Operating Systems

Threads

Carlos Alberto Llano R.

8 de septiembre de 2015

Contents

What is a thread?

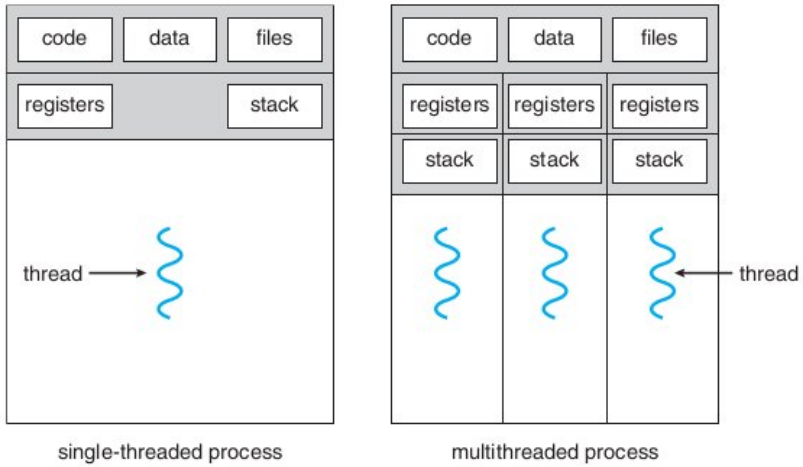
A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.

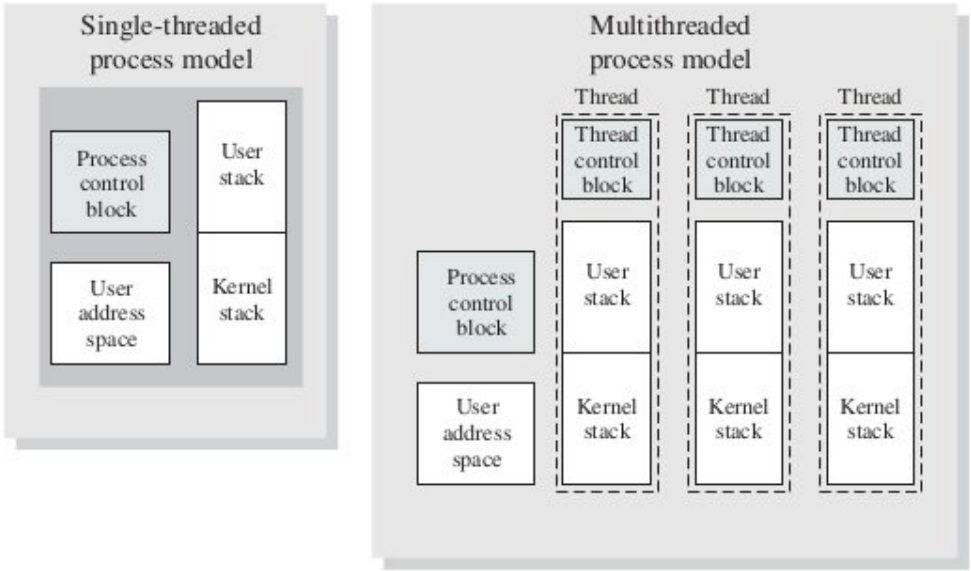
A thread is a basic unit of CPU utilization; it comprises a thread ID , a program counter, a register set, and a stack.

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.

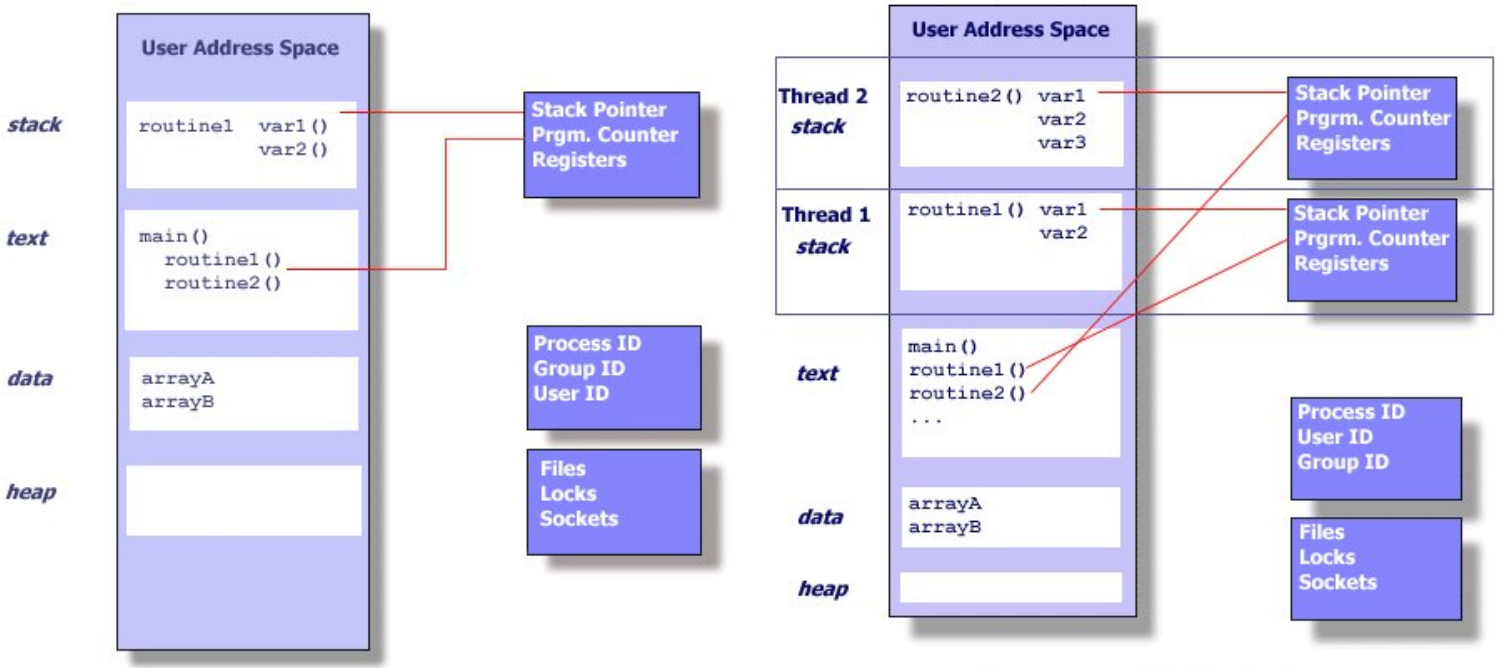
This means: That each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.

Thread model





Thread model



Why Threads?

Imagine what would happen if the web server ran as a traditional single-threaded process?

how would this work?
does it creates a separate process to service that request?

Why Threads?

- Most popular abstraction for concurrency
  - Process creation is time consuming and resource intensive.
  - All threads in one process share memory, file descriptors, etc.
- Allows one process to use multiple CPUs or cores
  - More efficient to use one process that contains multiple threads.
- Allows program to overlap I/O and computation
  - Same benefit as OS running emacs & gcc simultaneously
  - E.g., threaded web server services clients simultaneously
- Most kernels have threads, too

Threads applications examples

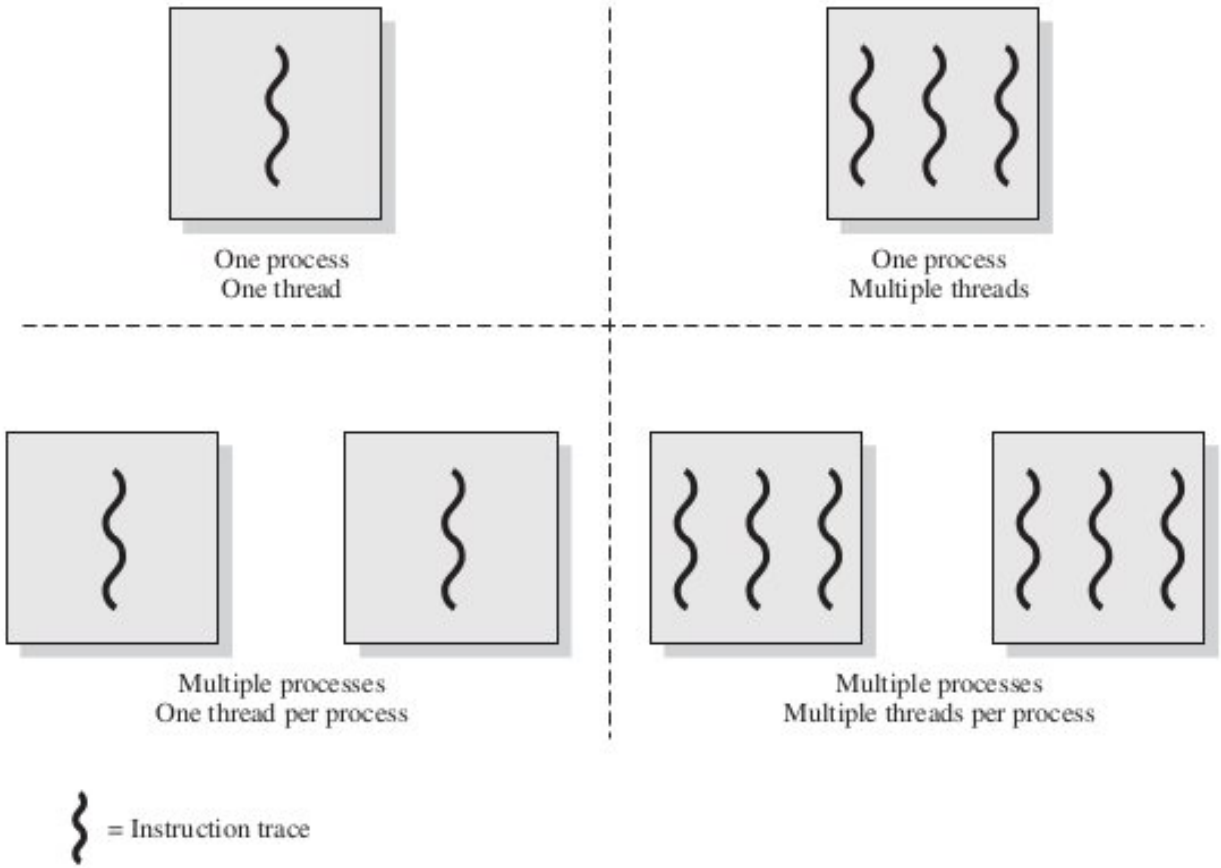
An application typically is implemented as a separate process with several threads of control.

Threads examples:

- A web browser might have one thread display images or text while another thread retrieves data from the network.
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Multithreading

Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process. Approaches:



Benefits

- It takes far less time to create a new thread in an existing process that to create a brand-new process. (Ten times faster than process creation).
- It takes less time to terminate a thread than a process.
- It takes less time to switch between two threads within the same process than to switch between process.
- Threads enhance efficiency in communication between different executing programs. (The benefit of sharing code and data).
- Multithreading an interactive application may allow a program to continue running even if part of it is blocked.
- Scalability.

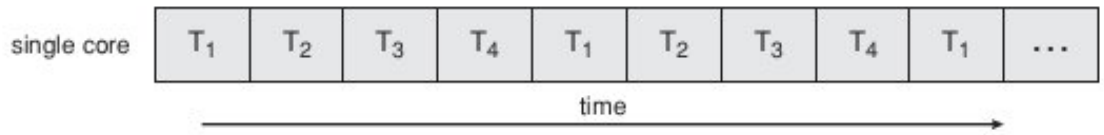
Parallelism and Concurrency

Parallelism and Concurrency

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.

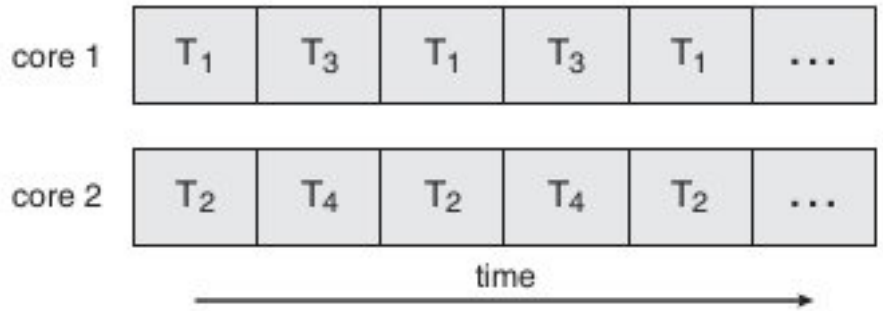
A recent trend in computer architecture is to produce chips with multiple cores, or CPUs on a single chip.

A multi-threaded application running on a traditional single-core chip would have to interleave the threads



Parallelism and Concurrency

On a multi-core chip, however, the threads could be spread across the available cores, allowing true parallel processing



Why use concurrence or parallelism?

There are two main reasons to use concurrency (or parallelism) in an application: separation of concerns and performance.

Separation of concerns

You can use concurrency (or parallelism) to separate distinct areas of functionality, even when the operations in these distinct areas need to happen at the same time.

Consider a processing-intensive application with a user interface, such as a DVD player application for a desktop computer.

Using threads in this way generally makes the logic in each thread much simpler, because the interactions between them can be limited to clearly identifiable points.

for performance

There are two ways to use concurrency or parallelims for performance. The first, and most obvi- ous, is to divide a single task into parts and run each in parallel (different cores), thus reducing the total runtime. This is task parallelism.

In terms of data, each thread performs the same operation on different parts of the data. This latter approach is called data parallelism.

For example dividing a large image up into pieces and performing the same digital image processing on each piece on different cores.

When not to use concurrency

Fundamentally, the only reason not to use concurrency is when the benefit is not worth the cost. Code using concurrency is harder to understand in many cases, so there's a direct intellectual cost to writing and maintaining multithreaded code, and the additional complexity can also lead to more bugs.

When not to use concurrency

the performance gain might not be as large as expected; there's an inherent overhead associated with launching a thread, because the OS has to allocate the associ- ated kernel resources and stack space and then add the new thread to the scheduler, all of which takes time.

When not to use concurrency

Each thread requires a separate stack space.

This is particularly a problem for 32-bit processes with a flat architecture where there's a 4 GB limit in the available address space:

if each thread has a 1 MB stack (as is typical on many systems), then the address space would be all used up with 4096 threads, without allowing for any space for code or static data or heap data.

Although 64-bit (or larger) systems don't have this direct address-space limit, they still have finite resources: if you run too many threads, this will eventually cause problems.

Amdahl's Law (1967) - Performance of Software on Multicore

A program (or algorithm) which can be parallelized can be split up into two parts:

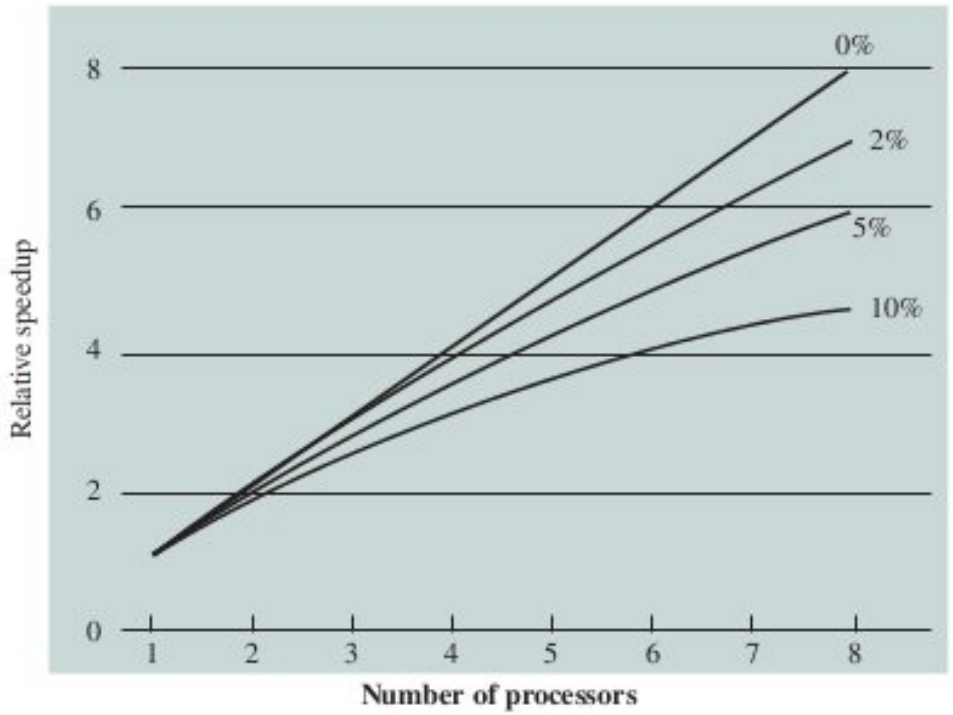
- A part which cannot be parallelized (inherently serial)
- A part which can be parallelized (infinitely parallelizable)

If f is the portion of the application that involves code that is parallelizable with no scheduling overhead. Then (1-f) it's the faction of the execution time involves code that is serial.

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} = \frac{1}{(1-f) + \frac{f}{N}}$$

Amdahl's Law (1967) - Performance of Software on Multicore

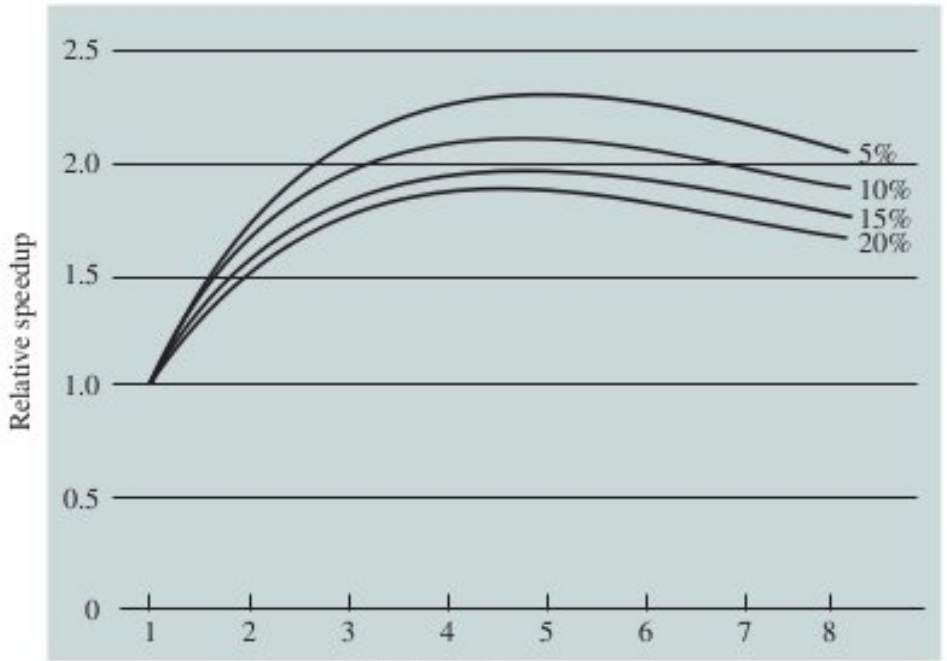
As an example, assume we have an application that is 90 % percent parallel and 10 % percent serial. If we run this application on a system with eight procesors cores, we can get a speedup of 4.7 times.



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

Amdahl's Law (1967) - Performance of Software on Multicore

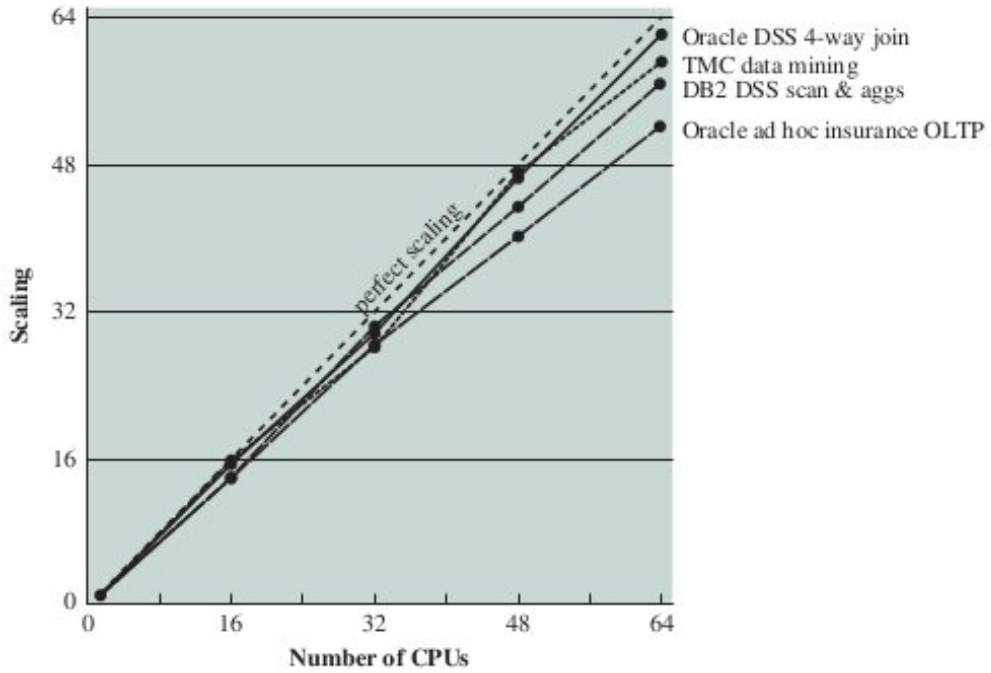
Software typically incurs overhead as a result of communication and distribution of work to multiple processors and cache coherence overhead.



(b) Speedup with overheads

Amdahl's Law (1967) - Performance of Software on Multicore

However, therea are numerous applications in which it is possible to effectively exploit a multicore system.



Multithreading Models

There are two types of threads to be managed in a modern system: User threads and 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 OSes support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

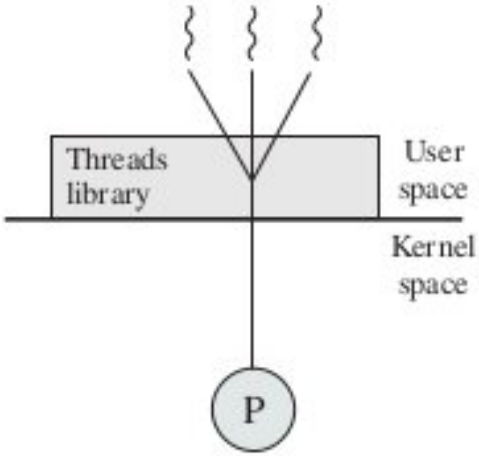
  In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies.

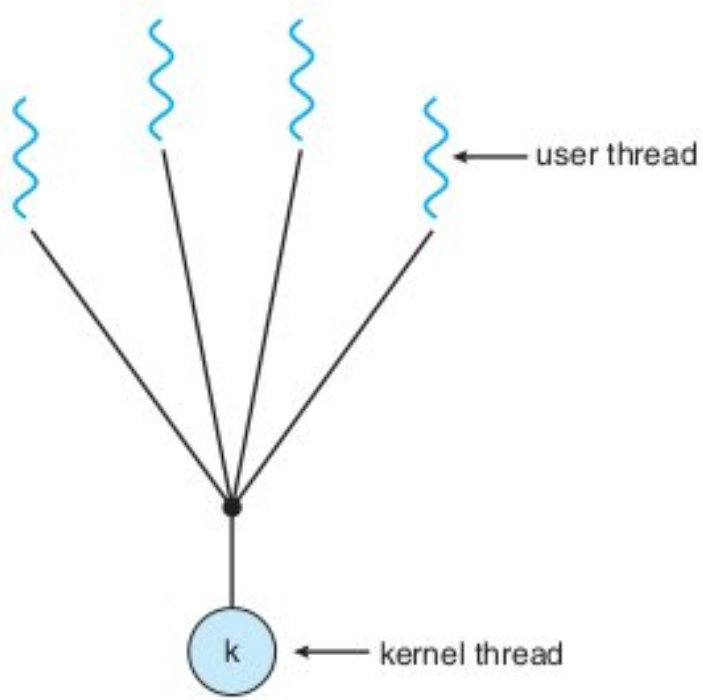
Many-To-One Model

- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would otherwise be able to continue. Because a single kernel thread can operate only on a single CPU.
- The many-to-one model does not allow individual processes to be split across multiple CPUs.

Green threads for Solaris and GNU Portable Threads implement the many-to-one model in the past, but few systems continue to do so today.

Many-To-One Model



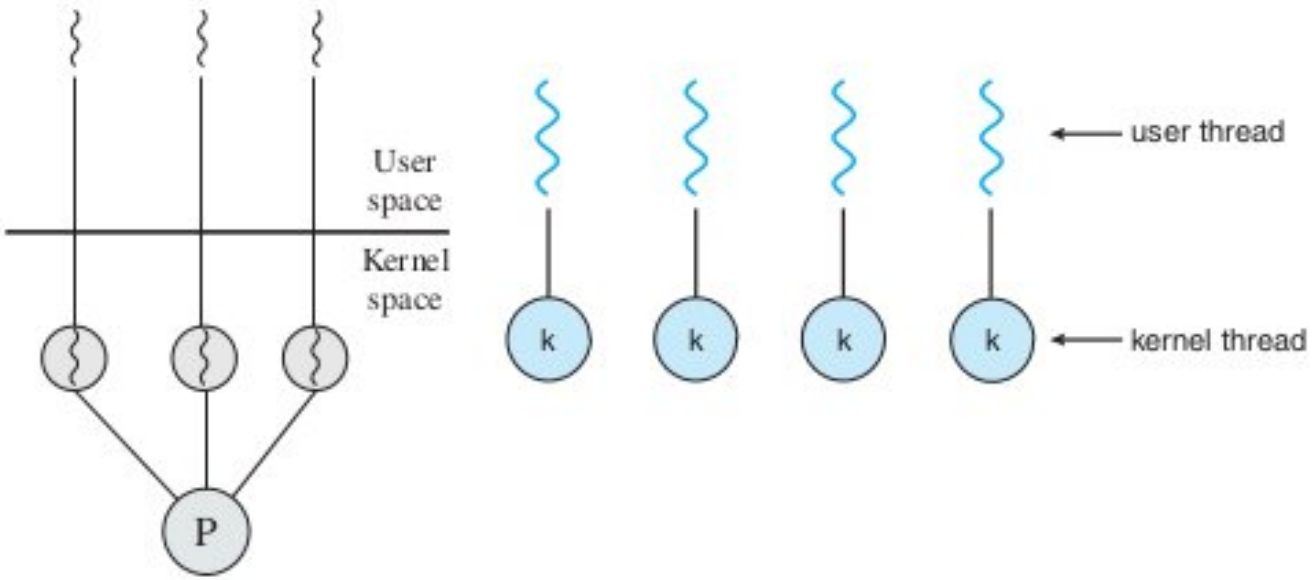


One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each user thread.
- One-to-one model overcomes the problems of the blocking system calls and the splitting of processes across multiple CPUs.
- However the overhead of managing the one-to-one model is more significant, involving more overhead and slowing down the system.
- Most implementations of this model place a limit on how many threads can be created.

Linux, along with the family of Windows operating systems implement the one-to-one model for threads.

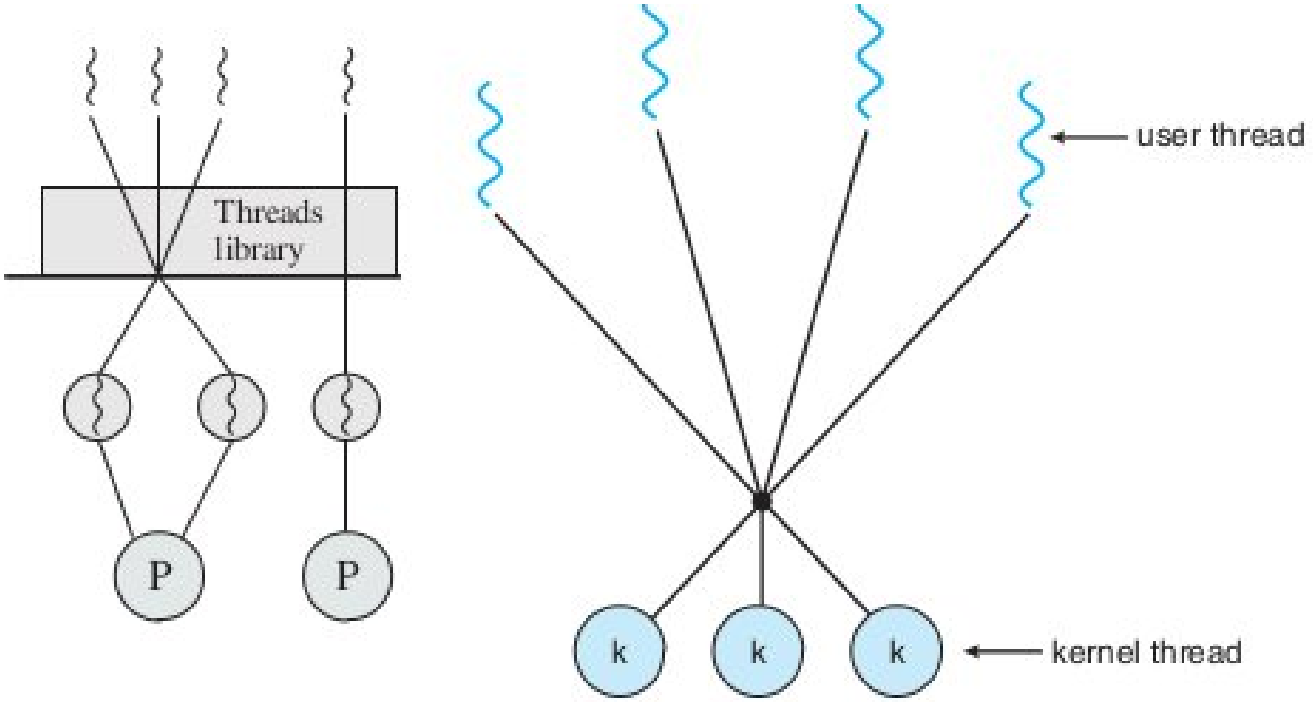
One-To-One Model



Many-To-Many Model

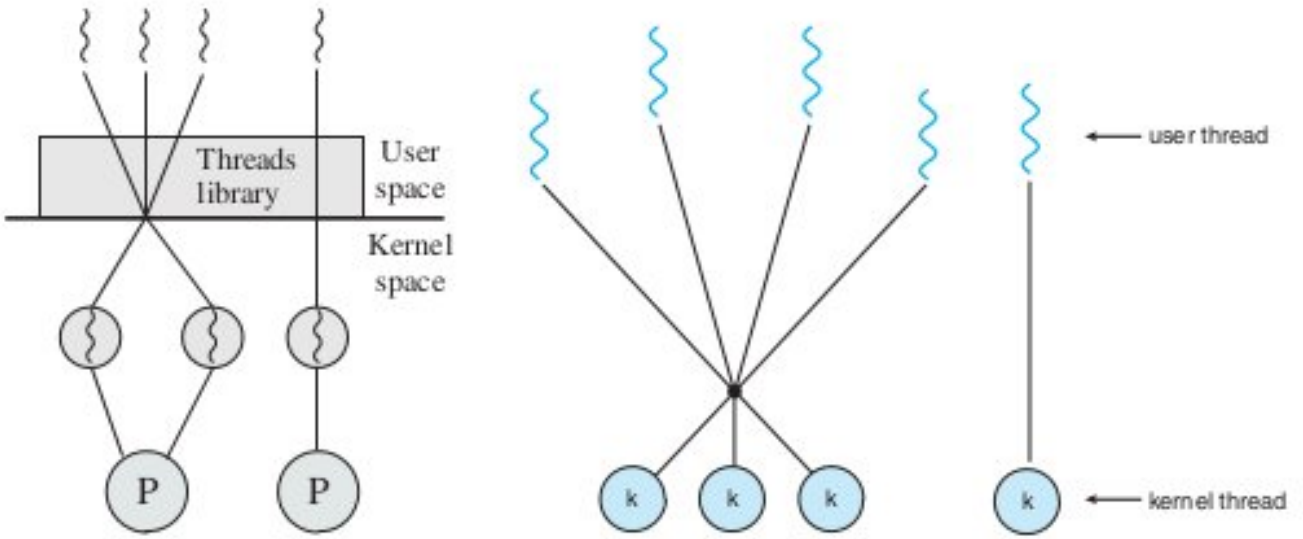
- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users have no restrictions on the number of threads created.
- Blocking kernel system calls do not block the entire process.
- Processes can be split across multiple processors.
- Individual processes may be allocated variable numbers of kernel threads, depending on the number of CPUs present and other factors.

Many-To-Many Model



Many-To-Many Model

- One popular variation of the many-to-many model is the two-tier model, which allows either many-to-many or one-to-one operation.
- IRIX, HP-UX, and Tru64 UNIX use the two-tier model, as did Solaris prior to Solaris 9.



Threads States

The key states for a thread are Running, Ready and Blocked.

Why not suspend?

Threads States

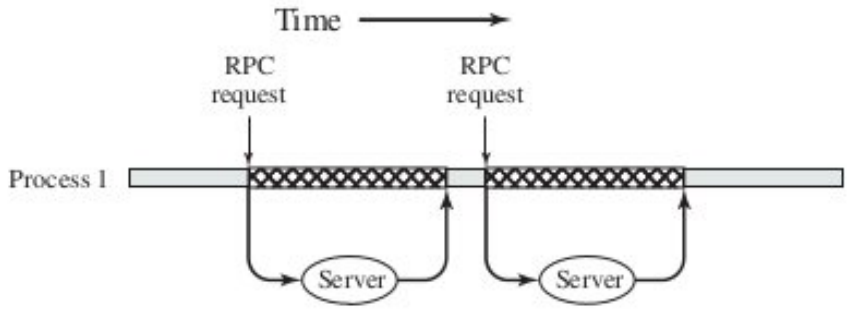
The key states for a thread are Running, Ready and Blocked.

Why not suspend? Because, Suspended states are more related to processes.

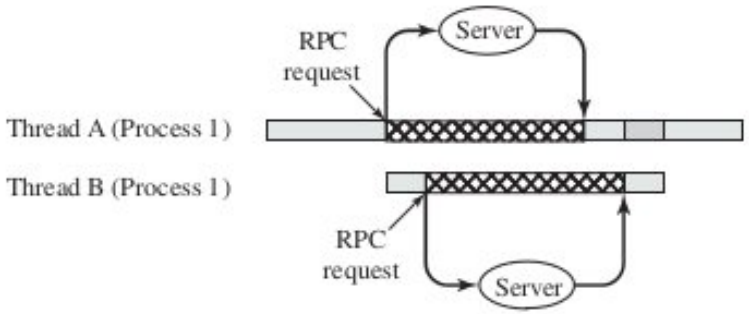
Basic thread operations associated with a change in thread state:

- ***Spawn***, a spawned thread could create another thread
- ***Block***, one thread is blocked because an I/O operation but others can be ready to run
- ***Unblock***, an event unblocks a blocked thread
- ***Finish***, resources allocated to a finished threat are then deallocated

RPC using threads



(a) RPC using single thread



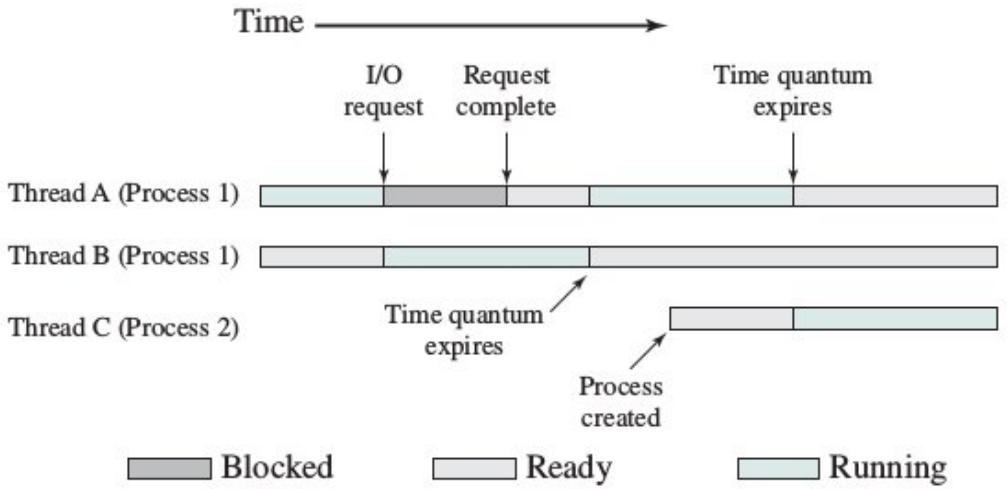
(b) RPC using one thread per server (on a uniprocessor)

Blocked, waiting for response to RPC

Blocked, waiting for processor, which is in use by Thread B

Running

RPC using threads



User threads - Activity

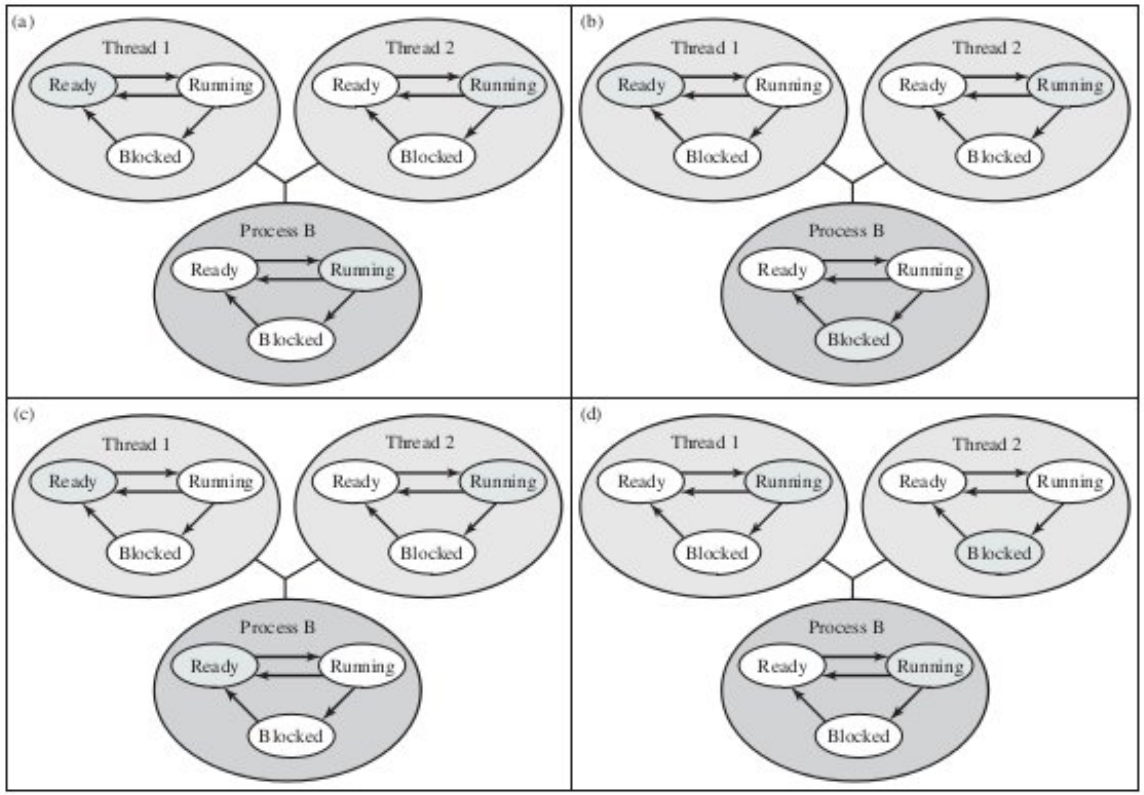
All of the work of thread management is done by the application and the kernel is not aware of the existence of threads.

- By default, an application begins with a single thread.
- This application and its thread are allocated to a single process management by the kernel.
- At any time, the application may spawn a new thread to run within the same process.
- Spawning is done invoking the thread library.

User threads - Activity

- The control is passed to that library. The library creates a data structure for the new thread and then passes control to one the threads, using some scheduling algorithm.
- When control is passed to the library, the context of the current thread is saved.
- When control is passed from the library to a thread, the context of that thread is restored.
- Context: user registers, pc and stack pointers.
- The kernel is unaware of this activity.

Relationship between User Thread states and Process states



Kernel threads

There is no thread management code in the application level.
Simply an application programing interface (API) to the kernel.

- The kernel maintains context for the process as a whole and for individual threads withing the process.
- The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
- Or, if one thread in a process is blocked, the kernel can schedule another thread of the same process.
- Transfer of control from one thread to another within the same process requires a mode switch to the kernel.

Kernel threads and process operation latencies

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

However, this depends on the nature of the applications involved. If most of the thread switches in an application require kernel mode access, then a User thread scheme may not perform much better than a Kernel thread schema.

Other arrangements

Threads: Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Thread synchronization

- All threads share the same address space y other resources such as open files.
- A given thread does a modification then that modification can be seen by other threads
- Synchronization mechanisms are needed for experimenting an ordered access to shared resources

Thread API

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 (user space and not a system call).
- The second approach is to implement a kernel-level library supported directly by the operating system (system call to the kernel).

Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java.

What are pthreads?

There are two primary ways of implementing a thread library:

- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
- For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
- Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary API's.

pthreads vs process

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

pthreads vs process

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Thread Creation - pthread_create

```
#include <pthread.h>
int
pthread_create(      pthread_t *          thread,
               const pthread_attr_t *     attr,
                     void *               (*start_routine)(void*),
                     void *               arg);
```

- The first, thread, is a pointer to a structure of type pthread t; we'll use this structure to interact with this thread, and thus we need to pass it to pthread create() in order to initialize it.
- The second argument, attr, is used to specify any attributes this thread might have. Some examples include setting the stack size or perhaps in- formation about the scheduling priority of the thread.

Thread Creation - pthread_create

```
#include <pthread.h>
int
pthread_create(      pthread_t *          thread,
               const pthread_attr_t *     attr,
                     void *               (*start_routine)(void*),
                     void *               arg);
```

- The third argument is the most complex, but is really just asking: which function should this thread start running in? In C, we call this a function pointer.
- the fourth argument, arg, is exactly the argument to be passed to the function where the thread begins execution.

Thread Completion - pthread_join

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.
- The first is of type pthread_t, and is used to specify which thread to wait for.
- The second argument is a pointer to the return value you expect to get back.

Note: Never return a pointer which refers to something allocated on the thread's call stack.

Thread Completion - pthread_detach

The pthread_detach() function marks the thread identified by thread as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

Thread ID pthread_self

The pthread_self() function returns the ID of the calling thread. This is the same value that is returned in *thread in the pthread_create(3) call that created this thread.

Terminating Threads - pthread_exit

There are several ways in which a thread may be terminated:

- The thread returns normally from its starting routine. Its work is done.
- The thread makes a call to the pthread_exit subroutine - whether its work is done or not.
- The thread is canceled by another thread via the pthread_cancel routine.
- The entire process is terminated due to making a call to either the exec() or exit().
- If main() finishes first, without calling pthread_exit explicitly itself

Terminating Threads - pthread_cancel

The pthread_cancel() function sends a cancellation request to the thread thread. If a thread is cancelable, its cancelability type is deferred, and a cancellation request is pending for the thread, then the thread is canceled when it calls a function that is a cancellation point.

Examples and Exercises

Examples and Exercises