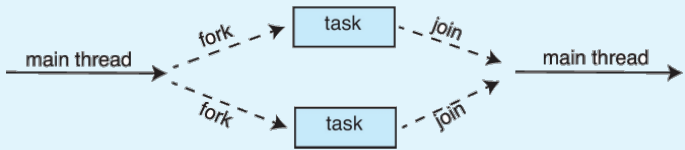
- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task – i.e,Tasks could be scheduled to run periodically

Fork-Join Parallelism

Multiple threads (tasks) are forked, and then joined.

Fork-Join Parallelism

Multiple threads (tasks) are forked, and then joined.



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

```
#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;
}
```

OpenMP

Run the for loop in parallel

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

Outline

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
- `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: 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

Signal Handling (cont'd)

Where should a signal be delivered for multi-threaded?

Signal Handling (cont'd)

Where should a signal be delivered for multi-threaded?

- Deliver the signal to the thread to which the signal applies

Signal Handling (cont'd)

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

Signal Handling (cont'd)

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

Signal Handling (cont'd)

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 Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is target thread

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is target thread
- Two general approaches:

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

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

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);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Thread Cancellation (cont'd)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

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

Thread Cancellation (cont'd)

- 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

Thread Cancellation (cont'd)

- 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

Thread Cancellation (cont'd)

- 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 Cancellation (cont'd)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

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

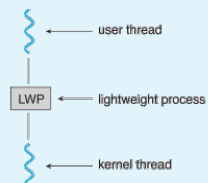
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    i.e., pthread_testcancel()
    Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled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

Thread-Local Storage

- Thread-local storage (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



Outline

Windows Threads

- Windows API – primary API for Windows applications
- 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

Windows Threads (cont'd)

The primary data structures of a thread include:

Windows Threads (cont'd)

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

Windows Threads (cont'd)

The primary data structures of a thread include:

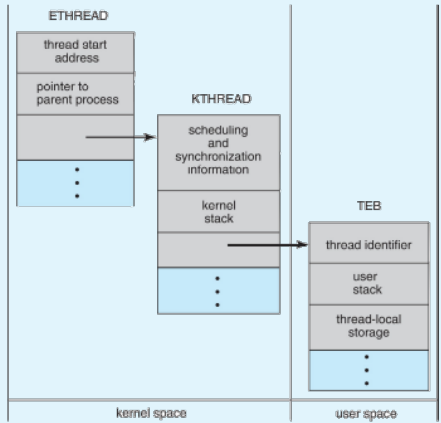
- ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
- KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space

Windows Threads (cont'd)

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 Threads Data Structures



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)

Thanks! & Questions?