Threads

Thread is a basic unit of CPU utilization.

It is also called Light Weight Process (LWP).

It consists thread ID, a program counter (PC), a register set, and a stack.

Thread 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 (heavy weight) process has a single thread of control.

If the process has multiple threads of control, it can do more than one task at a time.

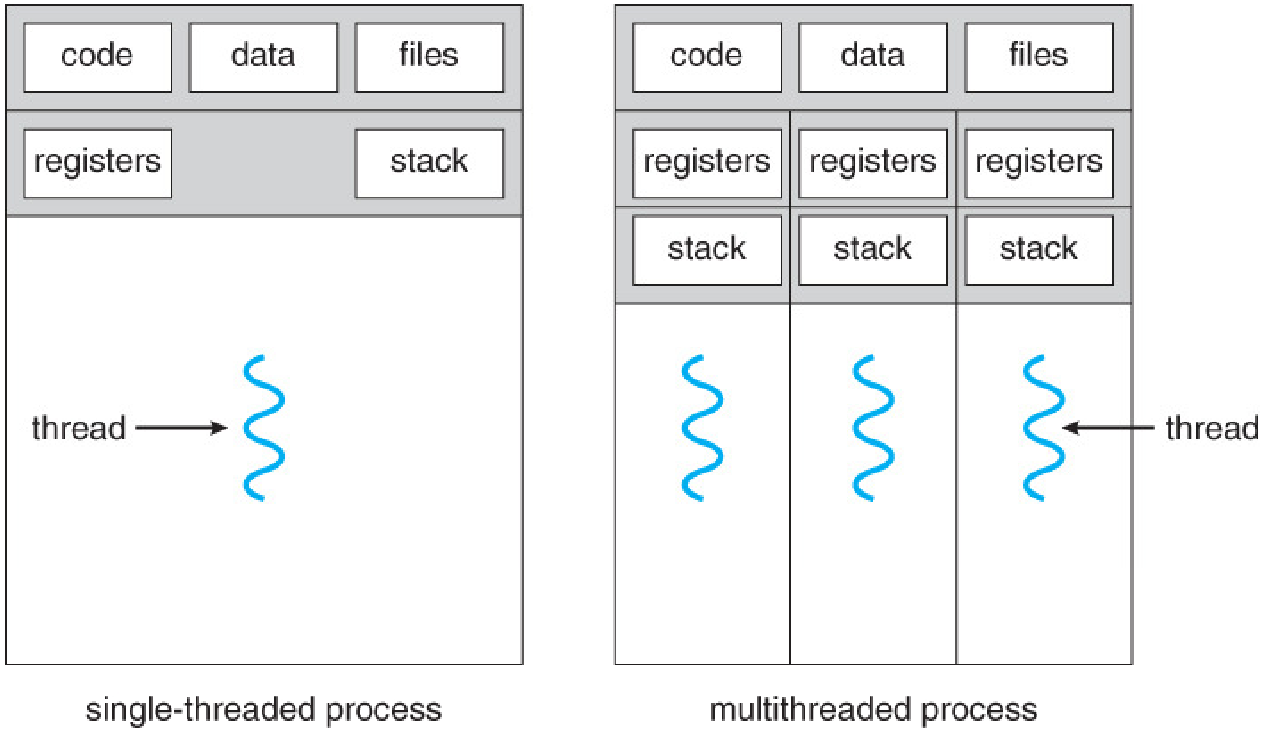

Fig: Single and multithreaded process.

Motivation

Many software packages that run on modern desktop PCs are multithreaded.

An application typically is implemented as a separate process with several threads of control.

For example, a word processer may have a thread for displaying graphics, another thread for reading key strokes from user, and another thread for performing spelling and grammar checking in the background.

In some 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 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.

The amount of time that a client might have to wait for its request to be serviced could be extremely large.

One solution to this problem is that the server run as a single process and when it receives a request, it may create a separate process to serve the request.

But the process creation is heavyweight.

Generally more efficient for one process that contains multiple-threads to serve the same purpose.

This technique is the multi-threaded web-server process.

The server should create a separate thread that would listen for client requests; when a request was made, rather than creating another process, it would create another thread to service the request.

Benefits

i) Responsiveness:

Multithreading is an interactive application may allow a program to continue executing even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

ii) Resource sharing:

Threads share the memory and resources of the process to which they belong.

iii) Economy:

Allocating memory and resources for process creation is costly.

Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads.

Generally, it is much more time consuming to create and manage processes than threads.

iv) Utilization of multiprocessor architectures

The benefits of multithreading can be generally increased in a multiprocessor architecture, where each thread may be running in parallel on a different processor.

In single processor architecture, the CPU generally moves between each thread so quickly as to create an illusion of parallelism, but in reality only one thread is running at a time.

User and Kernel threads

The support for threads may be provided at either the user level, for user threads, or by the kernel, for kernel threads.

User threads

User threads are supported above the kernel and are implemented by a thread library at the user level.

The library provides support for thread creation, scheduling, and management with no support from kernel.

The kernel is unaware of user level threads, so all thread creation and scheduling are done in user space without the need for kernel intervention.

Therefore, user level threads are generally fast to create and manage.

The drawback of user threads is that, if the kernel is single-threaded, then any user level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application.

Kernel threads

Kernel threads are supported directly by the operating system.

The kernel performs thread creation, scheduling, and management in kernel space.

The kernel thread management is done by the operating system, so the kernel threads generally slower to create and manage than the user threads.

If a kernel thread performs a blocking system call, the kernel can schedule another thread in application for execution.

Also in a multi processor environment, the kernel schedule threads on different processors.

Multithreading models

Many systems provide support for both user and kernel threads, resulting in different multithreading models.

Many-to-one model

The many-to-one model maps many user-level threads to one kernel thread.

In this model, the thread management is done in user space, so it is efficient, but the entire process will block if a thread makes a blocking system call.

Here only one thread can access the kernel at a time; therefore multiple threads are unable to run in parallel on multiprocessor.

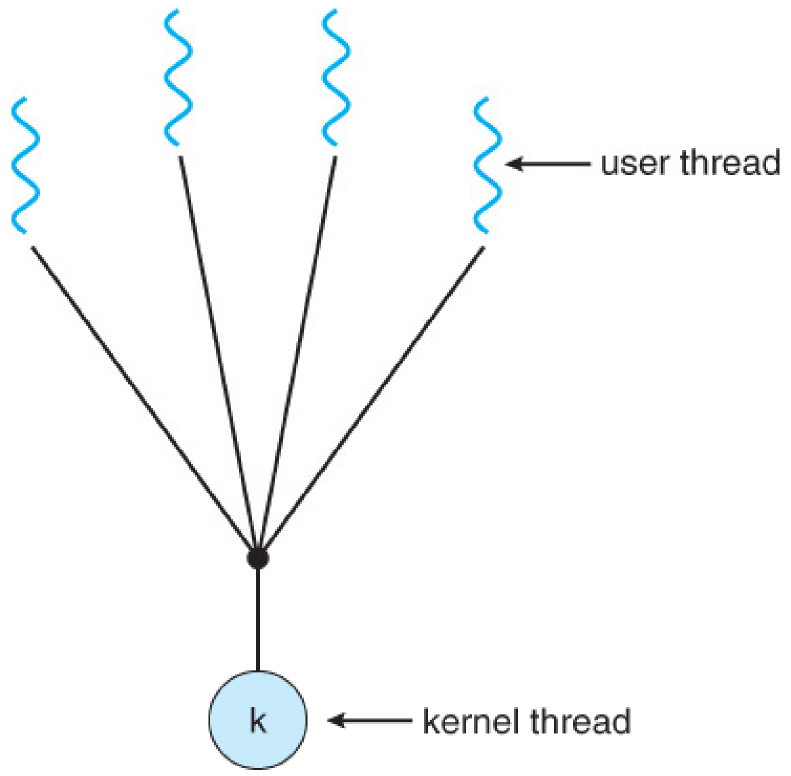

Fig: Many-to-one model.

One-to-one model

The one-to-one model maps each user-level threads to a kernel thread.

It allows another thread to run when a thread makes a blocking system call.

It also allows multiple threads to run in parallel on multiprocessor.

In this way one-to-one model provides more concurrency than the many-to-one model.

The drawback of this model is that creating a user thread requires creating the corresponding kernel thread.

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.

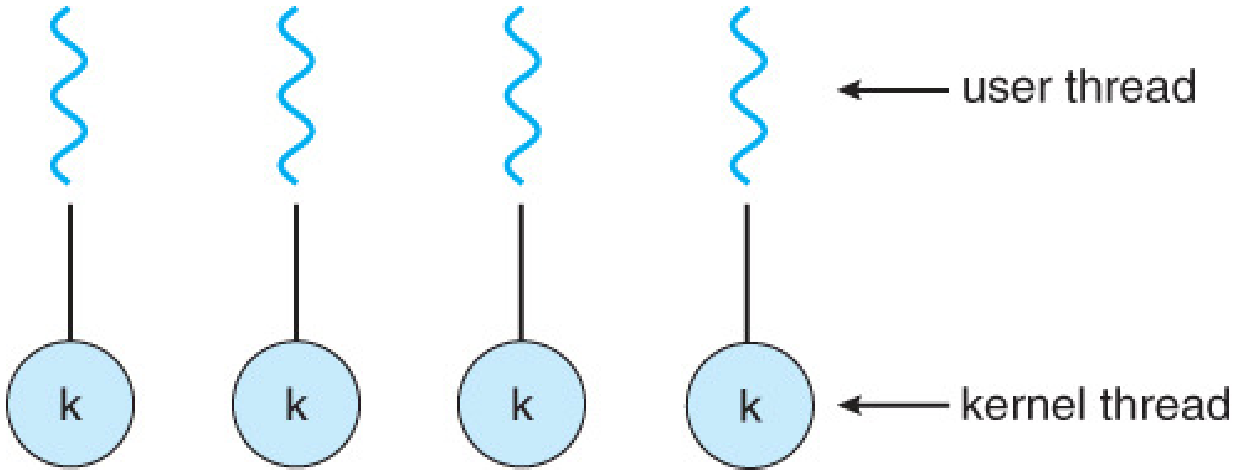

Fig: One-to-one model.

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.

For example, an application may be allocated more kernel threads on a multiprocessor than on a uniprocessor.

In this model, the developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.

Also when a thread performs a blocking system call, the kernel can schedule another thread for execution.

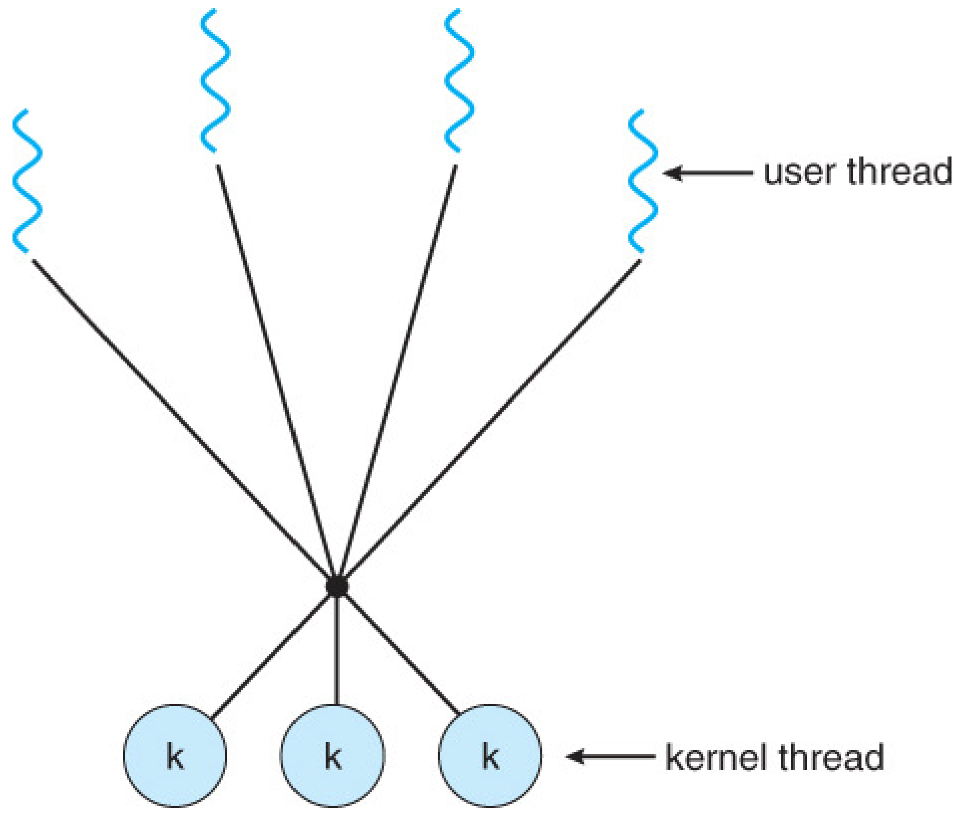

Fig: Many-to-many model.

Threading issues

The *fork* and *exec* system call

In multithreaded program, the semantics of the *fork* and *exec* system call change.

If one thread in a program calls *fork*, then in some systems new process duplicate all threads, and in some other systems the new process is single threaded.

The *exec* system call typically works in the same way.

If a thread invokes the *exec* system call, the program specified in the parameter to *exec* will replace the entire process – included all threads.

Cancellation

Thread cancellation is the task of terminating a thread before it has completed.

For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled.

A thread that is to be cancelled is called target thread.

The cancellation of target thread may occur in two different situations:

i) Asynchronous cancellation

One thread immediately terminates the target thread.

ii) Deferred cancellation

The target thread can periodically check if it should terminate, allowing the target thread 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 if a thread was cancelled while in the middle of updating data it is sharing with other threads.

This becomes troublesome with asynchronous cancellation.

The operating system will often reclaim system resources from a cancelled thread, but often will not reclaim all resources.

Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.

The deferred cancellation works by one thread indicating that a target thread is to be cancelled.

However, cancellation will occur only when the target thread checks to determine if it should be cancelled or not.

This allows a thread to check if it should be cancelled at a point when it can safely be cancelled.

These points are known as cancellation point.

Signal handling

A signal is used in the 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 signaled.

All signals follow the following pattern:

i) A signal is generated by the occurrence of a particular event.

ii) The signal is delivered to a process.

iii) Once delivered, the signal must be handled.

For example, a synchronous signal includes an illegal memory access and division by 0.

If a running program performs either of these actions, a signal is generated.

Synchronous signals are delivered to the same process that causing the signal.

When a signal is generated by an event external to a running process, that process receives the signal asynchronously.

For example, terminating a process with specific keystrokes (such as <Ctrl><Alt><Delete>).

Typically, an asynchronous signal is sent to another process.

A signal may be handled by one of two possible handlers:

i) A default signal handler

ii) 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-define signal handler function.

In this instance, the user defined function is called to handle the signal rather than the default action.

Both synchronous and asynchronous signals may be handled in different ways.

Some signals may be simply ignored (such as changing the size of the window); others may be handled by terminating the program (such as illegal memory access).

Handling signals in single-threaded programs is straightforward; signals are always delivered to a process.

However, delivering signals is more complicated in multithreaded programs, as the processes have several threads.

To delivering the signal, there are following options:

i) Deliver the signal to the thread to which the signal applies.

ii) Deliver the signal to every thread in the process.

iii) Deliver the signal to certain threads in the process.

iv) Assign a specific thread to receive all signals for the process.

The method for delivering a signal depends on the type of signal generated.

For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process.

However, the situation with asynchronous signals is not as clear.

Some asynchronous signals—such as a signal that terminates a process — should be sent to all threads.

Thread pools

In case of multithreaded web server, whenever the server receives a request, it creates a separate thread to service the request.

Although creating a separate thread is certainly superior to creating a separate process, but multithreaded server has potential problems.

Firstly, the amount of time required to create the thread prior to servicing the request, together with the fact that the thread will be discarded once it has completed its work.

Secondly, if we allow 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.

The general idea behind a thread pool is to create a number of threads at process start-up 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 – passing it the request to service.

Once a 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 become free.

In particular, the benefits of thread pools are as follows:

i) Servicing a request with an existing thread is often faster than waiting to create a thread.

ii) 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.

The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests.

Thread-specific data

Threads belonging to a process share the data of the process.

However, each thread might need its own copy of certain data in some circumstances.

Such data is known as thread-specific data.

For example, in a transaction-processing system, we might service each transaction in a separate thread.

Furthermore, each transaction may be assigned a unique identifier.

To associate each thread with its unique identifier, we could use thread specific data.