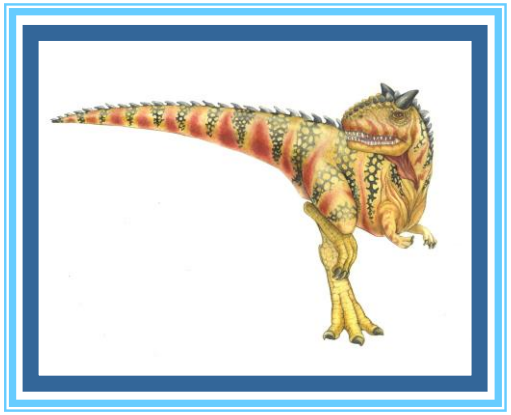
Chapter 4: Threads & Concurrency

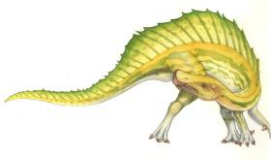


Chapter 4: Threads

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

Objectives

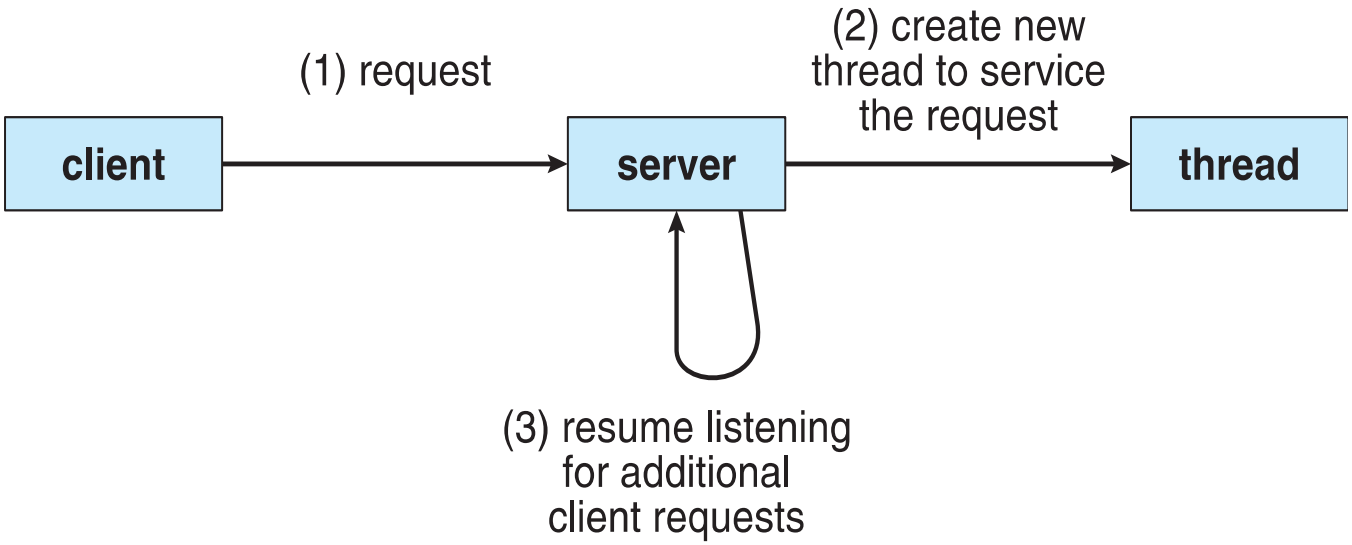
- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming

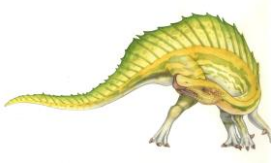


Motivation

- 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



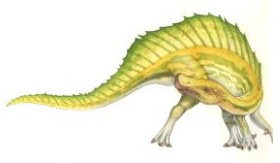


Benefits

- **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 multiprocessor architectures

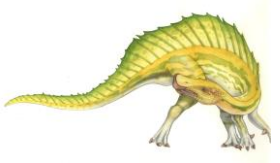
Multicore Programming

- **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



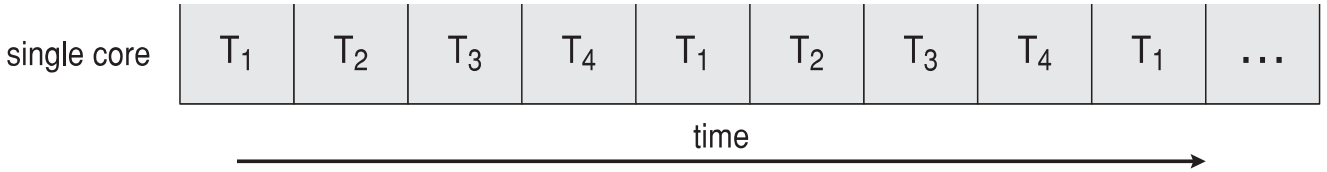
Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
    - For example, if we want to square every element in a large list, we can divide the list into chunks and assign each chunk to a different process (core).
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
    - For example, one thread can perform addition, another can perform subtraction, and another can perform multiplication.
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - AMD EPYC Genoa 9754 processor has **128 cores** and **256 threads**.
    - **256 threads**: Each core supports 2 threads, so 128 cores × 2 threads per core = 256 threads

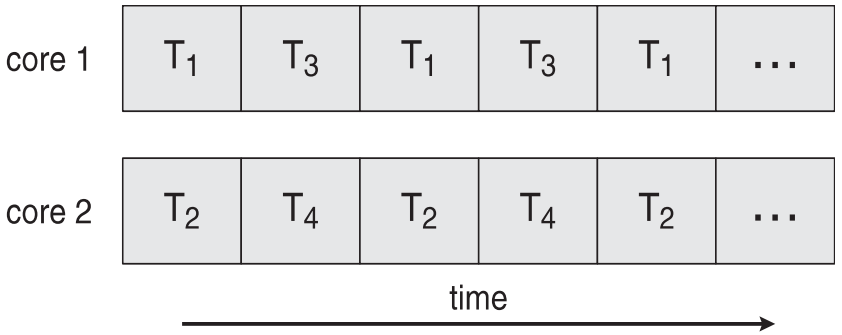


Concurrency vs. Parallelism

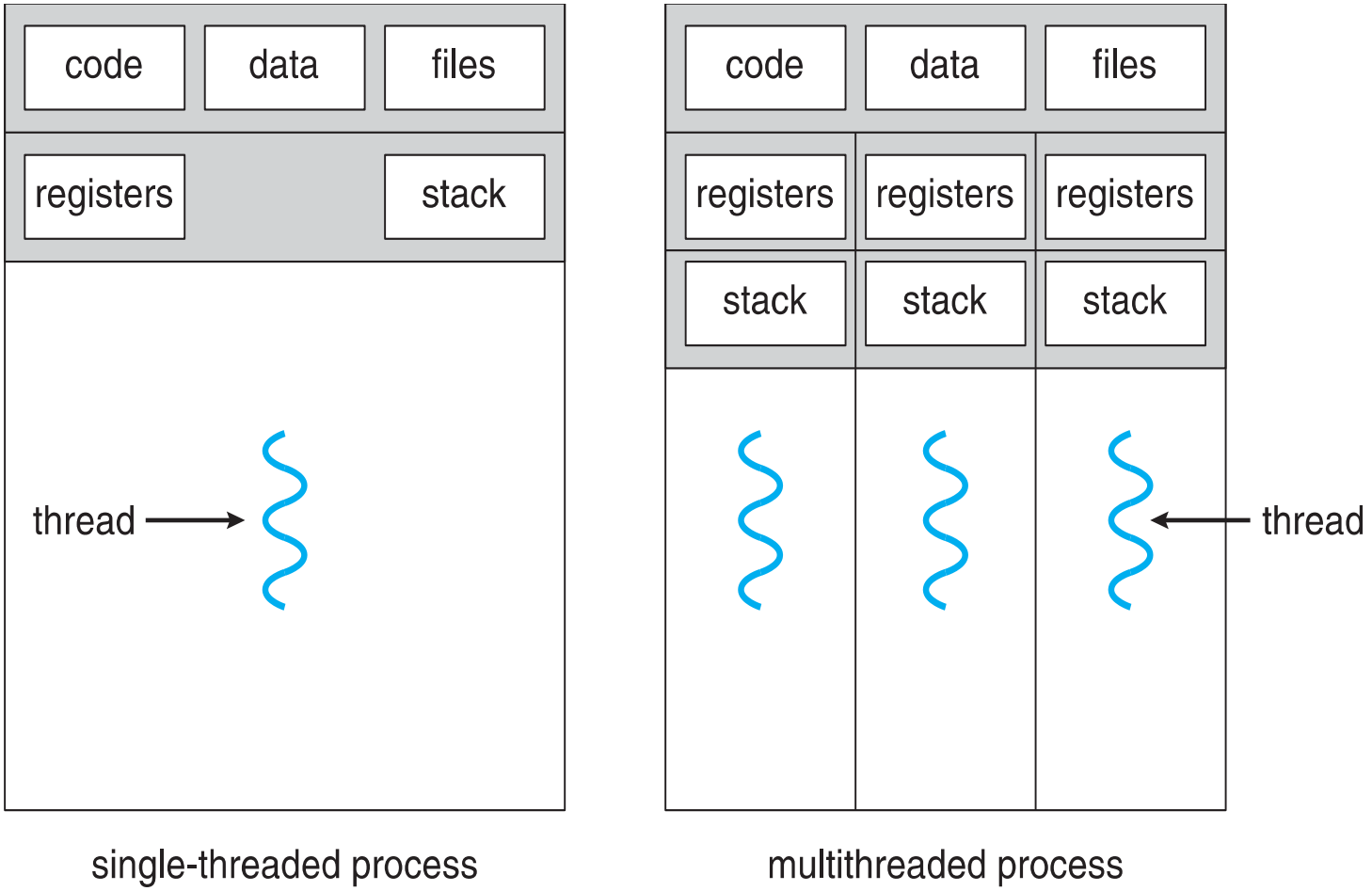
- **Concurrent execution on single-core system:**

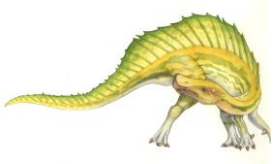


- **Parallelism on a multi-core system:**



Single and Multithreaded Processes





Amdahl's Law

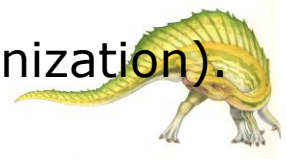
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

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

- 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?
  - Memory bandwidth limitations.
  - Cache coherence issues.
  - Communication overhead between cores.
  - Non-parallelizable parts of the workload (e.g., I/O, synchronization).



Example 1: Speedup with 4 processors

For $N = 4$ processors:

- $P = 0.40$ (40% parallelizable)
- $1 - P = 0.60$ (60% serial)

Using Amdahl's Law:

$$S(4) = \frac{1}{(1 - 0.40) + \frac{0.40}{4}} = \frac{1}{0.60 + 0.10} = \frac{1}{0.70} \approx 1.43$$

So with 4 processors, the speedup is approximately **1.43×**.

Example 2: Speedup with 8 processors

For $N = 8$ processors:

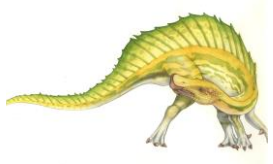
$$S(8) = \frac{1}{(1 - 0.40) + \frac{0.40}{8}} = \frac{1}{0.60 + 0.05} = \frac{1}{0.65} \approx 1.54$$

With 8 processors, the speedup is approximately **1.54×**.

Example 3: Speedup with 16 processors

For $N = 16$ processors:

$$S(16) = \frac{1}{(1 - 0.40) + \frac{0.40}{16}} = \frac{1}{0.60 + 0.025} = \frac{1}{0.625} \approx 1.60$$



Example with a Smaller Serial Portion:

Now let's assume **90%** of the task can be parallelized ($P = 0.90$), and the serial portion is only **10%**. Here's how the speedup behaves:

Speedup with 4 processors:

$$S(4) = \frac{1}{(1 - 0.90) + \frac{0.90}{4}} = \frac{1}{0.10 + 0.225} = \frac{1}{0.325} \approx 3.08$$

Speedup with 8 processors:

$$S(8) = \frac{1}{(1 - 0.90) + \frac{0.90}{8}} = \frac{1}{0.10 + 0.1125} = \frac{1}{0.2125} \approx 4.71$$

Speedup with 16 processors:

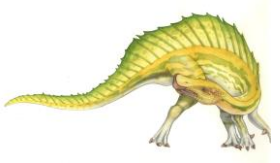
$$S(16) = \frac{1}{(1 - 0.90) + \frac{0.90}{16}} = \frac{1}{0.10 + 0.05625} = \frac{1}{0.15625} \approx 6.40$$

Speedup with 64 processors:

$$S(64) = \frac{1}{(1 - 0.90) + \frac{0.90}{64}} = \frac{1}{0.10 + 0.01406} = \frac{1}{0.11406} \approx 8.76$$

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
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

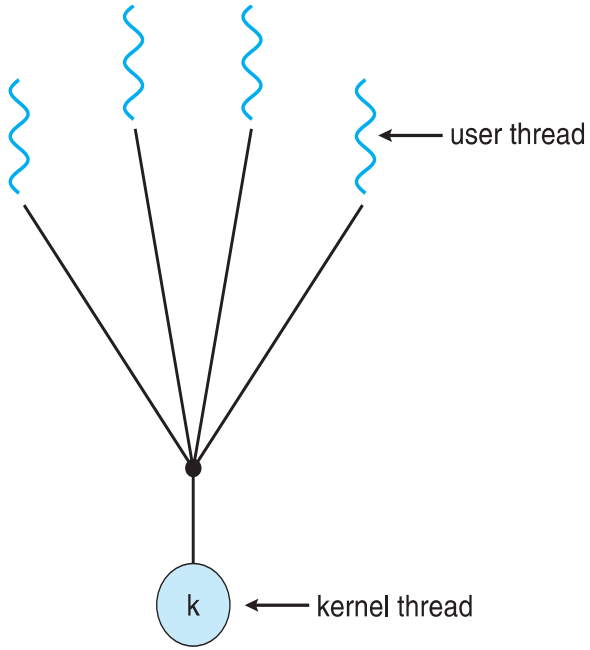


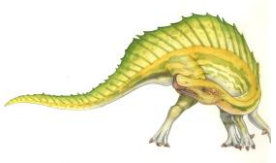
Multithreading Models

- 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**





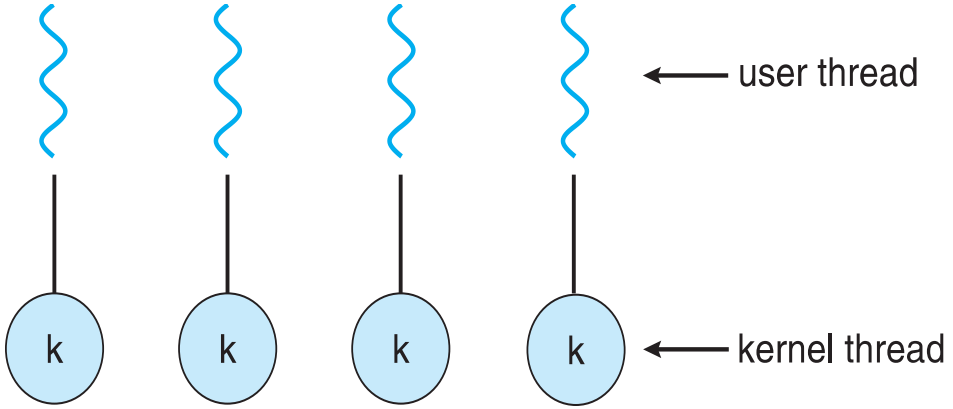
Many-to-One

```
import asyncio

async def task(name):
    print(f"{name}: Running task")
    await asyncio.sleep(1)
    print(f"{name}: Task finished")

async def main():
    # Simulating many-to-one threading
    await asyncio.gather(task("Thread 1"), task("Thread 2"), task("Thread 3"

if __name__ == "__main__":
    asyncio.run(main())
```

```
Thread 1: Running task
Thread 2: Running task
Thread 3: Running task
Thread 1: Task finished
Thread 2: Task finished
Thread 3: Task finished
```

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

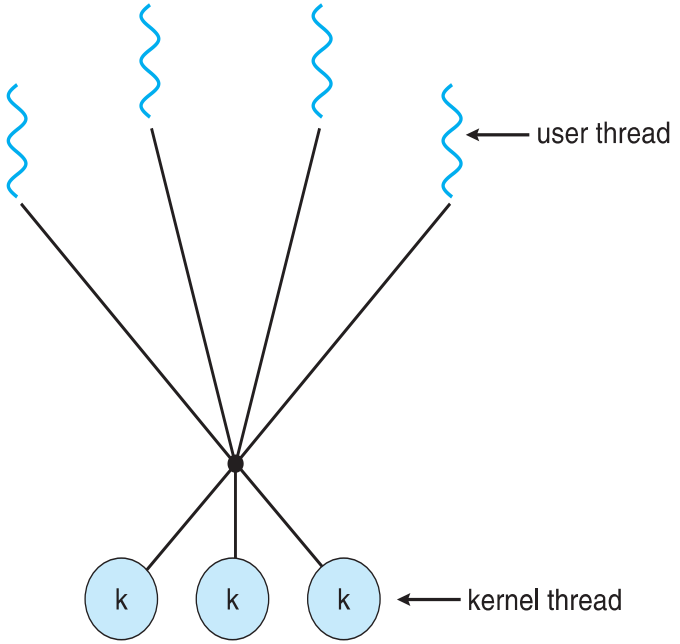


One-to-One

```
import threading
import time

def task(name):
    print(f"{name}: Starting task")
    time.sleep(1)
    print(f"{name}: Task finished")

if __name__ == "__main__":
    # Creating multiple threads (one-to-one mapping)
    t1 = threading.Thread(target=task, args=("Thread 1",))
    t2 = threading.Thread(target=task, args=("Thread 2",))

    t1.start()
    t2.start()
                    Thread 1: Starting taskThread 2: Starting task

    t1.join()       Thread 1: Task finishedThread 2: Task finished
    t2.join()
```

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



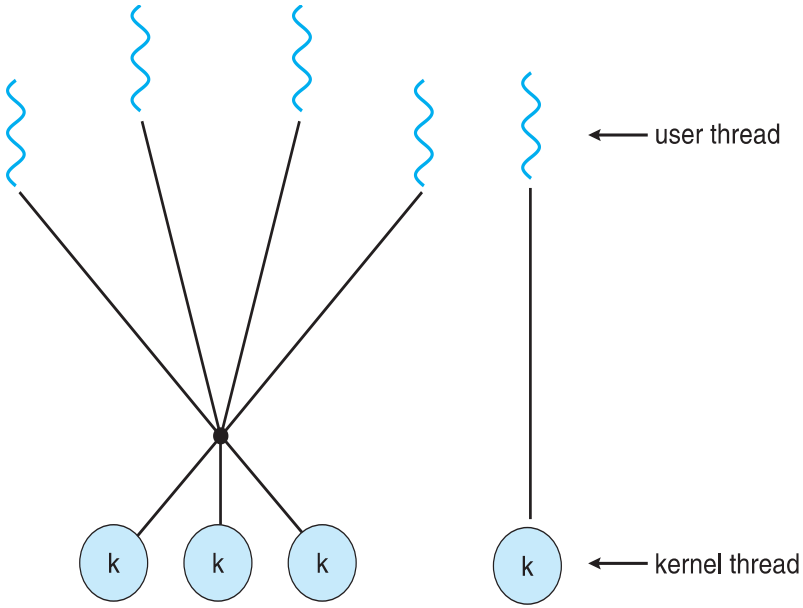
Many-to-Many Model

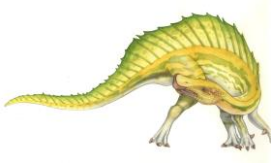
```
from concurrent.futures import ThreadPoolExecutor
import time

def task(name):
    print(f"{name}: Starting task")
    time.sleep(1)
    print(f"{name}: Task finished")

if __name__ == "__main__":
    # Simulating many-to-many threading using ThreadPoolExecutor
    with ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(task, "Thread 1")
        executor.submit(task, "Thread 2")
        executor.submit(task, "Thread 3")
```

```
Thread 1: Starting taskThread 2: Starting task

Thread 1: Task finishedThread 2: Task finished

Thread 3: Starting task
Thread 3: Task finished
```

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

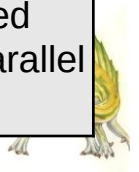




Thread Libraries

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

Platform	Multithreading Library	Description
POSIX (UNIX-like)	Pthreads (POSIX Threads)	Standard threading library for UNIX-like operating systems like Linux, FreeBSD, and macOS.
Java	java.util.concurrent / Java Threads	Native support for multithreading in Java via Thread class and ExecutorService in java.util.concurrent.
Windows	Windows Threads API (Win32 API)	Native Windows API for thread creation and management.
.NET (Windows, Linux, macOS)	System.Threading (C#/.NET)	Provides a managed, high-level API for multithreading across different platforms using C#.
OpenMP (C, C++, Fortran)	OpenMP	A portable library for parallel programming in shared-memory architectures.
Python	threading / concurrent.futures	Python's built-in threading libraries for managing threads and concurrency.
OpenMPI (Parallel Processing)	OpenMPI (for distributed parallel processing)	A message-passing interface standard used for distributed memory multithreading and parallel processing.



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

Example of the Issue:

Imagine a program with two threads, where:

- Thread 1 holds a lock and is accessing shared data.
- Thread 2 performs a `fork()`.

After `fork()`:

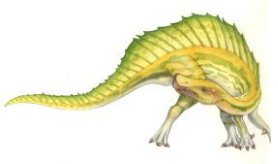
- The child process contains only Thread 2, but **Thread 1's lock remains held** in the parent process.
- The child may try to access the same shared data, resulting in **deadlock** or **inconsistent data access**.

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:
     1. default
     2. 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.)

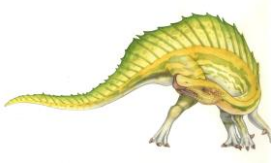
- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process



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
- On Linux systems, thread cancellation is handled through signals

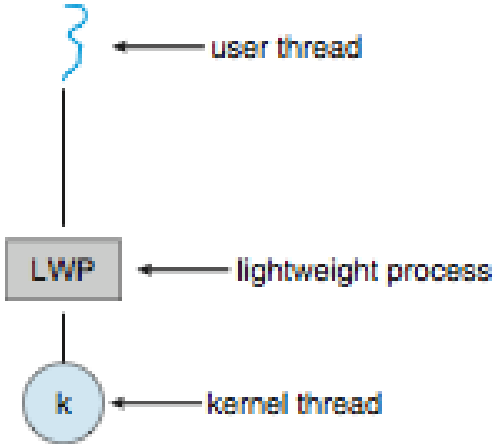
Thread-Local Storage

- **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

- Memory Overhead and Resource Management
- Initialization and Cleanup Complexity
- Performance Impact

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



- Complexity
- Upcall Overhead
- Synchronization Issues

End of Chapter 4