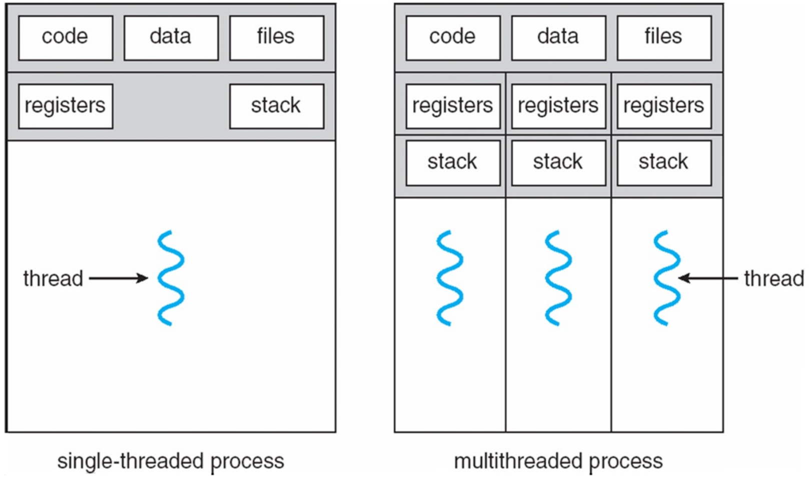
OPERATING SYSTEMS

Threads

FBA

Thread

➢ A thread is a path of execution within a process.



- A thread contains -
  - Thread ID
  - Program Counter
  - Register Set
  - Stack
- Shares with other threads belonging to the same process -
  - Code Section
  - Data Section
  - OS resources

➢ A traditional process has a single thread of control (Single Threaded Process)

➢ Process with multiple threads of control, can perform more than one task at a time (Multi Threaded Process)

Benefits

There are four major categories of benefits to multi-threading:

1. **Responsiveness** - One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.

1. **Resource sharing** - By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.

1. **Economy** - Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.

1. **Scalability, i.e. Utilization of multiprocessor architectures** - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors. (Note that single threaded processes can still benefit from multi-processor architectures when there are multiple processes contending for the CPU, i.e. when the load average is above some certain threshold.)

Multicore Programming

Multicore or **multiprocessor** systems putting pressure on programmers, challenges include:

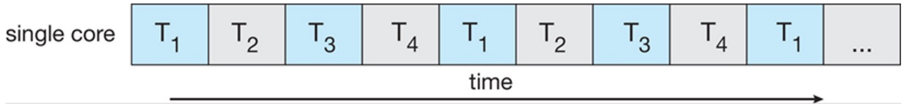
- ➔ Dividing activities
- ➔ Balance
- ➔ Data splitting
- ➔ Data dependency
- ➔ Testing and debugging

Parallelism implies a system can perform more than one task simultaneously

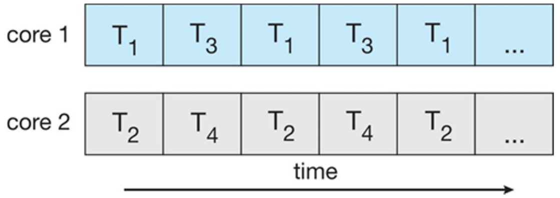
Concurrency supports more than one task making progress

- Single processor / core, scheduler providing concurrency

Concurrent execution on single-core system:



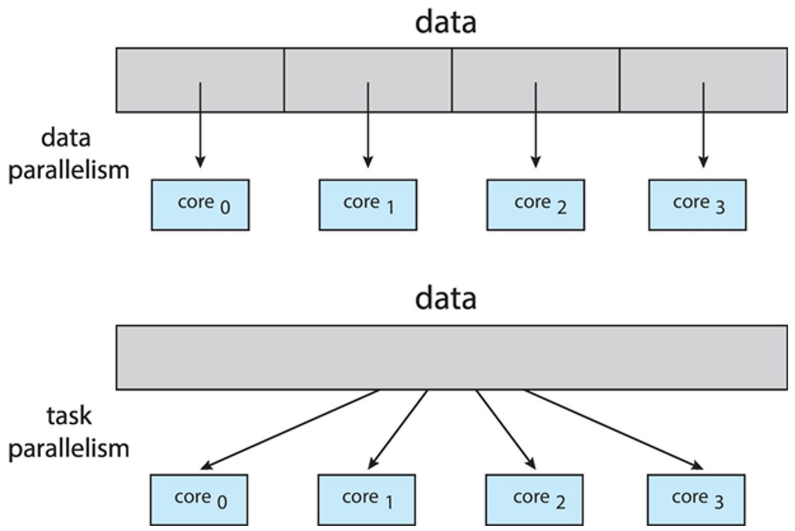
Parallelism on a multi-core system:



Multicore Programming

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

Task parallelism – distributes threads across cores, each thread performing unique operation



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 is number of 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

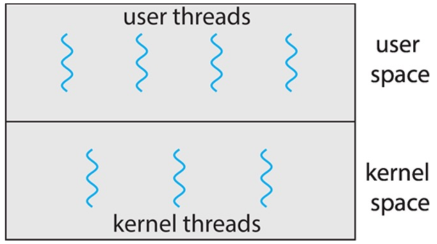
OPERATING SYSTEMS

Multithreading Models

FBA

User and Kernel Threads

- There are two types of threads to be managed in a modern system: User threads and kernel threads.
- In a specific implementation, the user threads must be mapped to kernel threads.



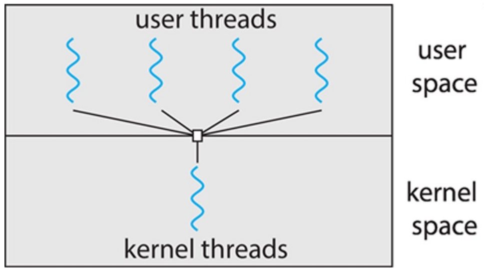
- **User threads** are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.

- **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 and/or to service multiple kernel system calls simultaneously.

Multithreading Models

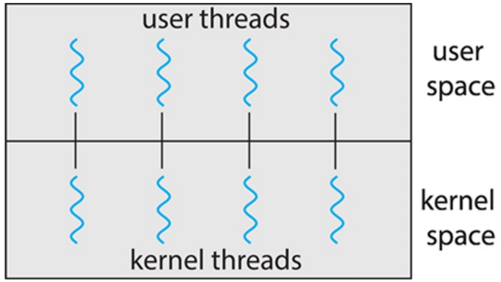
Many-to-One:

- 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
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



One to One:

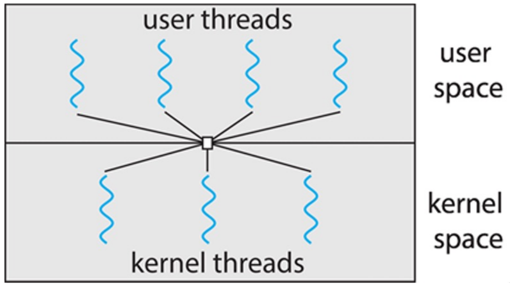
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



Multithreading Models

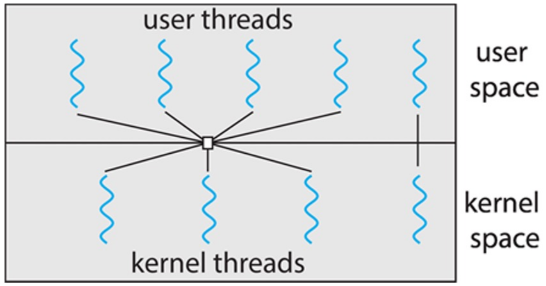
Many-to-Many:

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



OPERATING SYSTEMS

Thread Libraries

FBA

Thread library

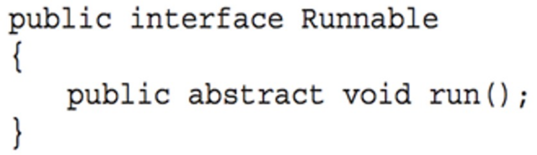
- Thread libraries provide programmers with an API for creating and managing threads.
- Thread libraries may be implemented either in user space or in kernel space. The former involves API functions implemented solely within user space, with no kernel support. The latter involves system calls, and requires a kernel with thread library support.
- There are three main thread libraries in use today:
    - POSIX Pthreads - may be provided as either a user or kernel library, as an extension to the POSIX standard.
    - Win32 threads - provided as a kernel-level library on Windows systems.
    - Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pthreads or Win32 threads depending on the system.

Pthreads

- The POSIX standard (IEEE 1003.1c) defines the specification for pThreads, not the implementation.
- pThreads are available on Solaris, Linux, Mac OSX, Tru64, and via public domain shareware for Windows.
- Global variables are shared amongst all threads.
- One thread can wait for the others to rejoin before continuing.
- pThreads begin execution in a specified function.
- Common in UNIX operating systems (Linux & Mac OS X)

Java Threads

- 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();
}
```

➔ Standard practice is to implement Runnable interface

OPERATING SYSTEMS

Threading Issues

FBA

Threading Issues

- fork() and exec() System Calls
- Thread cancellation
- Signal Handling
- Thread Pool:
- Thread Specific data

Threading Issues: fork() and exec() system calls

- If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?
- Some UNIX systems have chosen to have two versions of fork()
  - one that duplicates all threads
  - another that duplicates only the thread that invoked the fork() system call
- The exec() system call typically works in the same way as described previously
  - That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process — including all threads.

Threading Issues: thread cancellation

- Thread cancellation involves terminating a thread before it has completed
- A thread that is to be cancelled is often referred to as the target thread. Cancellation of a target thread may occur in two different scenarios:
  - **Asynchronous cancellation:** One thread immediately terminates the target thread
  - **Deferred cancellation:** The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Threading Issues: signal handling

- A signal is used in UNIX systems to notify a process that a particular event has occurred
- A signal may be received either synchronously or asynchronously depending on the source of and the reason for the event being signalled
- All signals, whether synchronous or asynchronous, follow the same pattern:
  - A signal is generated by the occurrence of a particular event
  - The signal is delivered to a process
  - Once delivered, the signal must be handled

Threading Issues: signal handling

- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process
- However, delivering signals is more complicated in multithreaded programs, where a process may have several threads
  - Where, then, should a signal be delivered?
- In general, the following options exist:
  - 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.

Threading Issues: thread specific data

- Threads belonging to a process share the data of the process
- Indeed, this data sharing provides one of the benefits of multithreaded programming
- However, in some circumstances, each thread might need its own copy of certain data
  - such data can be called thread-local storage (or TLS)
- For example, in a transaction-processing system, we might service each transaction in a separate thread

Threading Issues: thread pool

- Create a number of threads at the process start-up