

Processes vs Threads

Didem Unat

Lecture 13

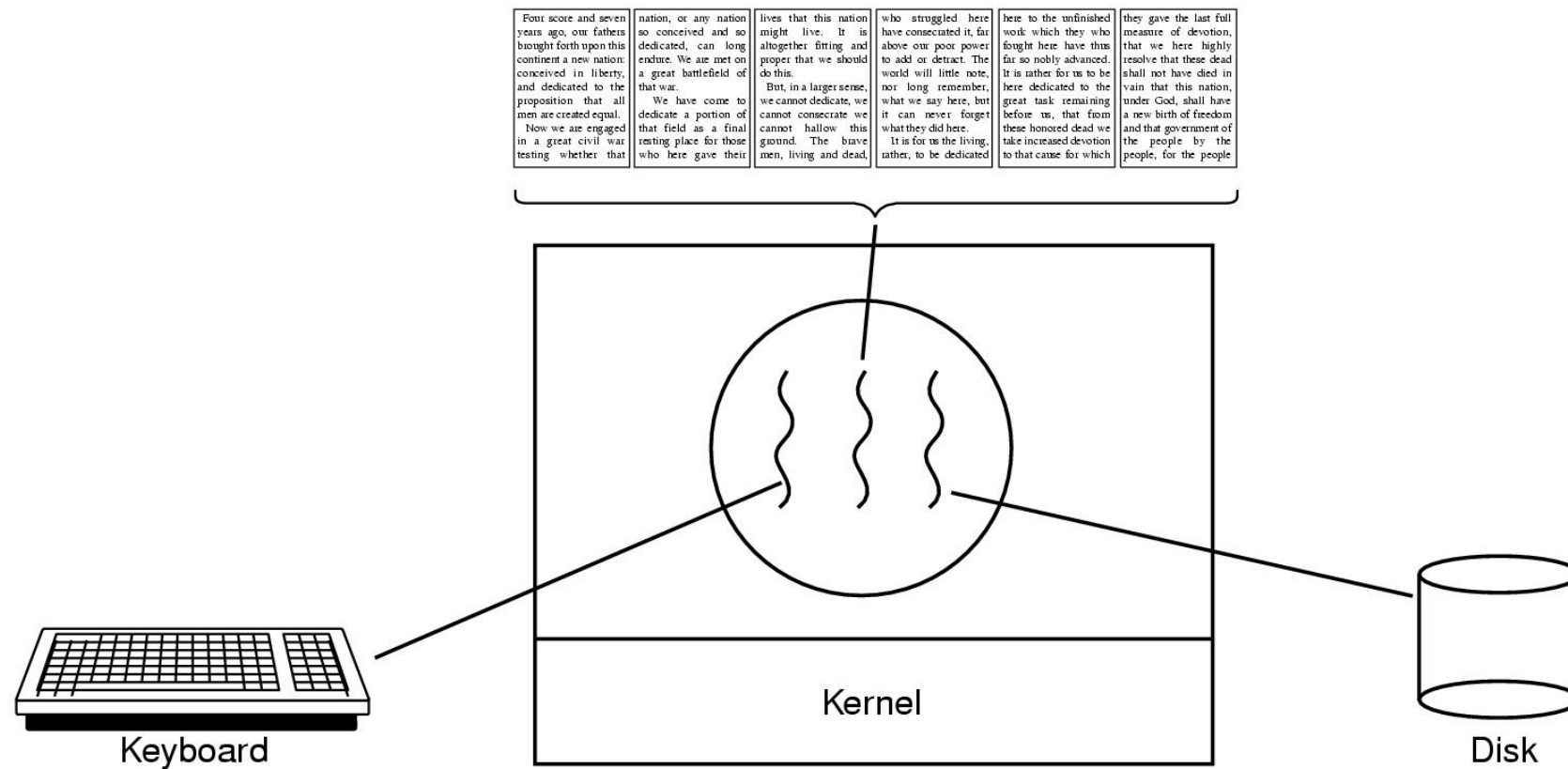
COMP304 - Operating Systems (OS)

Threads \leq Processes

- A process is an executing program with at least one thread of control
- A process can contain multiple threads of control
 - Threads run within application in parallel (concurrently)
- Process creation is relatively **heavy-weight** while thread creation is **light-weight**
- Modern kernels are generally multithreaded
- Reading from textbook
 - Chapter 4.1, 4.3, 4.6 , 4.7

Thread Usage - Example

- An editor process with three threads

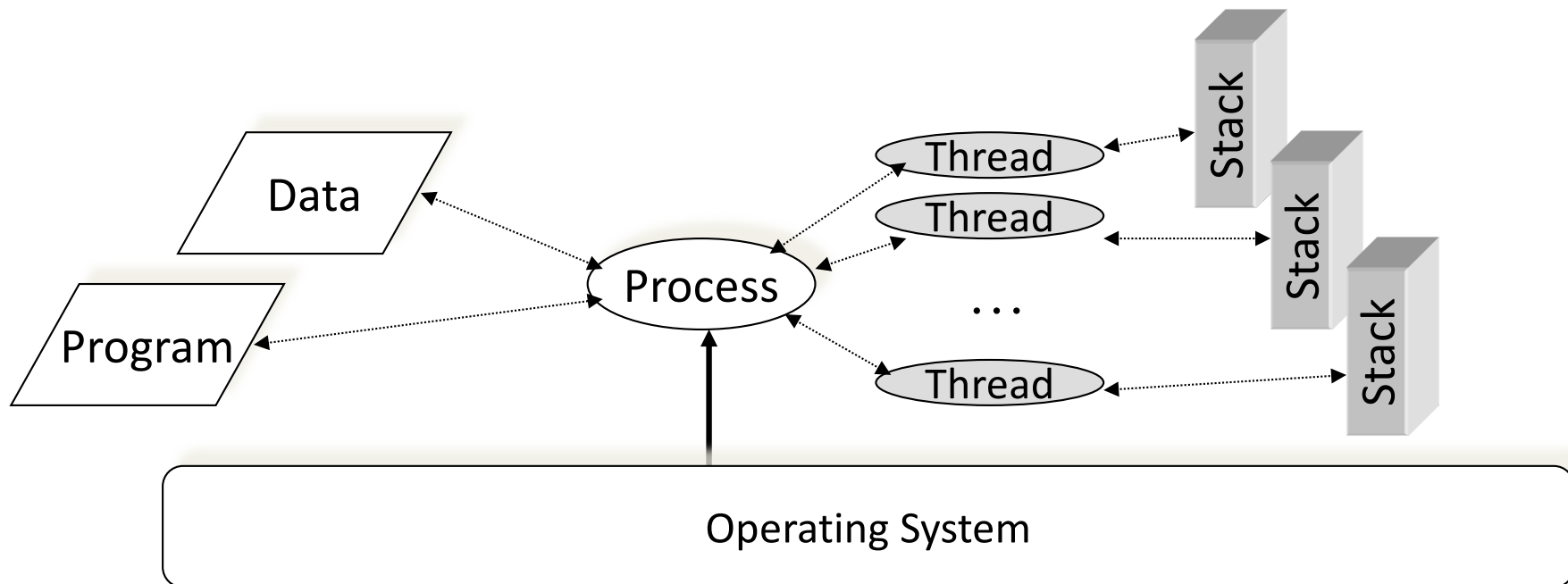


Why use threads?

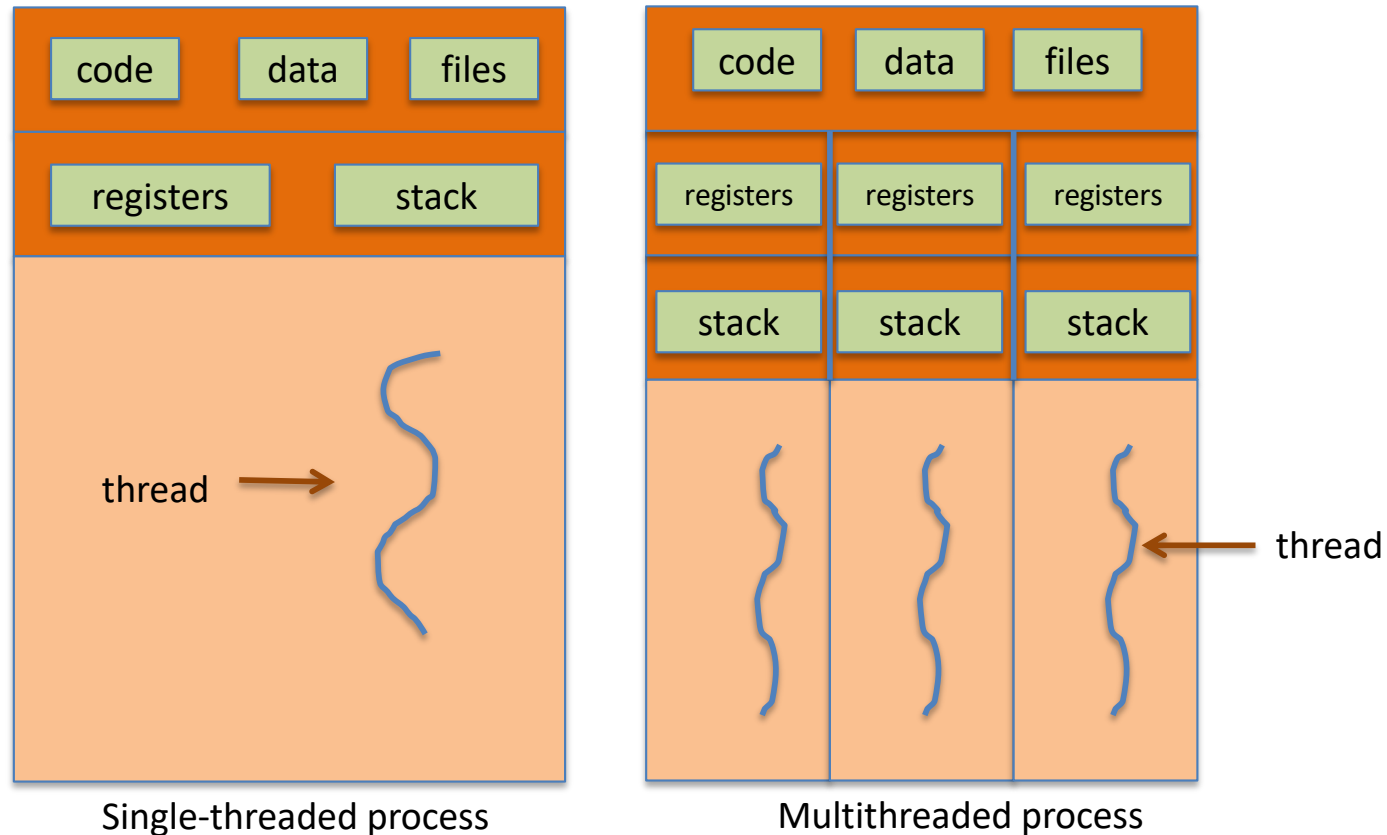
- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interactivity
- **Resource Sharing** – threads share resources of a process, easier than shared memory or message passing communication between processes
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching in some systems

Process and Thread

- **Process** is an infrastructure in which execution takes place – address space + resources
- **Thread** is a program in execution within a process context – each thread has its own stack



Single vs Multithreaded Process



- A thread has an ID, a program counter, a register set, and a stack
- Shares the code section, data section and OS resources (e.g. files) with other threads within the same process

Thread States

- Like a process, a thread can be in one of 3 states: ready, blocked (waiting), running
- A thread may wait:
 - For some external event to occur (such as I/O completion)
 - For some other thread to unblock it.
- When a program runs, it starts as a single-threaded process
 - Then it can create other threads
 - Typically the first thread is referred as the **master** thread and the others are **worker** threads

Thread Libraries

- **A thread library** provides a 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

User Threads and Kernel Threads

- **User threads** - management done by user-level thread 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

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the thread model provided by underlying OS
- Java threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

Java Multithreaded Program

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}
```

Creates java threads

class Summation implements Runnable

```
{
    private int upper;
    private Sum sumValue;

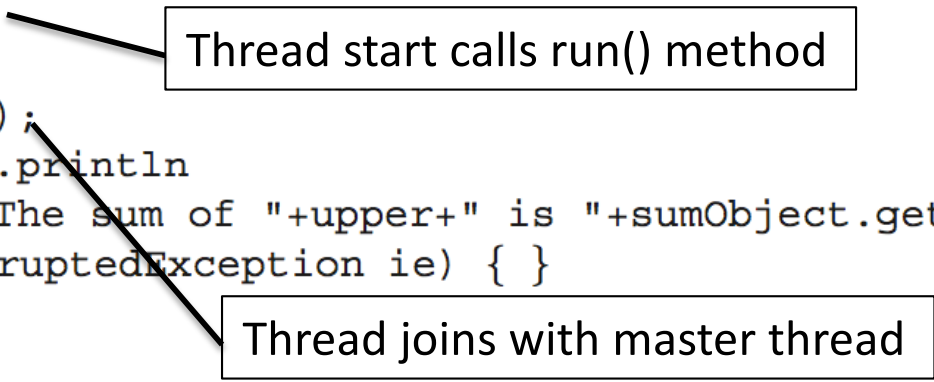
    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```

This method is executed by a thread when it is created

Java Multithreaded Program (cont.)

```
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>"); }
}
```



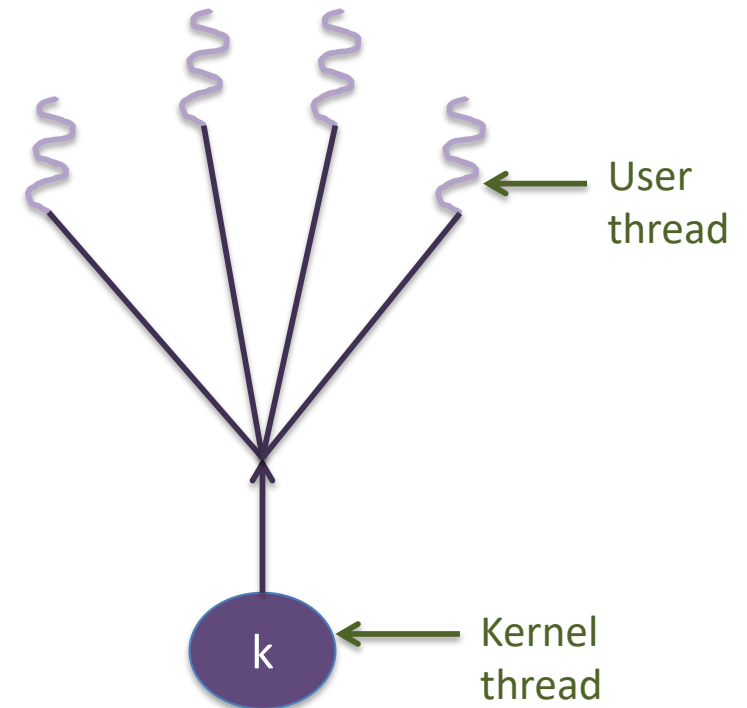
- How can threads share data?

Multi-Threading Models

- Many systems support both user and kernel level threads, resulting in different multi-threading models
 - Many-to-one
 - One-to-one
 - Many-to-many

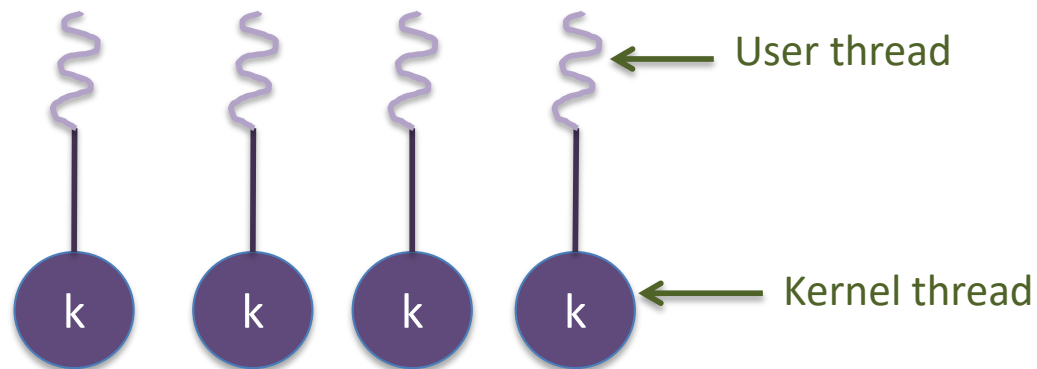
Many-to-One

- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads
 - Not common anymore
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Entire process will block if a thread makes a blocking system call



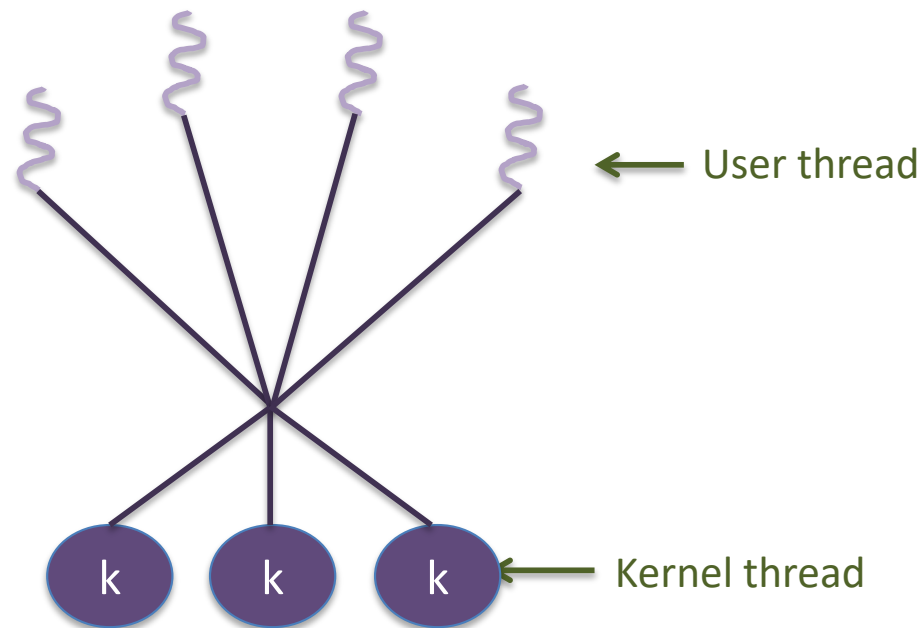
One-to-One

- Each user-level thread maps to a kernel thread
 - Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows NT/XP/2000, Linux, Solaris 9 and later



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads (usually fewer 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



Exercise

- Which of the following components of program state are shared across threads?
 - Register values
 - Heap memory
 - Stack
 - Global variables
 - Program counter
 - Scheduling properties (e.g. policy, priority)

Question

- A system with two dual-core processors has four processors available for scheduling.
 - A CPU-intensive application is running on this system.
 - All input is performed at program start-up, when a single file must be opened.
 - Similarly, all output is performed just before the program terminates, when the program results must be written to a single file.
 - Between startup and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it.
 - The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).
- How many threads will you create to perform the input and output? Explain.
- How many threads will you create for the CPU-intensive portion of the application? Explain.

Answer

- It only makes sense to create as many threads as there are blocking system calls, as the threads will be blocking. Creating additional threads provides no benefit. Thus, it makes sense to create a single thread for input and a single thread for output unless you can also perform parallel I/O
- Four. There should be as many threads as there are processing cores. Fewer would be a waste of processing resources, and any number > 4 would be unable to run simultaneously.

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 system calls change in a multithreaded program
- Threads and fork(): **think twice before mixing them.**
- Does **fork()** duplicate only the calling thread or all threads?
 - Some UNIXes have two versions of fork
 - Version 1: The child has only one thread
 - Version 2: The child has all the threads
- **Exec()** usually works as normal – replace the entire process including all threads
 - Call exec right after fork()

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
 - Synchronous : illegal memory access, division by zero
 - Asynchronous: key strokes to cancel a process
- In a single-threaded program, a signal is delivered to the process
- In a multi-threaded program, where 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
- Depends on the type of signal generated.

Thread Cancellation

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

OS Example: Windows Threads

- 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

OS Example: 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)
 - Different than fork()

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 thread data structures

Question

- Linux does not distinguish between processes and threads. However, many operating systems—such as Windows XP and Solaris—treat processes and threads differently. Typically, such systems use a notation wherein the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.
- Answer:
- On one hand, in systems where processes and threads are considered as similar entities, some of the operating system code could be simplified. A scheduler, for instance, can consider the different processes and threads on an equal footing without requiring special code to examine the threads associated with a process during every scheduling step. On the other hand, this uniformity could make it harder to impose process-wide resource constraints in a direct manner. Instead, some extra complexity is required to identify which threads correspond to which process and perform the relevant accounting tasks.

Reading

- From text book
 - Read Chapter 4.1, 4.3, 4.6 , 4.7
- Next lecture on POSIX Threads
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley