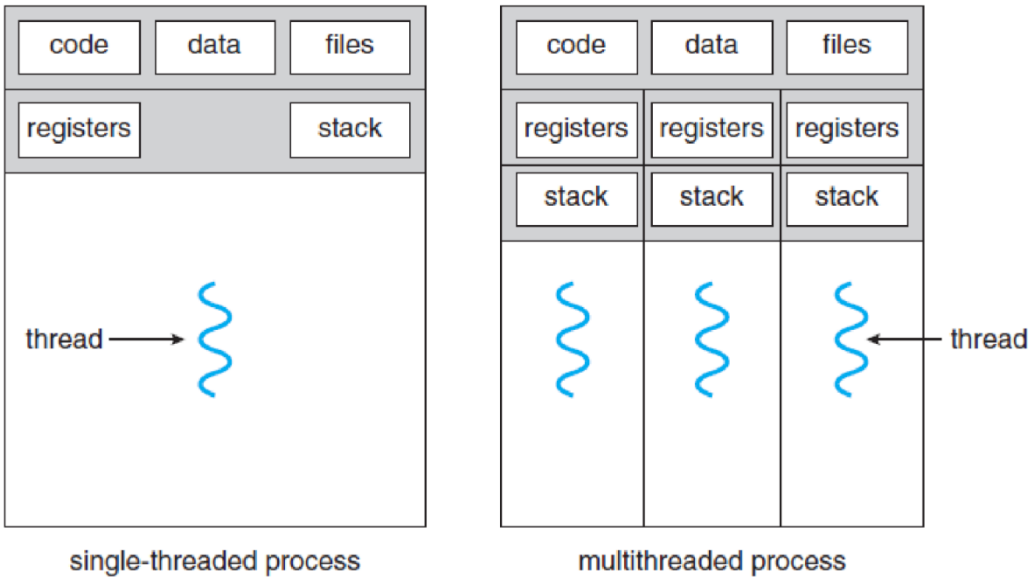
2.7 THREADS

2.7.1 Overview

A **thread** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)

- Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- As shown in Figure multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.

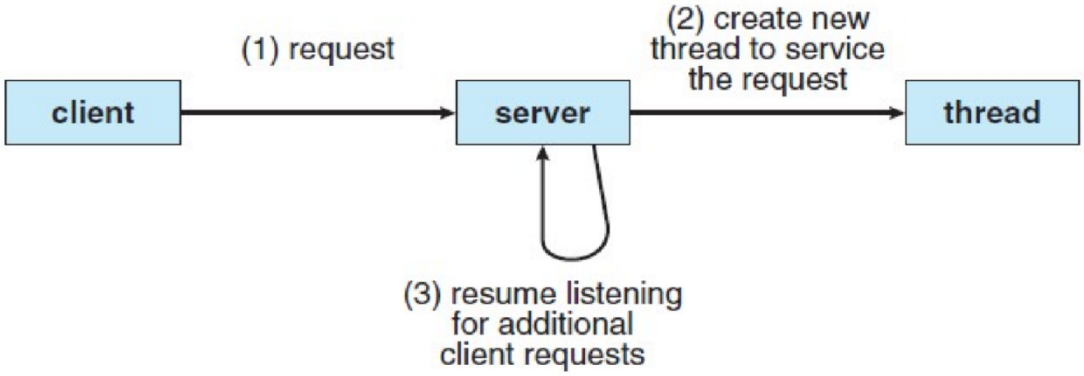


Single-threaded and multithreaded processes

2.7.1.1 Motivation

- Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others.
- This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.
- For example in a word processor, a background thread may check spelling and grammar while a foreground thread processes user input (keystrokes), while yet a third thread loads images from the hard drive, and a fourth does periodic automatic backups of the file being edited.
- Another example is a web server - Multiple threads allow for multiple requests to be satisfied simultaneously, without having to service requests sequentially or to fork

off separate processes for every incoming request. (The latter is how this sort of thing was done before the concept of threads was developed. A daemon would listen at a port, fork off a child for every incoming request to be processed, and then go back to listening to the port.)



Multithreaded server architecture

2.7.1.2 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.

2. 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.

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

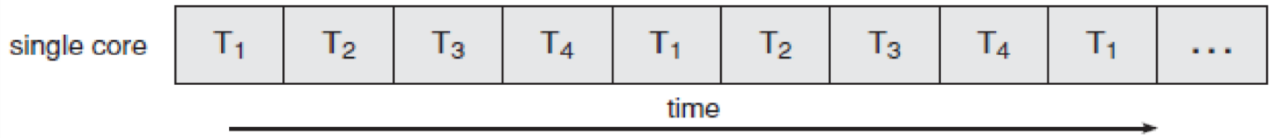
4. Scalability, i.e. Utilization of multiprocessor architectures - A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of a multi-threaded application may be split amongst available processors.

(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.)

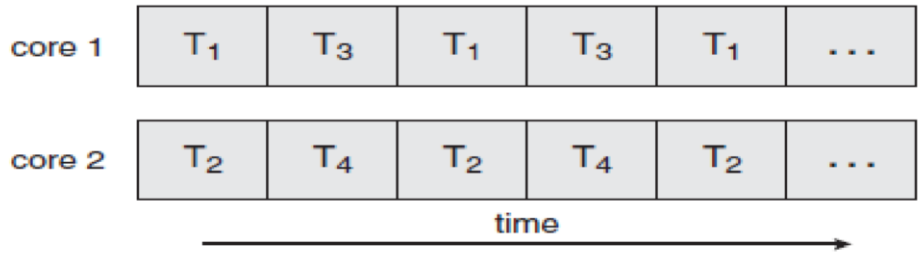
2.7.2 Multicore Programming

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



Concurrent execution on a single-core system.



Parallel execution on a multicore system

- 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.

2.7.2.1 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.

2.7.2.2 Types of Parallelism

In theory there are two different ways to parallelize the workload:

- **Data parallelism** divides the data up amongst multiple cores (threads), and performs the same task on each subset of the data. For example dividing a large image up into pieces and performing the same digital image processing on each piece on different cores.
- **Task parallelism** divides the different tasks to be performed among the different cores and performs them simultaneously.

In practice no program is ever divided up solely by one or the other of these, but instead by some sort of hybrid combination.

2.7.3 Multithreading Models

There are two types of threads to be managed in a modern system: User threads and kernel threads.

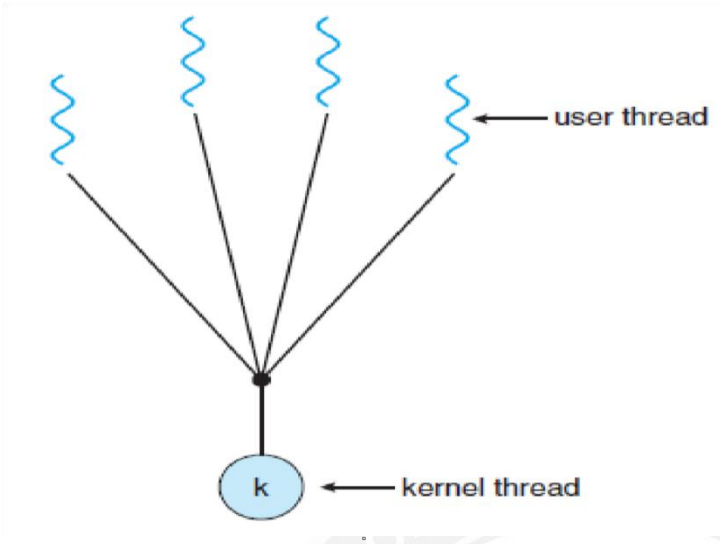
User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs.

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.

2.7.3.1 Many-To-One Model

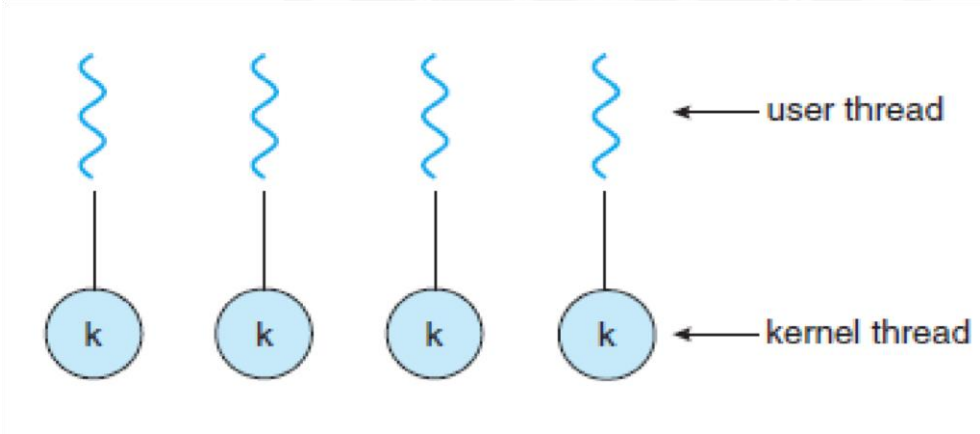
- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.
- Thread management is handled by the thread library in user space, which is very efficient.
- However, if a blocking system call is made, then the entire process blocks, even if the other user threads would 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

2.7.3.2 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 listed above involving 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 and Windows from 95 to XP implement the one-to-one model for threads.

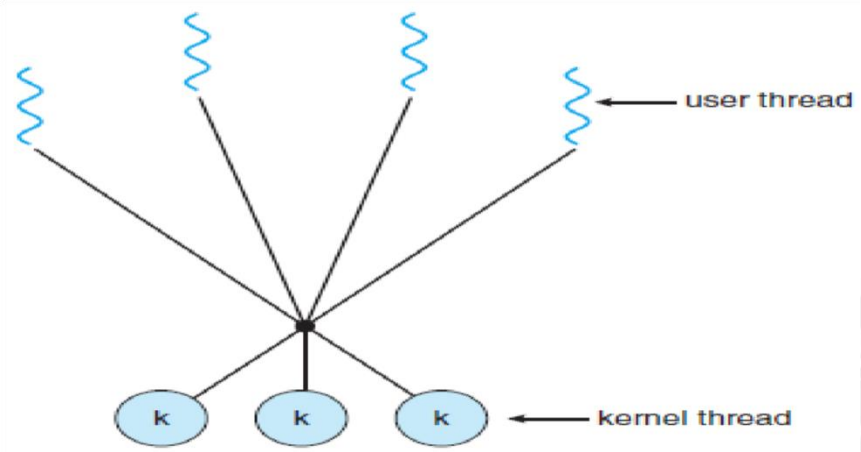


One-to-one model

2.7.3.3 Many-To-Many Model

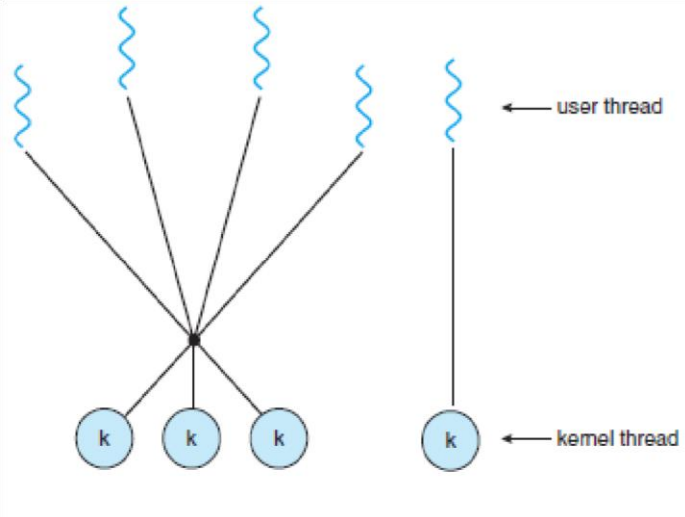
- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to one and many-to-one models.
- Users have no restrictions on the number of threads created.

- 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

- 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.



Two-level model

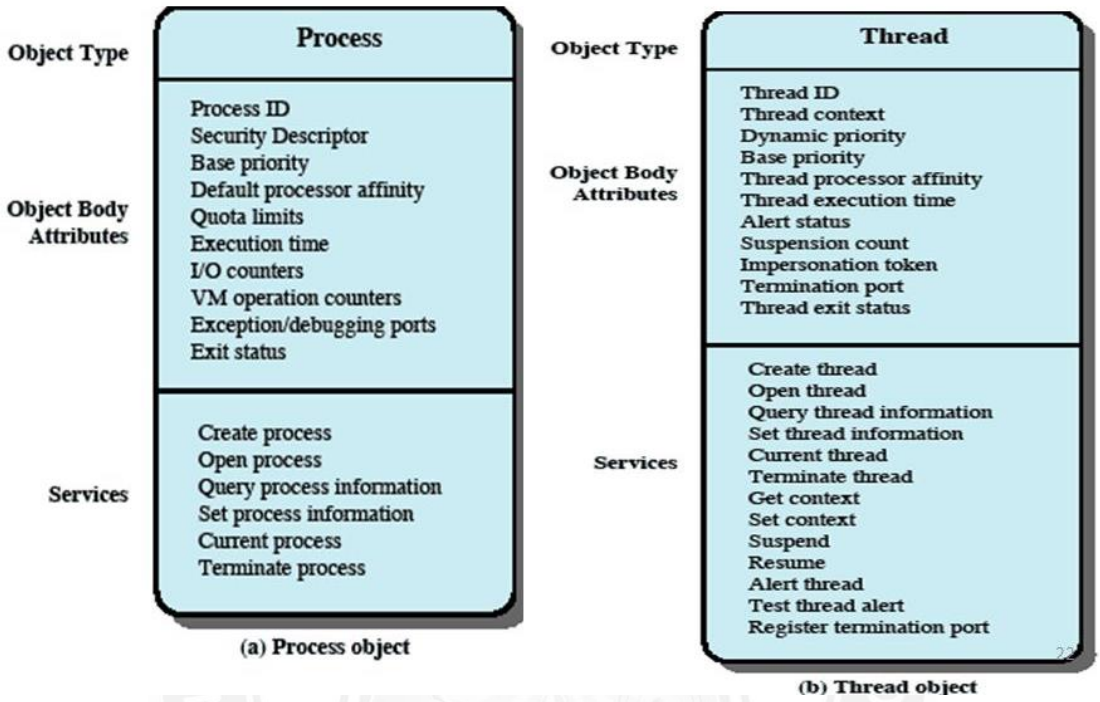
2.7.4 Windows 7 Threads and SMP Management

Windows process design is driven by the need to provide support for a variety of OS environments. Processes supported by different OS environments differ in a number of ways, including the following:

- How processes are named
- Whether threads are provided within processes
- How processes are represented
- How process resources are protected
- What mechanisms are used for interprocess communication and synchronization
- How processes are related to each other

Important characteristics of Windows processes are the following:

- Windows processes are implemented as objects.
- A process can be created as new process, or as a copy of an existing process.
- An executable process may contain one or more threads.
- Both process and thread objects have built-in synchronization capabilities.
- Figure illustrates the way in which a process relates to the resources it controls or uses.
- Each process is assigned a security access

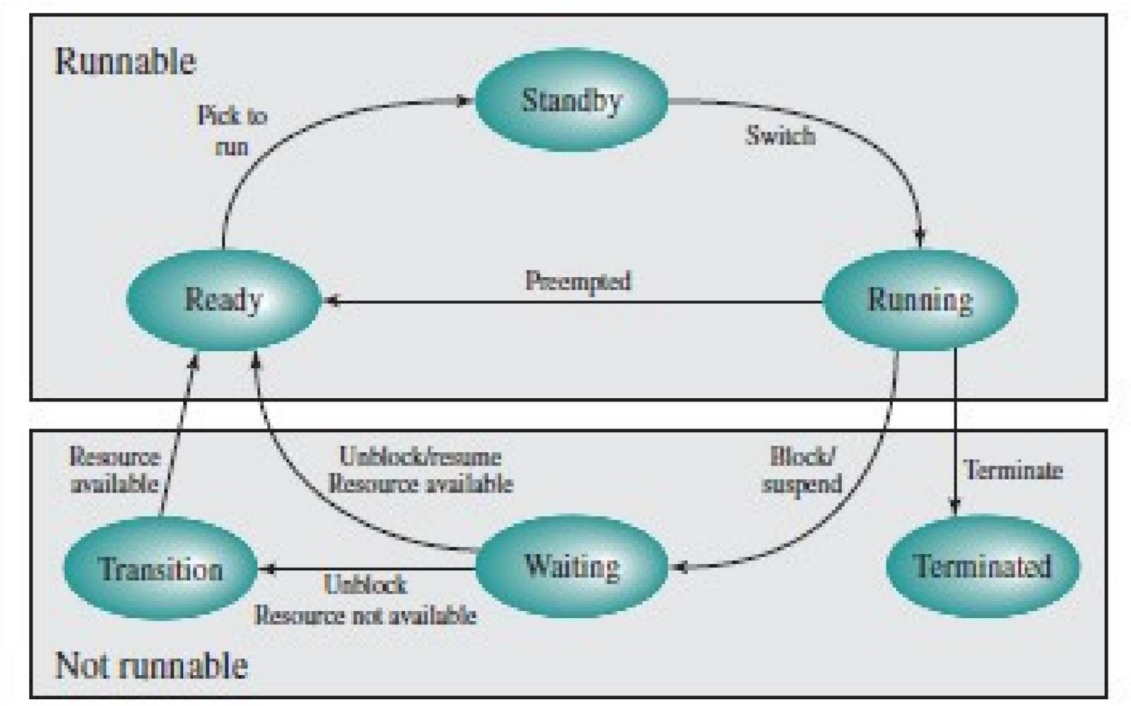


2.7.4.1 Thread States

An existing Windows thread is in one of six states

- **Ready:** A ready thread may be scheduled for execution. The Kernel dispatcher keeps track of all ready threads and schedules them in priority order.
- **Standby:** A standby thread has been selected to run next on a particular processor. The thread waits in this state until that processor is made available. If the standby thread's priority is high enough, the running thread on that processor may be preempted in favor of the standby thread. Otherwise, the standby thread waits until the running thread blocks or exhausts its time slice.
- **Running:** Once the Kernel dispatcher performs a thread switch, the standby thread enters the Running state and begins execution and continues execution until it is preempted by a higher priority thread, exhausts its time slice, blocks, or terminates. In the first two cases, it goes back to the Ready state.
- **Waiting:** A thread enters the Waiting state when (1) it is blocked on an event (e.g., I/O), (2) it voluntarily waits for synchronization purposes, or (3) an environment subsystem directs the thread to suspend itself. When the waiting condition is satisfied, the thread moves to the Ready state if all of its resources are available.

- **Transition:** A thread enters this state after waiting if it is ready to run but the resources are not available. For example, the thread's stack may be paged out of memory. When the resources are available, the thread goes to the Ready state.
- **Terminated:** A thread can be terminated by itself, by another thread, or when its parent process terminates. Once housekeeping chores are completed, the thread is removed from the system, or it may be retained by the Executive 6 for future reinitialization.



2.7.4.2 Symmetric Multiprocessing Support

- Windows supports SMP hardware configurations. The threads of any process, including those of the executive, can run on any processor.
- In the absence of affinity restrictions, explained in the next paragraph, the kernel dispatcher assigns a ready thread to the next available processor.
- This assures that no processor is idle or is executing a lower-priority thread when a higher priority thread is ready.
- Multiple threads from the same process can be executing simultaneously on multiple processors.
- As a default, the kernel dispatcher uses the policy of **soft affinity** in assigning threads to processors:
- The dispatcher tries to assign a ready thread to the same processor it last ran on.

- This helps reuse data still in that processor's memory caches from the previous execution of the thread.
- It is possible for an application to restrict its thread execution only to certain processors (**hard affinity**).