Processes and Threads

Process traditionally considered as embodying two distinct characteristics:

Resource ownership

- process includes a virtual address space to hold the process image
- usually now thought of as a *process* or *task*

Scheduling/execution

- follows an execution path that may be interleaved with other processes
- usually now referred to as a *thread* or *lightweight process*

These two characteristics are treated independently by the operating system

Thread

- an execution state (running, ready, etc.)
- saved thread context when not running
- has 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

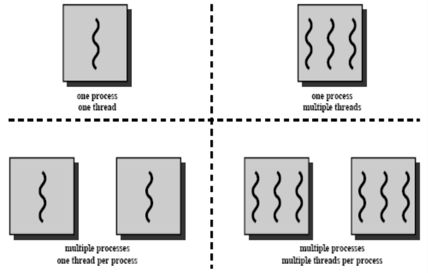
Multithreading

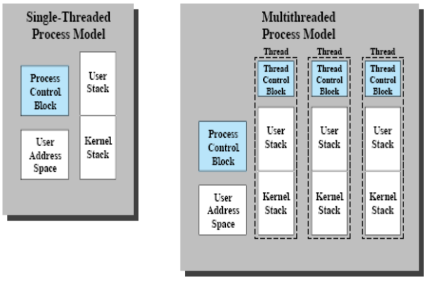
definition: operating system supports multiple threads of execution within a single process

MS-DOS supports a single thread

UNIX supports multiple user processes but only supports one thread per process

(modern) Windows, Solaris, Linux, Mach, and OS/2 support multiple threads





Threads: Benefits and Issues

Takes less time to create a new thread than a new process

Less time to terminate a thread than a process

Less time to switch between two threads within the same process

Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

Suspending a process involves suspending all threads of the process since all threads share the same address space

Termination of a process, terminates all threads within the process

Threads in a Single-User Multiprocessing System

Foreground to background work

In a spreadsheet program, one thread could display menus and read user input while another thread executes user commands and updates the spreadsheet.

Improves perceived speed of the application.

Asynchronous processing

A word processor may write its RAM buffer to disk once every minute, via a self-contained thread that runs without further supervision from the rest of the process.

Speed of execution

One thread can compute results from one batch of data while another thread retrieves the next batch of data from secondary storage. The threads may achieve simultaneous execution on a multiprocessor machine, but even on a uniprocessor system one thread may be able to run while the other is blocked on I/O, improving overall speed of execution.

Modular program structure

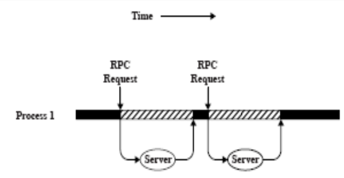
Thread States

States associated with a change in thread state:

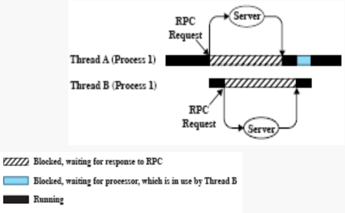
- Spawn
  - Spawn another thread
- Block
- Unblock
- Finish
  - Deallocate register context and stacks

RPC Using Single Thread

Remote procedure call (RPC) is a technique by which two or more programs, typically executing on different machines, interact by using procedure call/return syntax and semantics.

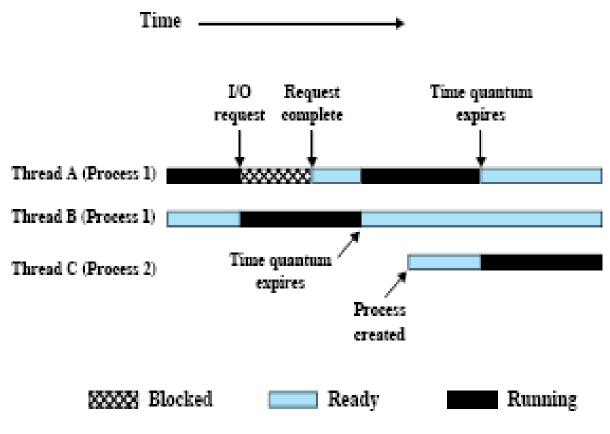


Using a single thread of execution, the calling program must wait for a response from each server before proceeding.


RPC Request
Server
Thread A (Process 1)
Thread B (Process 1)
RPC Request
Server
Blocked, waiting for response to RPC
Blocked, waiting for processor, which is in use by Thread B
Running

Using a separate thread for each RPC, the second call can proceed while the first thread is waiting for a response.

Multithreading on a Uniprocessor



User-Level Threads

All thread management is done by the application

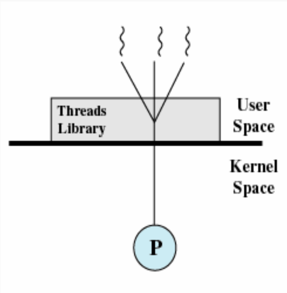
The kernel is not aware of the existence of threads

Can be supplied by a user-level library installed on top of the operating system (*pthread* library).

Thread switching is done within the threads library, so no user-kernel-user mode switches are involved.

Thread scheduling logic is largely embedded within the application program, and so can be customized.

Portability.



OS system calls are typically blocking, so the entire process will be blocked, not just the calling thread.

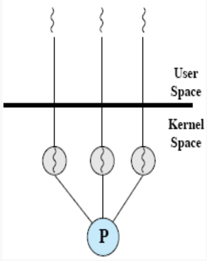
OS kernel assigns the process to a single processor, and it cannot then take advantage of multiple processors if they are available.

Kernel-Level Threads

Kernel maintains context information for the process and the threads

Scheduling is done on a thread basis

Windows is an example of this approach



Thread switching is done by the OS kernel, so each thread switch will require a user-kernel-user mode switch sequence.

Less portability.

VAX Running UNIX-Like Operating System

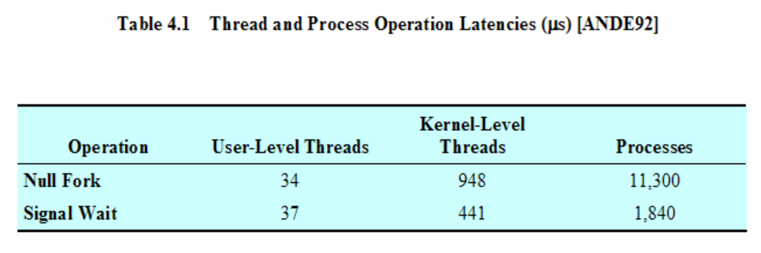


Table 4.1 Thread and Process Operation Latencies (μs) [ANDE92]

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

null fork pure overhead of forking a process/thread

signal wait overhead of synchronizing two processes/threads together

KLTs seem to provide an order-of-magnitude speedup versus single-threaded process.

ULTs seem to provide a similar speedup versus KLTs.

However, whether this holds true in practice depends very much on the specific nature of the application program.

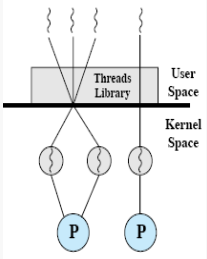
Combined Approaches

Example is Solaris

Thread creation done in the user space

Bulk of scheduling and synchronization of threads occurs within application

Would seem to potentially offer the advantages of both ULTs and KLTs



Relationship Between Threads and Processes

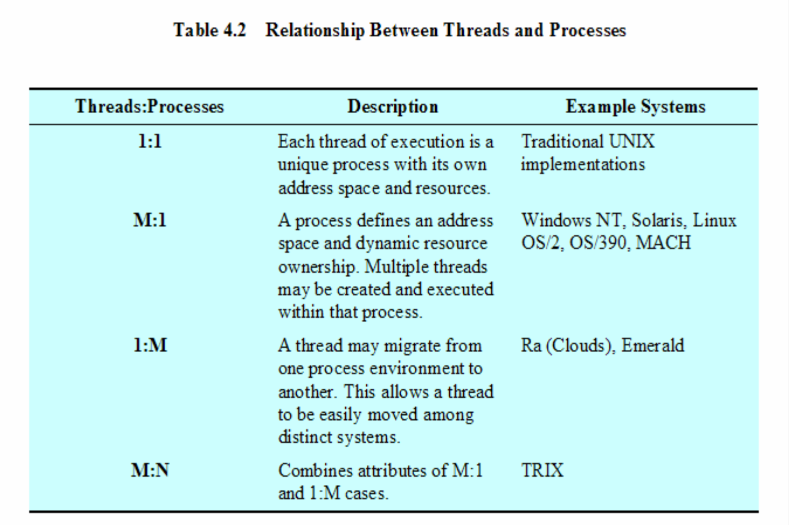


Table 4.2 Relationship Between Threads and Processes

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

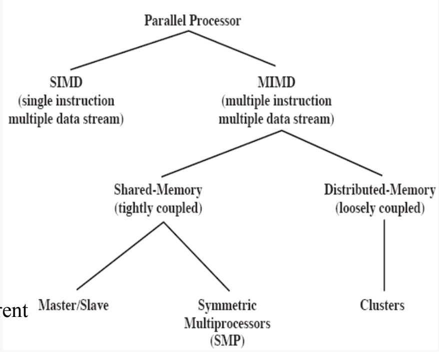
Categories of Computer Systems

Single Instruction Single Data (SISD) stream

Single processor executes a single instruction stream to operate on data stored in a single memory

Single Instruction Multiple Data (SIMD) stream

Each instruction is executed on a different set of data by the different processors


Parallel Processor
SIMD (single instruction multiple data stream)
MIMD (multiple instruction multiple data stream)
Shared-Memory (tightly coupled)
Distributed-Memory (loosely coupled)
Master/Slave
Symmetric Multiprocessors (SMP)
Clusters

Multiple Instruction Single Data (MISD) stream

A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. Never implemented

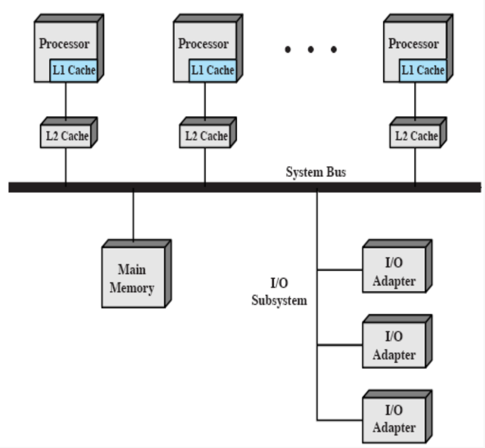
Multiple Instruction Multiple Data (MIMD)

A set of processors simultaneously execute different instruction sequences on different data sets

Symmetric Multiprocessing

Kernel can execute on any processor

Typically each processor does self-scheduling form the pool of available process or threads



Multiprocessor OS Design Considerations

Simultaneous concurrent processes or threads

- kernel routines must be reentrant
- deadlock and invalid system states must be avoided

Scheduling

- may be performed by any processor, so conflicts must be avoided
- with KLTs, the threads of a single process may be scheduled across multiple processors

Synchronization

- may share resources, like address space, among collection of active threads
- must be able to enforce mutual exclusion and event ordering

Memory management

- multi-ported memories to support flexible scheduling
- management schemes must be managed in a cross-processor manner

Reliability and fault tolerance

- failures of single processors should not disable entire system

Microkernel Architecture

Small operating system core

Contains only essential core operating systems functions

Many services traditionally included in the operating system are now external subsystems

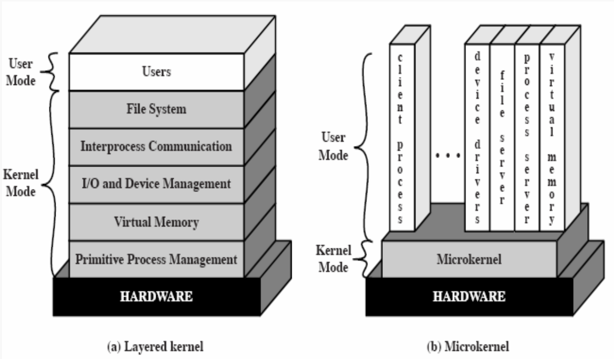
Device drivers

File systems

Virtual memory manager

Windowing system

Security services



(a) Layered kernel

(b) Microkernel

Benefits of a Microkernel Organization

Uniform interface on request made by a process

- don’t distinguish between kernel-level and user-level services
- all services are provided by means of message passing

Extensibility

- allows the addition of new services, affecting only a subset of the system

Flexibility

- new features added, existing features can be subtracted
- users can select among alternate versions of services

Benefits of a Microkernel Organization

Portability

- changes needed to port the system to a new processor are most likely made in the microkernel, not in the other services

Reliability

- modular design
- small microkernel can be rigorously tested

Distributed system support

- message are sent without knowing what the target machine is

Object-oriented operating system

- components are objects with clearly defined interfaces that can be interconnected to form software

What’s the catch?

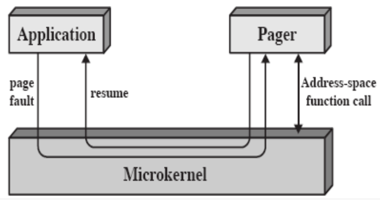
- performance (relative to layered designs)
- continued refinement may close the gap

Microkernel Design

Microkernel must contain the functions that depend directly on the hardware, and the functions needed to support the servers and applications operating in user mode.

Low-level memory management

- mapping each virtual page to a physical page frame
- inter-process protection, page replacement logic can be external to the kernel (e.g., Mach)



Interprocess communication

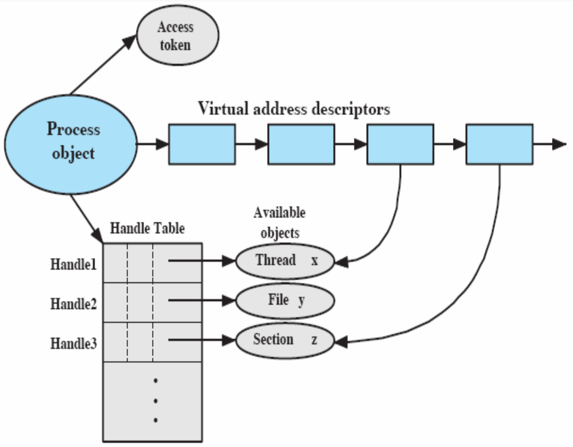
I/O and interrupt management

Windows Processes

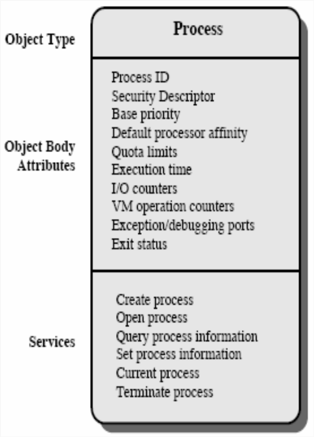
Implemented as objects

An executable process may contain one or more threads

Both processes and thread objects have built-in synchronization capabilities



Windows Process and Thread Objects



Object Type

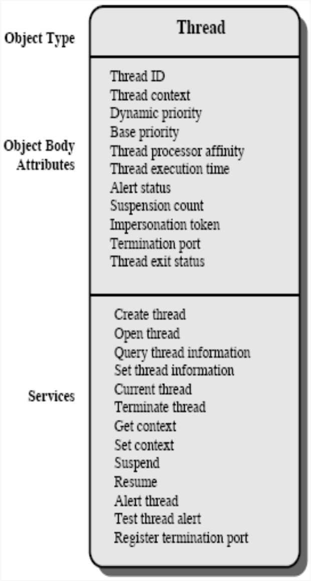
Process

Object Body Attributes

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

Services

Create process
Open process
Query process information
Set process information
Current process
Terminate process



Object Type

Thread

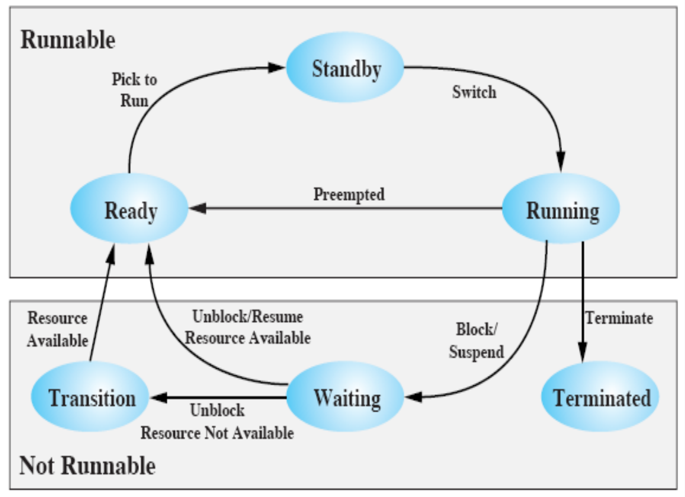
Object Body Attributes

Thread ID
Thread context
Dynamic priority
Base priority
Thread processor affinity
Thread execution time
Alert status
Suspension count
Impersonation token
Termination port
Thread exit status

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

Windows 2000 Thread States



Linux Task Data Structure

State

Scheduling information

Identifiers

Interprocess communication

Links

Times and timers

File system

Address space

Processor-specific context

Linux States of a Process

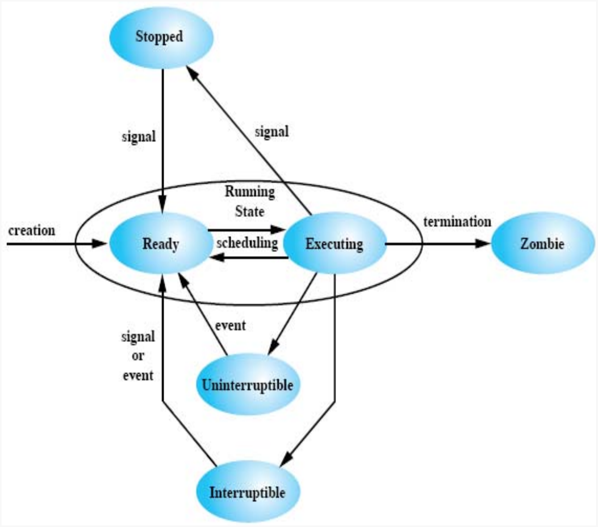
Running

Interruptable

Uninterruptable

Stopped

Zombie


Stopped
signal
signal
Running State
creation
Ready
scheduling
Executing
termination
Zombie
event
signal or event
Uninterruptible
Interruptible