THREADS

- 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.
- Ex : A web browser might have one thread display images or text while another thread retrieves data from the network 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 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. 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 as shown in below fig.

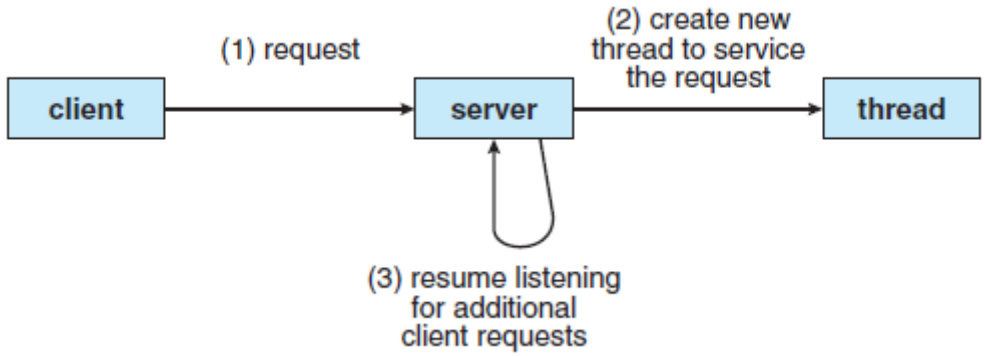

Fig : Multithreaded server architecture services

- Most operating-system kernels are now multithreaded. Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling.

Benefits

The benefits of multithreaded programming are:

1. **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.

2. **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. **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.

4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

User Threads and Kernel Threads

User Threads :

- Supported above the kernel & are implemented by a thread library at the user level.
- The library provides support for thread creation, scheduling & management with no support from kernel (because the kernel is unaware of user-level threads).
- All thread creation and scheduling are done in user space without the need for kernel intervention.
- Example : if the kernel is single-threaded, then any user level threads performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application.
- Ex: POSIX Pthreads,, Mach C-threads etc.

Kernel Threads :

- Are supported directly by the OS.
- Kernel performs thread creation, scheduling and management in kernel space.
- Because thread management is done by the OS
- Kernel threads are generally slower to create and manage than user threads.
- Since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread for execution.
- Ex : Windows 2000, BeOS,Windows NT

Multithreading Models:

- Some relationship must exist between user threads and kernel threads.
- Three common ways of establishing such a relationship:

✓ The Many-To-One Model,
✓ The One-To-One Model, And
✓ The Many-To Many Model.

1. 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
- 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.
- Ex : **Green threads**

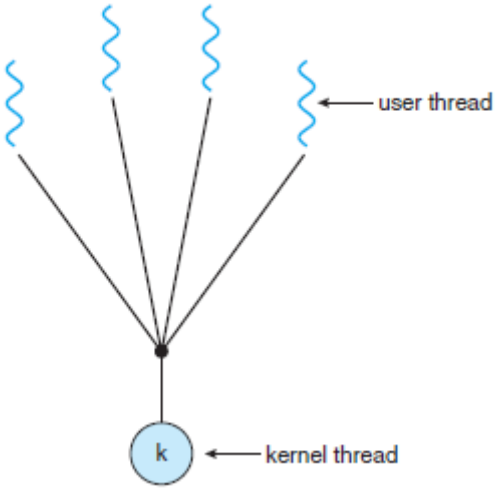

Fig : Many to One Model

2. 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.

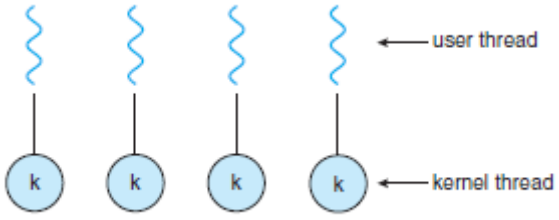

Fig : one – to – one Model

3. 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.

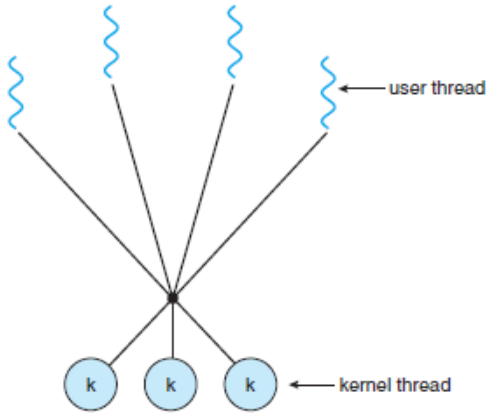

Fig : Many-to-many model.

- 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 user wishes,
- it does not result in true concurrency, because the kernel can schedule only one thread at a time.
- The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application.

Threading Issues

The issues to consider in designing multithreaded programs.

a. The fork() and exec() System Calls

- The **fork() system call** is used to create a separate, duplicate process.
- The semantics of the fork() and exec() system calls change in a multithreaded program.

- 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 and
  - ✓ another that duplicates only the thread that invoked the fork() system call.

- The **exec() system call** invokes the specified in the parameter to exec() will replace the entire process—including all threads.
- Which of the two versions of fork() to use depends on the application.

- If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process.

b. 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:

1. **Asynchronous cancellation**. One thread immediately terminates the target thread.

2. **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

- The difficulty with cancellation occurs in situations where resources have been allocated to a cancelled thread or where a thread is cancelled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation.

- With deferred cancellation, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be cancelled. This allows a thread to check if it should be cancelled at a point when it can safely be cancelled. Such points are called as **cancellation points.**

c. 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.
- All signals, whether synchronous or asynchronous, follow the same pattern:
    1. A signal is generated by the occurrence of a particular event.
    2. The signal is delivered to a process.
    3. Once delivered, the signal must be handled.

- A signal may be ***handled*** by one of two possible handlers:
    1. A default signal handler
    2. A user-defined signal handler

- Every signal has a **default signal handler** that the kernel runs when handling that signal.
- This default action can be overridden by a **user-defined signal handler** that is called to handle the signal.
- Signals are handled in different ways.
- Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.
- Signals are always delivered to a process.
- 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.

d. Thread Pools

- The general idea behind a thread pool is to create a number of threads at process startup and place them into a pool, where they sit and wait for work. When a server receives a request, it awakens a thread from this pool—if one is available—and passes it the request for service. Once the thread completes its service, it returns to the pool and awaits more work. If the pool contains no available thread, the server waits until one becomes free.

Why Thread Pool?

Scenario :

- whenever the server receives a request, it creates a separate thread to service the request.
- The first issue concerns the amount of time required to create the thread, together with the fact that the thread will be discarded once it has completed its work
- The second issue is more troublesome. If we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**.

Benefits of Thread pools :

1. Servicing a request with an existing thread is faster than waiting to create a thread.

2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.