The basic idea is that the several components in any complex system will perform particular subfunctions that contribute to the overall function.

—THE SCIENCES OF THE ARTIFICIAL,
Herbert Simon
Processes and Threads

*Processes have two characteristics:

**Resource Ownership**

- Process includes a virtual address space to hold the process image
  - the OS performs a protection function to prevent unwanted interference between processes with respect to resources

**Scheduling/Execution**

- Follows an execution path that may be interleaved with other processes
  - a process has an execution state (Running, Ready, etc.) and a dispatching priority and is scheduled and dispatched by the OS
Processes and Threads

- The unit of dispatching is referred to as a \textit{thread} or \textit{lightweight process}

- The unit of resource ownership is referred to as a \textit{process} or \textit{task}

- \textbf{Multithreading} - The ability of an OS to support multiple, concurrent paths of execution within a single process
Single Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach.

- MS-DOS is an example.

Figure 4.1  Threads and Processes [ANDE97]
Multithreaded Approaches

- The right half of Figure 4.1 depicts multithreaded approaches.

- A Java run-time environment is an example of a system of one process with multiple threads.

Figure 4.1  Threads and Processes [ANDE97]
Processes

- The unit or resource allocation and a unit of protection
- A virtual address space that holds the process image
- Protected access to:
  - processors
  - other processes
  - files
  - I/O resources
One or More Threads in a Process

Each thread has:

• an execution state (Running, Ready, etc.)
• saved thread context when not running
• an execution stack
• some per-thread static storage for local variables
• access to the memory and resources of its process (all threads of a process share this)
Threads vs. Processes

Figure 4.2 Single Threaded and Multithreaded Process Models
Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs
Thread Use in a Single-User System

- Foreground and background work
- Asynchronous processing
- Speed of execution
- Modular program structure
Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
  - suspending a process involves suspending all threads of the process
  - termination of a process terminates all threads within the process
**Thread Execution States**

The key states for a thread are:
- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:
- Spawn
- Block
- Unblock
- Finish
RPC Using Single Thread

(a) RPC Using Single Thread
RPC Using One Thread per Server

(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
Multithreading on a Uniprocessor

Figure 4.4 Multithreading Example on a Uniprocessor
Thread Synchronization

- It is necessary to synchronize the activities of the various threads
- all threads of a process share the same address space and other resources
- any alteration of a resource by one thread affects the other threads in the same process
Types of Threads

User Level Thread (ULT)

Kernel level Thread (KLT)
User-Level Threads (ULTs)

- All thread management is done by the application.
- The kernel is not aware of the existence of threads.
Relationships Between ULT States and Process States

Figure 4.6 Examples of the Relationships between User-Level Thread States and Process States
Advantages of ULTs

- Thread switching does not require kernel mode privileges
- Scheduling can be application specific
- ULTs can run on any OS
Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - as a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked

- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
Overcoming ULT Disadvantages

Jacketing

• converts a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads
Kernel-Level Threads (KLTs)

- Thread management is done by the kernel
- no thread management is done by the application
- Windows is an example of this approach
Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- Kernel routines can be multithreaded.
Disadvantage of KLTs

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel.

<table>
<thead>
<tr>
<th>Operation</th>
<th>User-Level Threads</th>
<th>Kernel-Level Threads</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Fork</td>
<td>34</td>
<td>948</td>
<td>11,300</td>
</tr>
<tr>
<td>Signal Wait</td>
<td>37</td>
<td>441</td>
<td>1,840</td>
</tr>
</tbody>
</table>

Table 4.1  Thread and Process Operation Latencies (μs)
Combined Approaches

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- Solaris is an example
# Relationship Between Threads and Processes

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

Table 4.2  Relationship between Threads and Processes
Performance Effect of Multiple Cores

Figure 4.7 (a) Speedup with 0%, 2%, 5%, and 10% sequential portions

Figure 4.7 (b) Speedup with overheads
Database Workloads on Multiple-Processor Hardware

Figure 4.8 Scaling of Database Workloads on Multiple Processor Hardware
Applications That Benefit

◆ Multithreaded native applications
  ◆ characterized by having a small number of highly threaded processes

◆ Multiprocess applications
  ◆ characterized by the presence of many single-threaded processes

◆ Java applications

◆ Multiinstance applications
  ◆ multiple instances of the application in parallel
Processes and services provided by the Windows Kernel are relatively simple and general purpose

- implemented as objects
- created as new process or a copy of an existing
- an executable process may contain one or more threads
- both processes and thread objects have built-in synchronization capabilities
Relationship Between Process and Resource

Figure 4.10  A Windows Process and Its Resources

Figure 4.12  A Windows Process and Its Resources
Windows makes use of two types of process-related objects:

**Processes**
- an entity corresponding to a user job or application that owns resources

**Threads**
- a dispatchable unit of work that executes sequentially and is interruptible
Windows Process and Thread Objects

(a) Process object

Object Type
- Process ID
- Security Descriptor
- Base priority
- Default processor affinity
- Quota limits
- Execution time
- I/O counters
- VM operation counters
- Exception/debugging ports
- Exit status

Object Body Attributes
- Create process
- Open process
- Query process information
- Set process information
- Current process
- Terminate process

Services
- Create thread
- Open thread
- Query thread information
- Set thread information
- Current thread
- Terminate thread
- Get context
- Set context
- Suspend
- Resume
- Alert thread
- Test thread alert
- Register termination port

(b) Thread object
# Windows Process Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process ID</td>
<td>A unique value that identifies the process to the operating system.</td>
</tr>
<tr>
<td>Security descriptor</td>
<td>Describes who created an object, who can gain access to or use the object, and who is denied access to the object.</td>
</tr>
<tr>
<td>Base priority</td>
<td>A baseline execution priority for the process's threads.</td>
</tr>
<tr>
<td>Default processor affinity</td>
<td>The default set of processors on which the process's threads can run.</td>
</tr>
<tr>
<td>Quota limits</td>
<td>The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.</td>
</tr>
<tr>
<td>Execution time</td>
<td>The total amount of time all threads in the process have executed.</td>
</tr>
<tr>
<td>I/O counters</td>
<td>Variables that record the number and type of I/O operations that the process's threads have performed.</td>
</tr>
<tr>
<td>VM operation counters</td>
<td>Variables that record the number and types of virtual memory operations that the process's threads have performed.</td>
</tr>
<tr>
<td>Exception/debugging ports</td>
<td>Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally, these are connected to environment subsystem and debugger processes, respectively.</td>
</tr>
<tr>
<td>Exit status</td>
<td>The reason for a process's termination.</td>
</tr>
</tbody>
</table>

Table 4.3 Windows Process Object Attributes
## Windows Thread Object Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread ID</td>
<td>A unique value that identifies a thread when it calls a server.</td>
</tr>
<tr>
<td>Thread context</td>
<td>The set of register values and other volatile data that defines the execution state of a thread.</td>
</tr>
<tr>
<td>Dynamic priority</td>
<td>The thread's execution priority at any given moment.</td>
</tr>
<tr>
<td>Base priority</td>
<td>The lower limit of the thread's dynamic priority.</td>
</tr>
<tr>
<td>Thread processor affinity</td>
<td>The set of processors on which the thread can run, which is a subset or all of the processor affinity of the thread's process.</td>
</tr>
<tr>
<td>Thread execution time</td>
<td>The cumulative amount of time a thread has executed in user mode and in kernel mode.</td>
</tr>
<tr>
<td>Alert status</td>
<td>A flag that indicates whether a waiting thread may execute an asynchronous procedure call.</td>
</tr>
<tr>
<td>Suspension count</td>
<td>The number of times the thread's execution has been suspended without being resumed.</td>
</tr>
<tr>
<td>Impersonation token</td>
<td>A temporary access token allowing a thread to perform operations on behalf of another process (used by subsystems).</td>
</tr>
<tr>
<td>Termination port</td>
<td>An interprocess communication channel to which the process manager sends a message when the thread terminates (used by subsystems).</td>
</tr>
<tr>
<td>Thread exit status</td>
<td>The reason for a thread's termination.</td>
</tr>
</tbody>
</table>

Table 4.4 Windows Thread Object Attributes
Multithreaded Process

Achieves concurrency without the overhead of using multiple processes

Threads within the same process can exchange information through their common address space and have access to the shared resources of the process

Threads in different processes can exchange information through shared memory that has been set up between the two processes
Thread States

Figure 4.12 Windows Thread States
Symmetric Multiprocessing Support (SMP)

Threads of any process can run on any processor

Soft Affinity
- the dispatcher tries to assign a ready thread to the same processor it last ran on
- helps reuse data still in that processor’s memory caches from the previous execution of the thread

Hard Affinity
- an application restricts thread execution to certain processors
Solaris Process

makes use of four thread-related concepts:

- **Process**
  - includes the user’s address space, stack, and process control block

- **User-level Threads**
  - a user-created unit of execution within a process

- **Lightweight Processes (LWP)**
  - a mapping between ULTs and kernel threads

- **Kernel Threads**
  - fundamental entities that can be scheduled and dispatched to run on one of the system processors
Processes and Threads in Solaris

Figure 4.13  Processes and Threads in Solaris [MCDO07]
Traditional Unix vs Solaris

UNIX Process Structure

<table>
<thead>
<tr>
<th>Process ID</th>
<th>User IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal Dispatch Table</td>
<td>Memory Map</td>
</tr>
<tr>
<td>Priority</td>
<td>Signal Mask</td>
</tr>
<tr>
<td>Registers</td>
<td>STACK</td>
</tr>
<tr>
<td>File Descriptors</td>
<td>Processor State</td>
</tr>
</tbody>
</table>

Solaris Process Structure

<table>
<thead>
<tr>
<th>Process ID</th>
<th>User IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal Dispatch Table</td>
<td>Memory Map</td>
</tr>
<tr>
<td>Priority</td>
<td>Signal Mask</td>
</tr>
<tr>
<td>Registers</td>
<td>STACK</td>
</tr>
<tr>
<td>File Descriptors</td>
<td>Processor State</td>
</tr>
</tbody>
</table>

Figure 4.14 Process Structure in Traditional UNIX and Solaris [LEWI96]
A Lightweight Process (LWP) Data Structure Includes:

- An LWP identifier
- The priority of this LWP
- A signal mask
- Saved values of user-level registers
- The kernel stack for this LWP
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure
Solaris Thread States

Figure 4.15 Solaris Thread States
Interrupts as Threads

Most operating systems contain two fundamental forms of concurrent activity:

Processes (threads)
- cooperate with each other and manage the use of shared data structures by primitives that enforce mutual exclusion and synchronize their execution

Interrupts
- synchronized by preventing their handling for a period of time
Solaris Solution

• Solaris employs a set of kernel threads to handle interrupts
  ■ an interrupt thread has its own identifier, priority, context, and stack
  ■ the kernel controls access to data structures and synchronizes among interrupt threads using mutual exclusion primitives
  ■ interrupt threads are assigned higher priorities than all other types of kernel threads
Linux Tasks

A process, or task, in Linux is represented by a task_struct data structure.

This structure contains information in a number of categories.
Linux

Process/Thread Model

Figure 4.16 Linux Process/Thread Model
Linux Threads

- Linux does not recognize a distinction between threads and processes.
- A new process is created by copying the attributes of the current process.
- User-level threads are mapped into kernel-level processes.
- The new process can be cloned so that it shares resources.
- The clone() call creates separate stack spaces for each process.
Linux
Clone ()
Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE_CLEARID</td>
<td>Clear the task ID.</td>
</tr>
<tr>
<td>CLONE_DETACHED</td>
<td>The parent does not want a SIGCHLD signal sent on exit.</td>
</tr>
<tr>
<td>CLONE_FILES</td>
<td>Shares the table that identifies the open files.</td>
</tr>
<tr>
<td>CLONE_FS</td>
<td>Shares the table that identifies the root directory and the current working directory, as well as the value of the bit mask used to mask the initial file permissions of a new file.</td>
</tr>
<tr>
<td>CLONE_IDLETASK</td>
<td>Set PID to zero, which refers to an idle task. The idle task is employed when all available tasks are blocked waiting for resources.</td>
</tr>
<tr>
<td>CLONE_NEWNS</td>
<td>Create a new namespace for the child.</td>
</tr>
<tr>
<td>CLONE_PARENT</td>
<td>Caller and new task share the same parent process.</td>
</tr>
<tr>
<td>CLONE_PTRACE</td>
<td>If the parent process is being traced, the child process will also be traced.</td>
</tr>
<tr>
<td>CLONE_SETTID</td>
<td>Write the TID back to user space.</td>
</tr>
<tr>
<td>CLONE_SETTLS</td>
<td>Create a new TLS for the child.</td>
</tr>
<tr>
<td>CLONE_SIGHAND</td>
<td>Shares the table that identifies the signal handlers.</td>
</tr>
<tr>
<td>CLONE_SYVSEM</td>
<td>Shares System V SEM_UNDO semantics.</td>
</tr>
<tr>
<td>CLONE_THREAD</td>
<td>Inserts this process into the same thread group of the parent. If this flag is true, it implicitly enforces CLONE_PARENT.</td>
</tr>
<tr>
<td>CLONE_VFORK</td>
<td>If set, the parent does not get scheduled for execution until the child invokes the <code>execve()</code> system call.</td>
</tr>
<tr>
<td>CLONE_VM</td>
<td>Shares the address space (memory descriptor and all page tables).</td>
</tr>
</tbody>
</table>
Mac OS X Grand Central Dispatch (GCD)

- Provides a pool of available threads
- Designers can designate portions of applications, called *blocks*, that can be dispatched independently and run concurrently
- Concurrency is based on the number of cores available and the thread capacity of the system
Block

- A simple extension to a language
- A block defines a self-contained unit of work
- Enables the programmer to encapsulate complex functions
- Scheduled and dispatched by queues
- Dispatched on a first-in-first-out basis
- Can be associated with an event source, such as a timer, network socket, or file descriptor
Summary

- User-level threads
  - created and managed by a threads library that runs in the user space of a process
  - a mode switch is not required to switch from one thread to another
  - only a single user-level thread within a process can execute at a time
  - if one thread blocks, the entire process is blocked

- Kernel-level threads
  - threads within a process that are maintained by the kernel
  - a mode switch is required to switch from one thread to another
  - multiple threads within the same process can execute in parallel on a multiprocessor
  - blocking of a thread does not block the entire process

- Process/related to resource ownership

- Thread/related to program execution