

# Threads

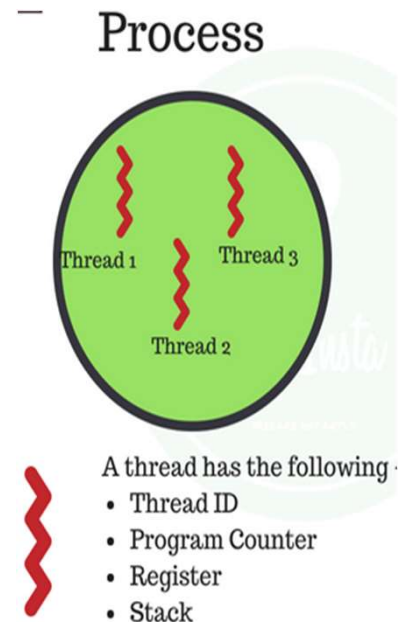
---

LECTURE 4



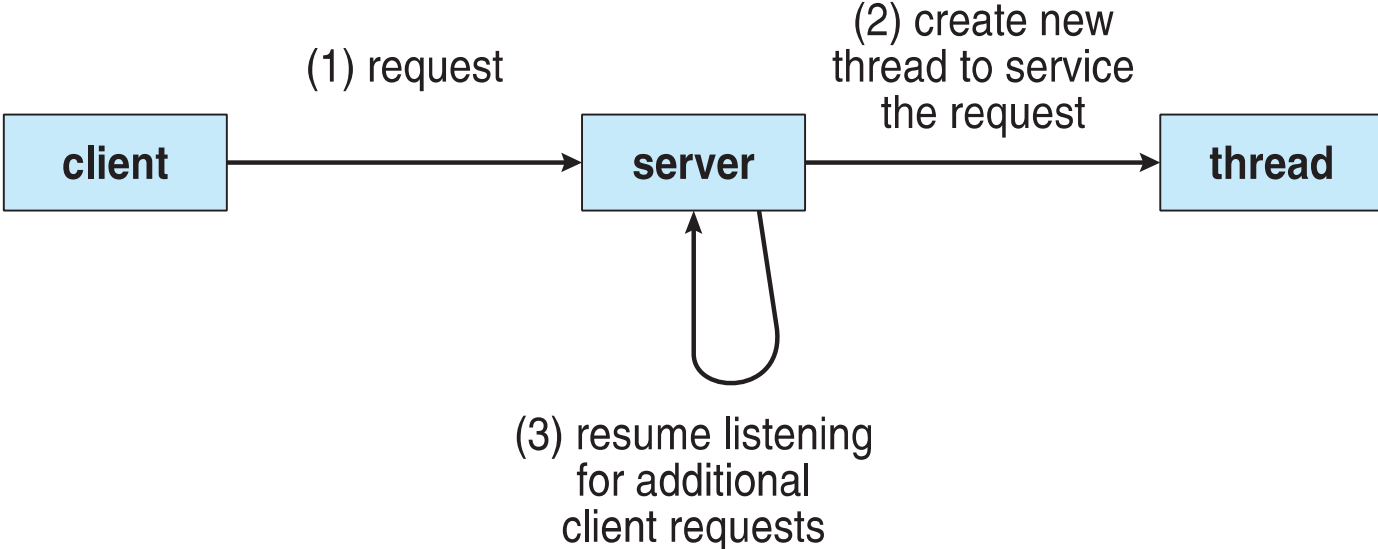
# Thread

- A thread is a basic unit of CPU utilization.
- A traditional process has a single thread of control.
- If a process has multiple threads, it can perform more than one task at a time.
- A thread comprises of a
  - Thread ID – Unique ID for a thread in execution
  - Program counter – Keeps track of instruction to execute
  - System Register set – Active Variables of thread
  - Stack – All execution history (Can be used for debugging)
- Most modern applications are multithreaded
- Threads run within the 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

---



# More Benefits

---

- **Responsiveness** – may allow continued execution if part of the process is blocked
- **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 by assign each thread to a processor

# Threads vs. Processes

Thread	Process
Thread means segment of a process	Process means any program is in execution
Thread is called light weight process	Processes are normally heavy weight
Thread takes less time for creation	Process takes more time for creation
Thread takes less time to terminate	Process takes more time to terminate
Thread takes less time for context switching	Process takes more time for context switching
Thread consumes less resources	Process consumes more resources
Thread has Parents' PCB, its own Thread Control Block and Stack and common Address space	Process has its own Process Control Block, Stack and Address Space



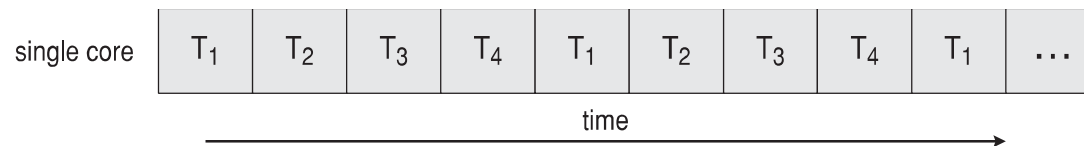
# Multicore Programming

---

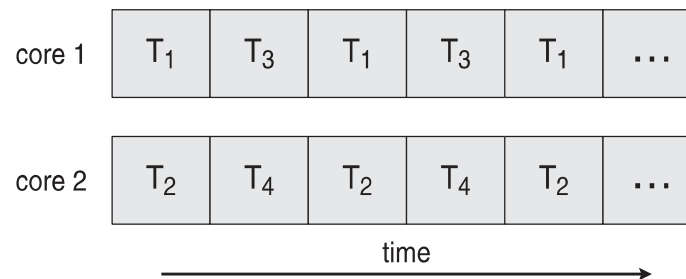
- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
  - Identifying tasks - Examine applications to find activities that can be performed concurrently.
  - Data splitting - To prevent the threads from interfering with one another.
  - Data dependency - Dependent tasks need to be synchronized to assure access in the proper order.
  - Testing and debugging - More difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.
  - Balance.
- 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:



- Parallelism on a multi-core system:



# Multithreading Models

---

- There are two types of threads to manage in a modern system: **User threads** and **kernel threads**.
- **User threads** are supported above the kernel, **without kernel support**. Application programmers put **them** into their programs.
- **Kernel threads** are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple simultaneous tasks or **to service multiple kernel system calls simultaneously**.
- In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.
  - **Many-to-One**
  - **One-to-One**
  - **Many-to-Many**



# User Threads vs. Kernel Threads

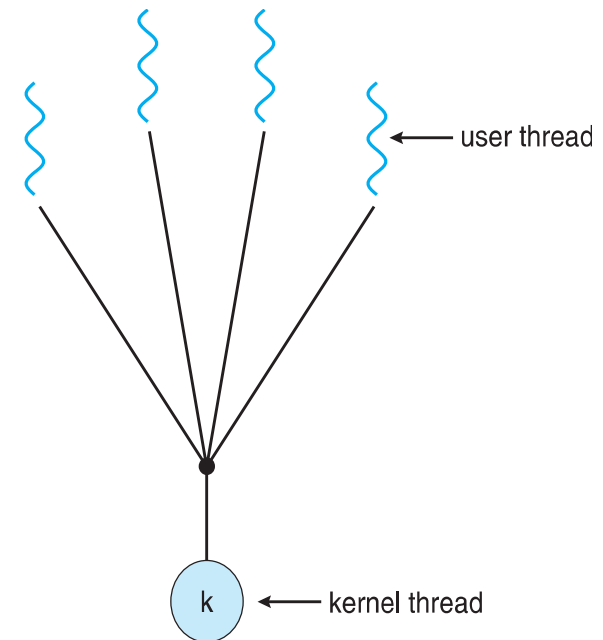
---

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX (Portable Operating System Interface) Pthreads
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
  - Windows
  - Solaris
  - Linux
  - Mac OS X

# Multithreading Models

## Many to One

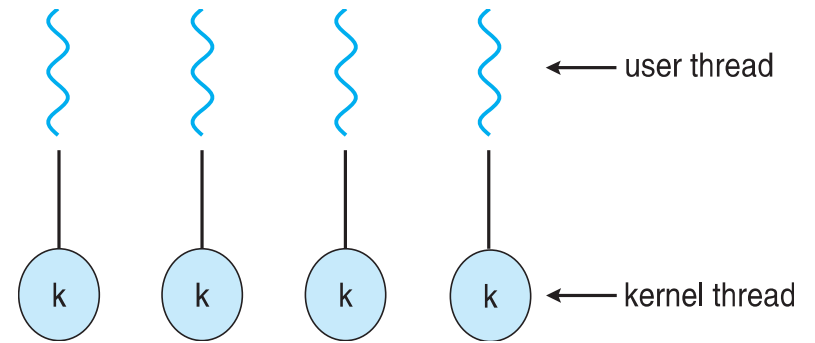
- Many user-level threads mapped to a single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on the multicore system because only one may be in the kernel at a time
- Green threads for Solaris implement the many-to-one model in the past.
- Few systems continue to do so today.



# Multithreading Models

## One to One

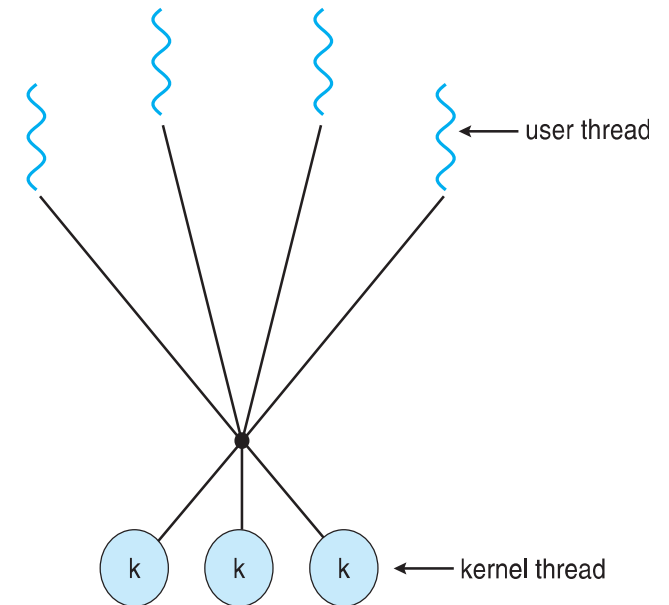
- Each user-level thread maps to the kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process is sometimes restricted due to overhead.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



# Multithreading Models

## 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
  - Solaris prior to version 9
  - Windows with the ThreadFiber package



# Thread Library


---

- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space.
- There are three main thread libraries in use today:
  1. **POSIX Pthreads** – It can be a user or kernel library, as an extension to the POSIX standard.
  2. **Win32 threads** - provided as a kernel-level library on Windows systems.
  3. **Java threads** - Since Java runs on a JVM, the implementation of threads is based upon the OS that JVM is running on, i.e. either Pthreads or Win32 threads depending on the OS.

---

# Thread 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
  - Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- 

---

# Thread Library

## Windows

- Creating threads using the Windows thread library is similar to the Pthreads
- Threads are created in the Windows API using the `CreateThread()`
- Thread attributes include
  - Security information
  - Size of the stack
  - Flag that can be set to indicate if the thread is to start in a suspended state

---

# Thread Library

## Java

- Java threads are managed by the JVM
- Typically implemented using the threads 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();
}
```



---


# 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 (Open Multi-Processing)
  - Grand Central Dispatch

---

# Implicit Threading

## 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
    - Tasks could be scheduled to run periodically, which is useful for tasks that need to be performed on a regular basis, such as updating a cache or sending out notifications.
- 

# Implicit Threading

## 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;
}
```

---

# Implicit Threading

## 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
- 

---


# Implicit Threading

## Grand Central Dispatch

- Two types of dispatch queues:
- **serial** – blocks removed in FIFO order.
- **concurrent** – removed in FIFO order but several may be removed at a time
  - Three system wide queues with priorities
    - low,
    - Default
    - high

---

# 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
- 

---

# Threading Issues

## Semantics of `fork()` and `exec()` system calls

- Does `fork()` duplicate only the calling thread or all threads?
  - System dependant: Some UNIXes have two versions of `fork`
- `exec()` usually works as normal – replace the running process including all threads: If the new process execs right away, there is no need to copy all the other threads. If it doesn't, then the entire process should be copied.

# Threading Issues

---

## 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



# Threading Issues

---

## Signal Handling

- 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 **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_t tid;
/* create the thread */
pthread_create(&tid, 0, worker, NULL);
. . .
/* cancel the thread */
pthread_cancel(tid);
```

# Threading Issues

---

## Thread Local Storage (TLS)

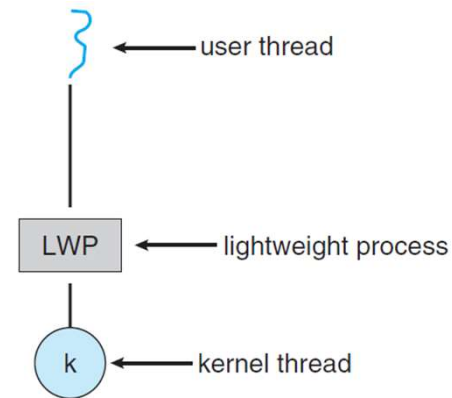
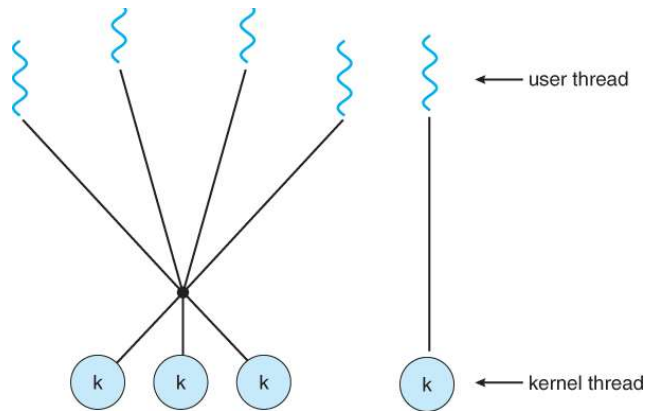
- Thread-local storage (TLS) **allows each thread to have its own copy of data**
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to static data
  - TLS is unique to each thread

# Threading Issues: Scheduler Activation

Scheduler Activation is a technique for communication between the user-level thread and the kernel-level thread

Many implementations of threads provide a virtual processor as an interface between the user thread and the kernel thread, particularly for the many-to-many model.

This virtual processor is known as a "Lightweight Process", LWP.



# Threading Issues: Scheduler Activation

---

- There is a one-to-one correspondence between LWPs and kernel threads.
- The number of kernel threads available, ( and hence the number of LWPs ) may change dynamically.
- The application ( user level thread library ) maps user threads onto available LWPs.
- The kernel communicates to the user-level thread library when certain events occur ( such as a thread about to block ) via an ***upcall***, which is handled in the thread library by an ***upcall handler***.

If the kernel thread blocks, then the LWP blocks, which blocks the user thread.

---

# Exercise 1

- Write a C program that creates a new thread and displays a message “Hello” from the thread.
- Analyze the working of the program by getting the ids of both parent and child thread using `pthread_self()`.
- Below are some important methods from `<pthread.h>`
  - `pthread_create`
  - `pthread_join`

---

## Exercise 2

- Write a C program that takes a shell argument and then calculates the incremental sum itself. For example, if the argument passed is 3, the incremental sum will be 6 (1+2+3).
- The output of the program should be the incremental sum, e.g., 6 as above.
- The incremental sum function should run in the child thread and the result of the sum should be displayed in the parent thread.
- The child should exit before parent can display the sum.
- Below are some important methods from `<pthread.h>`
  - `pthread_create`
  - `pthread_join`

---

# Practice

- Write a C program that takes a shell argument and then calculates the incremental sum and the mean. For example, if the argument passed is 3, the incremental sum will be 6 ( $1+2+3$ ) and mean would be  $2(1+2+3/3)$ .
- The output of the program should be the incremental sum and mean.
- The program should follow the below mentioned attributes
  - The incremental sum function should run in one child thread
  - The mean function should run in another child thread
  - The result of the sum and mean should be displayed in the parent thread.
- In total there would be 3 threads, e.g., 2 child and 1 parent thread.