Threads

Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues

Outcomes

- Relate the role of threads in process management system.
- Solve the issues of threads in multicore programming and multithreading model.
- Survey similarities and the differences between process and threads.

What is Thread?

- A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.
- A thread shares with its peer threads few information like code segment, data segment and open files.

What is Thread?

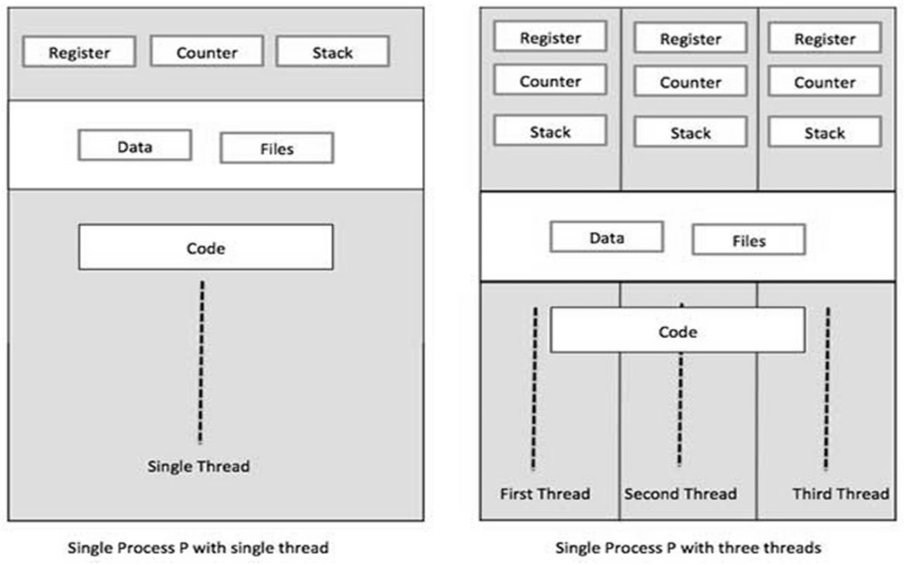
- A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.
- It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
- A traditional (or ***heavyweight***) process has a single thread of control.
- If a process has multiple threads of control, it can perform more than one task at a time.

Characteristics of Threads

- A thread is also called a **lightweight process**.
- Threads provide a way to improve application performance through parallelism.
- Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.
- Each thread belongs to exactly one process and no thread can exist outside a process.
- Each thread represents a separate flow of control.
- Threads have been successfully used in implementing network servers and web server.
- They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

Characteristics of Threads

The following figure shows the working of a single-threaded and a multithreaded process



Comparison between Process and Threads

Process	Thread
Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

- In certain situations, a single application may be required to perform
- several similar tasks. For example, a web server accepts client requests for
- web pages, images, sound, and so forth. A busy web server may have several
- (perhaps thousands of) clients concurrently accessing it. If the web server ran
- as a traditional single-threaded process, it would be able to service only one
- client at a time, and a client might have to wait a very long time for its request
- to be serviced.
- One solution is to have the server run as a single process that accepts
- requests. When the server receives a request, it creates a separate process
- to service that request. In fact, this process-creation method was in common
- use before threads became popular. Process creation is time consuming and
- resource intensive, however. If the new process will perform the same tasks as
- the existing process, why incur all that overhead? It is generally more efficient
- to use one process that contains multiple threads. If the web-server process is
- multithreaded, the server will create a separate thread that listens for client
- requests. When a request is made, rather than creating another process, the
- server creates a new thread to service the request and resume listening for
- additional requests.

Multiple server Architecture

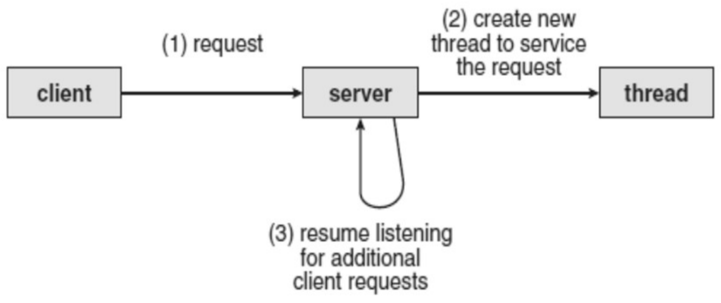

(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Figure 4.2 Multithreaded server architecture.

Advantages of Thread

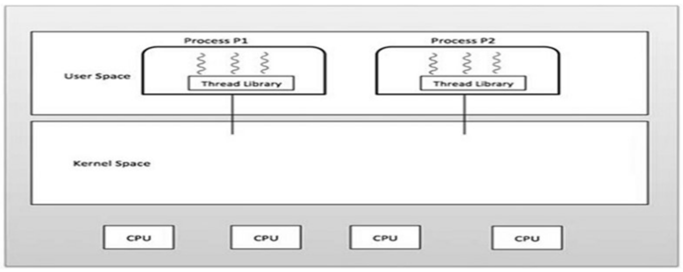
- **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing.
- **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

Types of Thread

- Threads are implemented in following two ways −
- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core

User Level Threads

- In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts.
- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.
- The application starts with a single thread.



User Level Threads

- ▶**Advantages**
- ▶Thread switching does not require Kernel mode privileges.
- ▶User level thread can run on any operating system.
- ▶Scheduling can be application specific in the user level thread.
- ▶User level threads are fast to create and manage.
- ▶**Disadvantages**
- ▶In a typical operating system, most system calls are blocking.
- ▶Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

- In this case, thread management is done by the Kernel.
- There is no thread management code in the application area.
- Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded.
- All of the threads within an application are supported within a single process.
- The Kernel maintains context information for the process as a whole and for individuals threads within the process.
- Scheduling by the Kernel is done on a thread basis.
- The Kernel performs thread creation, scheduling and management in Kernel space.
- Kernel threads are generally slower to create and manage than the user threads.

Kernel Level Threads

- **Advantages**
- Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- If one thread in a process is blocked, the Kernel can schedule another thread of the same process.
- Kernel routines themselves can be multithreaded.
- **Disadvantages**
- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Multithreading Models

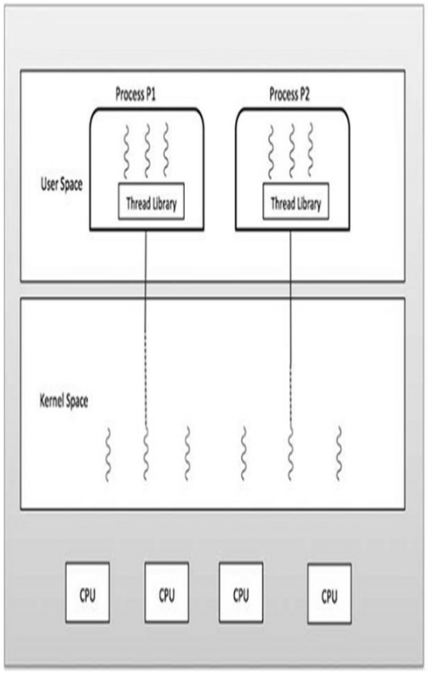
- Ultimately, a relationship must exist between user threads and kernel
- threads.
- Some operating system provide a combined user level thread and Kernel level thread facility.
- In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.
- Multithreading models are three types
- **Many to many relationship.**
- **Many to one relationship.**
- **One to one relationship.**

Distinction between *parallelism* and *concurrency*

- The distinction between ***parallelism*** and ***concurrency*** in this discussion.
- A system is parallel if it can perform more than one task simultaneously.
- In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress.
- Thus, it is possible to have concurrency without parallelism.

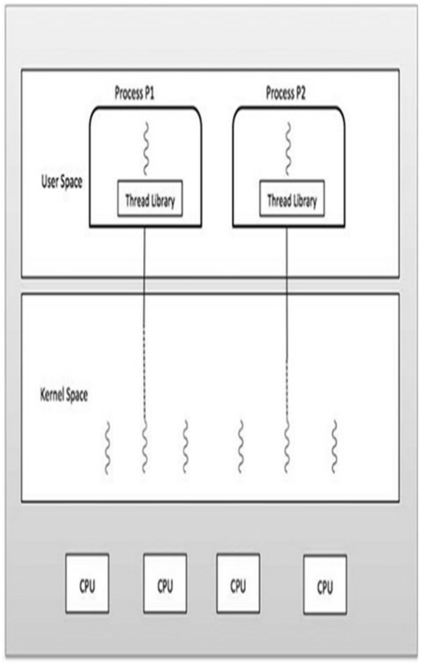
Many to One Model

- The many-to-one model maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient
- However, the entire process will block if a thread makes a blocking system call.
- Also, because only one thread can access the kernel at a time ,multiple threads are unable to run in parallel on multicore systems.



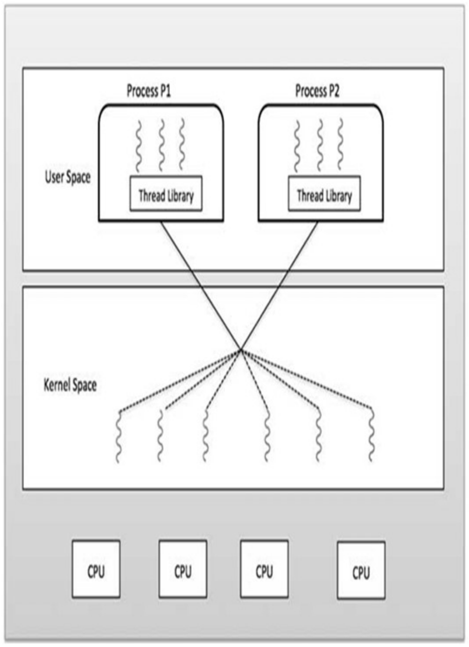
One to One Model

- The one-to-one model maps each user thread to a kernel thread.
- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- It also allows multiple threads to run in parallel on multiprocessors.
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.



Many to Many Model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor).
- Let's consider the effect of this design on concurrency.
- Whereas the many to- one model allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time.



Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded

Multicore Programming

▶**Multicore Programming**

▶A recent trend in computer architecture is to produce chips with multiple ***cores***, or CPUs on a single chip.

▶A multi-threaded application running on a traditional single-core chip would have to interleave the threads, as shown in Figure a.

▶ On a multi-core chip, however, the threads could be spread across the available cores, allowing true parallel processing, as shown in Figure b.

Fig a. **Concurrent execution on a single-core system.**

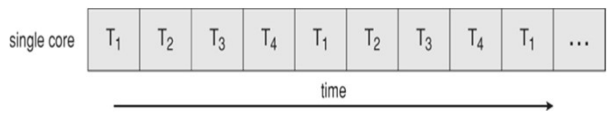
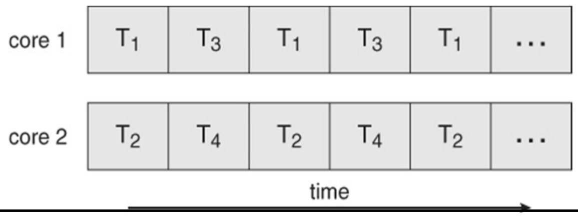


Fig b. **Parallel execution on a multicore system**



Multicore Programming contd..

▶For operating systems, multi-core chips require new scheduling algorithms to make better use of the multiple cores available.

▶ As multi-threading becomes more pervasive and more important (thousands instead of tens of threads), CPUs have been developed to support more simultaneous threads per core in hardware.

▶**Programming Challenges**

▶For application programmers, there are five areas where multi-core chips present new challenges:

▶**Identifying tasks** - Examining applications to find activities that can be performed concurrently.

▶ **Balance** - Finding tasks to run concurrently that provide equal value. I.e. don't waste a thread on trivial tasks.

▶ **Data splitting** - To prevent the threads from interfering with one another.

▶ **Data dependency** - If one task is dependent upon the results of another, then the tasks need to be synchronized to assure access in the proper order.

▶**Testing and debugging** - Inherently more difficult in parallel processing situations, as the race conditions become much more complex and difficult to identify.

Thread Libraries

- A **thread library** provides the programmer with an API for creating and managing threads.
- There are two primary ways of implementing a thread library.
- The first approach is to provide a library entirely in user space with no kernel support.
- All code and data structures for the library exist in user space.
- This means that invoking a function in the library results in a local function call in user space and not a system call.
- The second approach is to implement a kernel-level library supported directly by the operating system.
- In this case, code and data structures for the library exist in kernel space.
- Invoking a function in the API for the library typically results in a system call to the kernel.

TYPESOF THREADS LIBRARY

- Three main thread libraries
- : POSIX Pthreads, Windows, and Java.
- Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.
- The Windows thread library is a kernel-level library available on Windows systems.
- The Java thread API allows threads to be created andmanaged directly in Java programs.
- However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally implemented using a thread library available on the host system.
- This means that on Windows systems, Java threads are typically implemented using the Windows API; UNIX and Linux systems often use Pthreads.

EXAMPLE- basic thread creation

- describe basic thread creation using these three thread libraries.
- As an illustrative example, we design a multithreaded program that performs the summation of a non-negative integer in a separate thread using the well-known summation function:

$$sum = \sum_{i=0}^{N} i$$

- For example, if *N* were 5, this function would represent the summation of integers from 0 to 5, which is 15.
- Each of the three programs will be run with the upper bounds of the summation entered on the command line.
- Thus, if the user enters 8, the summation of the integer values from 0 to 8 will be output.

Synchronous and Asynchronous threading

- With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently. Each thread runs independently of every other
- thread, and the parent thread need not know henits child terminates.Because
- the threads are independent, there is typically little data sharing between
- threads. Asynchronous threading is the strategy used in the multithreaded
- server illustrated in Figure 4.2.
- Synchronous threading occurs when the parent thread creates one or more
- children and then must wait for all of its children to terminate before it resumes
- —the so-called ***fork-join*** strategy. Here, the threads created by the parent
- perform work concurrently, but the parent cannot continue until this work
- has been completed. Once each thread has finished its work, it terminates
- and joins with its parent. Only after all of the children have joined can the
- parent resume execution. Typically, synchronous threading involves significant
- data sharing among threads.

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, in this example the runner( ) function:
- **Pthread.jpeg**

runner() function

- **runner() function**-When this program begins, a single thread of control begins in main().
- After some initialization, main() creates a second thread that begins control in the runner() function.
- Both threads share the global data sum.

- All Pthreads programs must include the **pthread.h** header file. The statement **pthread_t tid** declares the identifier for the thread we will create.
- Each thread has a set of attributes, including stack size and scheduling information
- The **pthread _attr t attr** declaration represents the attributes for the thread.
- A separate thread is created with the **pthread_create()** function call.

- At this point, the program has two threads: the initial (or parent) thread in main() and the summation (or child) thread performing the summation operation in the runner() function.
- This program follows the fork-join strategy described earlier: after creating the summation thread, the parent thread will wait for it to terminate by calling the pthread join() function.
- The summation thread will terminate when it calls the function pthread exit().
- Once the summation thread has returned, the parent thread will output the value of the shared data sum.

Windows Threads

•Similar to pThreads.
•Examine the code example to see the differences, which are mostly syntactic & nomenclature:
•**Windowsthread.jpeg**

In situations that require waiting for multiple threads to complete, the
WaitForMultipleObjects() function is used. This function is passed four
parameters:
1. The number of objects to wait for
2. A pointer to the array of objects
3. A flag indicatingwhether all objects have been signaled
4. A timeout duration (or INFINITE)
For example, if THandles is an array of thread HANDLE objects of size N, the
parent thread can wait for all its child threads to complete with this statement:
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);

Java Threads

▶ALL Java programs use Threads - even "common" single-threaded ones.

▶The creation of new Threads requires Objects that implement the Runnable Interface, which means they contain a method "public void run( )" . Any descendant of the Thread class will naturally contain such a method. (In practice the run( ) method must be overridden / provided for the thread to have any practical functionality.)

▶Creating a Thread Object does not start the thread running - To do that the program must call the Thread's "start( )" method. Start( ) allocates and initializes memory for the Thread, and then calls the run( ) method. (Programmers do not call run( ) directly.)

▶Because Java does not support global variables, Threads must be passed a reference to a shared Object in order to share data, in this example the "Sum" Object.

▶Note that the JVM runs on top of a native OS, and that the JVM specification does not specify what model to use for mapping Java threads to kernel threads. This decision is JVM implementation dependant, and may be one-to-one, many-to-many, or many to one.. (On a UNIX system the JVM normally uses PThreads and on a Windows system it normally uses windows threads.)

▶**JavaThreads.jpeg**

Threading Issues

- **• There are a variety of issues to consider with multithreaded programming**
- - Semantics of fork() and exec() system calls
- - Thread cancellation
- • Asynchronous or deferred
- - Signal handling
- • Synchronous and asynchronous

Semantics of fork() and exec()

▶**• Recall that when** fork() **is called, a separate, duplicate process is created**

▶**• How should** fork() **behave in a multithreaded program?**

- Should all threads be duplicated?

- Should only the thread that made the call to fork() be duplicated?

▶**• In some systems, different versions of** fork() **exist depending on the desired behavior**

- Some UNIX systems have fork1() and forkall()

• fork1() only duplicates the calling thread

• forkall() duplicates all of the threads in a process

▶- In a POSIX-compliant system, fork() behaves the same as fork1()

Semantics of fork() and exec()

▶• **The** exec() **system call continues to behave as expected**

- Replaces the entire process that called it, including all threads!

• If planning to call exec() **after** fork()**, then there is no need to duplicate all of the threads in the calling process**

- All threads in the child process will be terminated when exec() is called

- Use fork1(), rather than forkall() if using in conjunction with exec()

Thread Cancellation

- **•Thread cancellation is the act of terminating a thread before it has completed**
- Example - clicking the stop button on your web browser will stop the thread that is rendering the web page
- **The thread to be cancelled is called the target thread**
- **Threads can be cancelled in a couple of ways**
- -**Asynchronous cancellation-** terminates the target thread immediately
- Thread may be in the middle of writing data ... not so good
- **Deferred cancellation -**allows the target thread to periodically check if it should be cancelled
- Allows thread to terminate itself in an orderly fashion

Signal Handling

▶**Signals are used in UNIX systems to notify a process that a particular event has occurred**

- CTRL-C is an example of an asynchronous signal that might be sent to a process

▶ An **asynchronous signal** is one that is generated from outside the process that receives it

- Divide by 0 is an example of a **synchronous signal** that might be sent to a process

▶ A synchronous signal is delivered to the same process that caused the signal to occur

▶**• All signals follow the same basic pattern:**

A signal is generated by particular event

The signal is delivered to a process

The signal is handled by a signal handler (all signals are handled exactly once)

Thread Pools

▶**• In applications where threads are repeatedly being created/destroyed thread pools might provide a performance benefit**

▶- Example: A server that spawns a new thread each time a client connects to the

▶system and discards that thread when the client disconnects!

▶ **A thread pool is a group of threads that have been pre-created and are available to do work as needed**

- Threads may be created when the process starts

▶ A thread may be kept in a queue until it is needed

▶ After a thread finishes, it is placed back into a queue until it is needed again

▶ Avoids the extra time needed to spawn new threads when they’re needed

Thread Pools

- • **Advantages of thread pools:**
- - Typically faster to service a request with an existing thread than create a new thread (performance benefit)
- - Bounds the number of threads in a process
- • The only threads available are those in the thread pool
- • If the thread pool is empty, then the process must wait for a thread to re-enter the pool before it can assign work to a thread
- • Without a bound on the number of threads in a process, it is possible for a process to create so many threads that all of the system resources are exhausted