CSC 553 Operating Systems

Lecture 4- Threads

Processes and Threads

- 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 the entity that is scheduled and dispatched by the OS

Processes and Threads

- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***
- The unit of resource ownership is referred to as a ***process*** or ***task***
- ***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

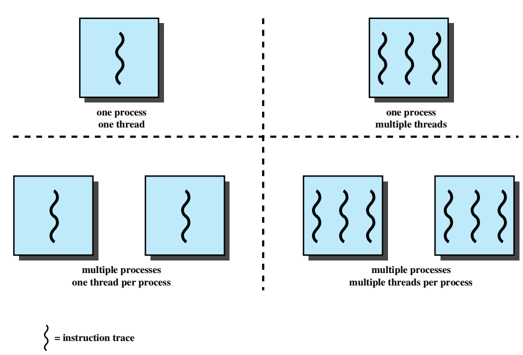


Figure 4.1 Threads and Processes

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

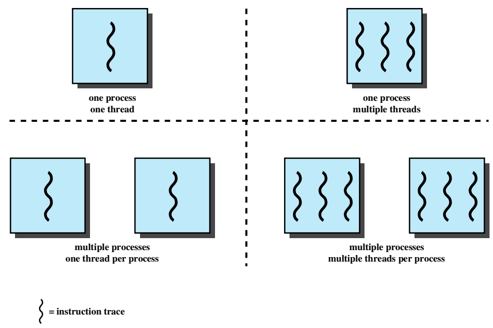


Figure 4.1 Threads and Processes

Process

- Defined in a multithreaded environment as "the unit of resource allocation and a unit of protection"
- Associated with processes:
  - A virtual address space that holds the process image
  - Protected access to:
    - Processors
    - Other processes (for interprocess communication)
    - Files
    - I/O resources (devices and channels)

One or More Threads in a Process

Each thread has:

- An execution state (Running, Ready, etc.)
- A 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 processes, shared with all other threads in that process

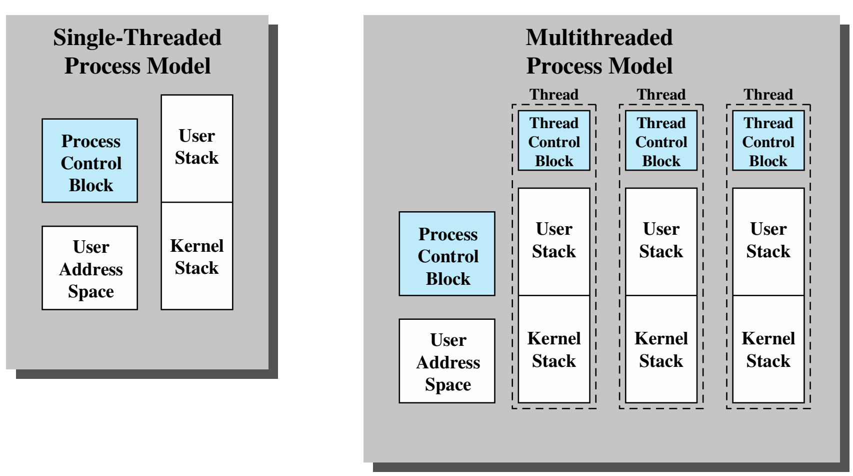
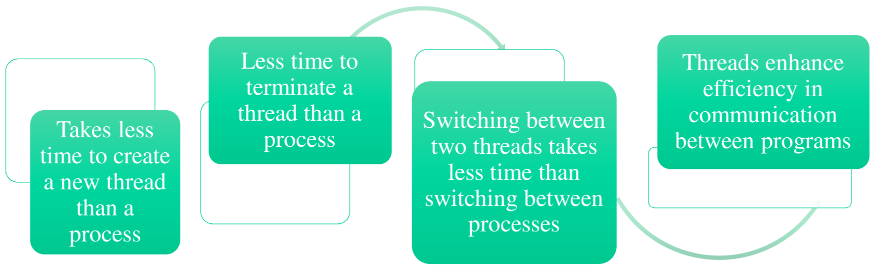


Figure 4.2 Single Threaded and Multithreaded Process Models

Key Benefits of Threads



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

Threads 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

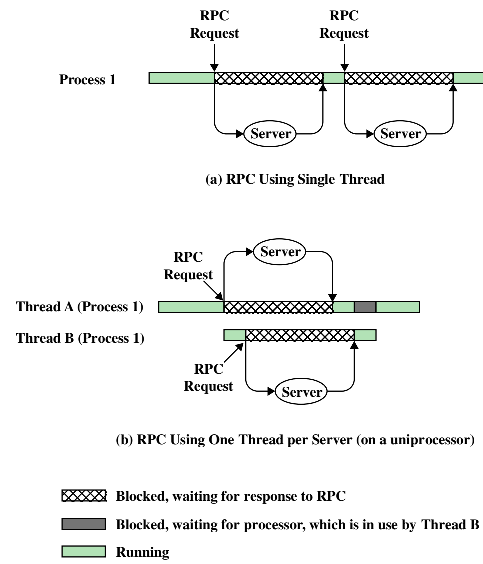


Figure 4.3 Remote Procedure Call (RPC) Using Threads

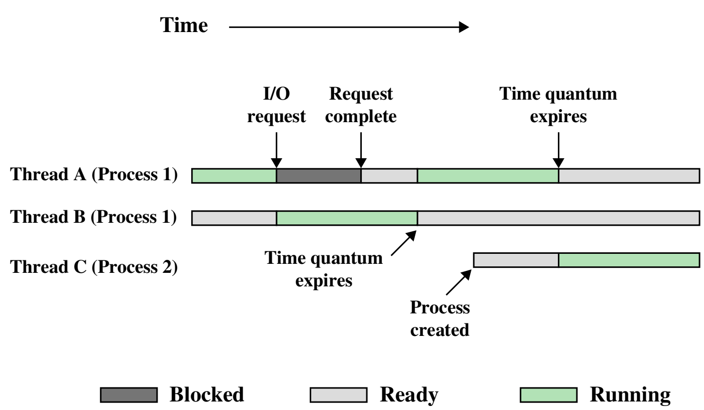
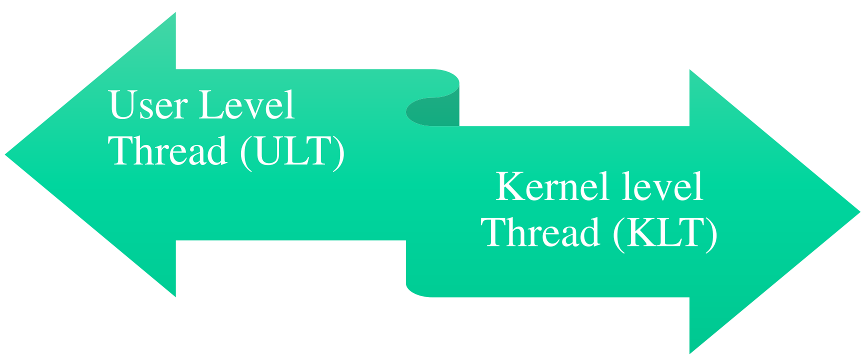


Figure 4.4 Multithreading Example on a Uniprocessor

Threads Synchronization

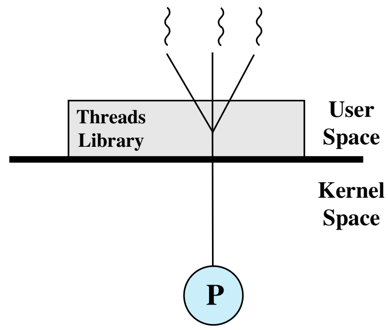
- It is necessary to synchronize the activities of the various threads
  - All threads of a process share the same address space and other resources
  - Any alteration of a resource by one thread affects the other threads in the same process

Types of Threads



User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level

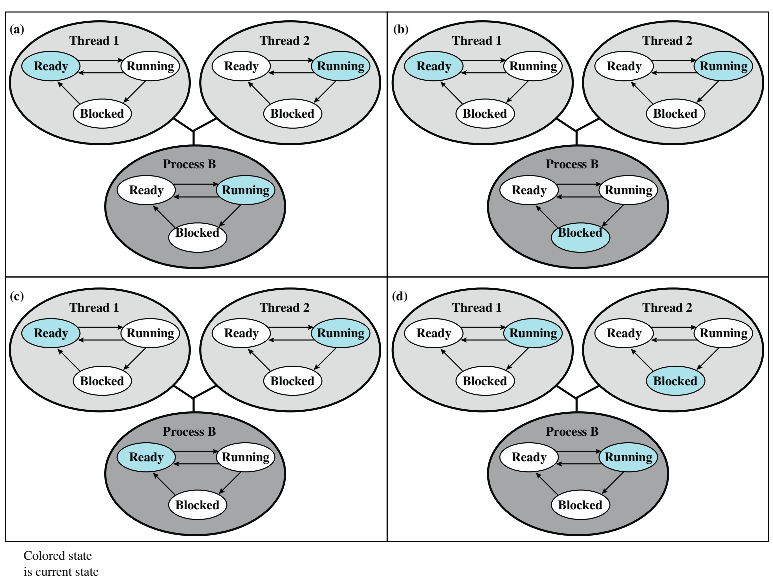
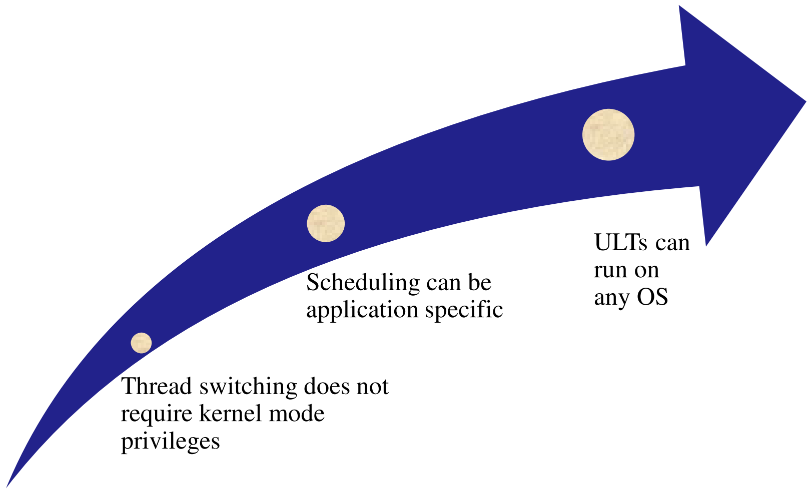


Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

Advantages of ULTs



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 as well
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
  - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time

Overcoming ULTs Disadvantages

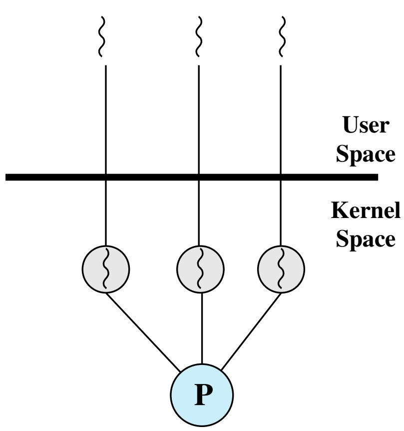
Jacketing

- Purpose is to convert a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads

- However, this approach eliminates the main advantage of threads

Kernel Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the kernel
- There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
- 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 themselves can be multithreaded

Disadvantages of KLTs

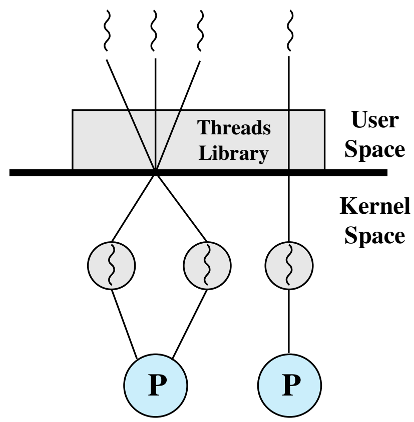
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

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

Thread and Process Operation Latencies (µs)

Combined Approaches

- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application
- Solaris is a good example



(c) Combined

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

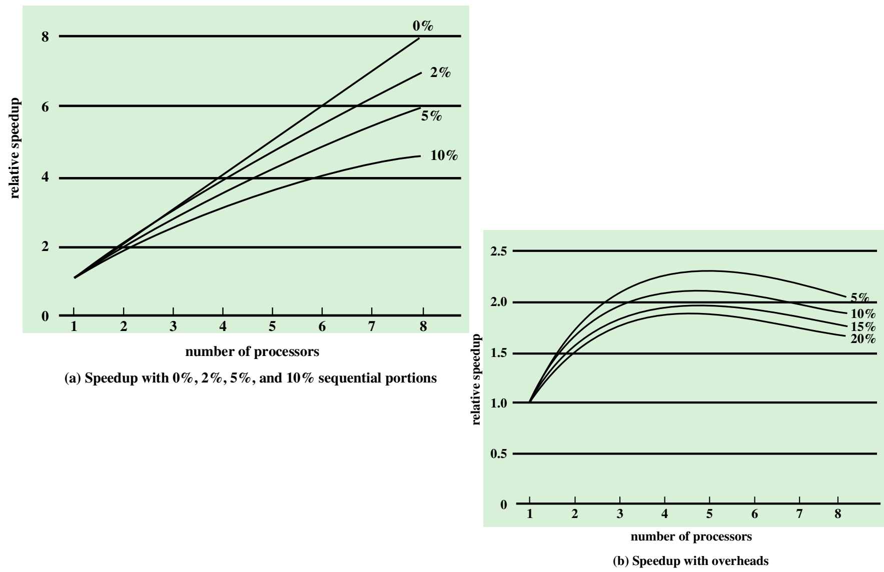


Figure 4.7 Performance Effect of Multiple Cores

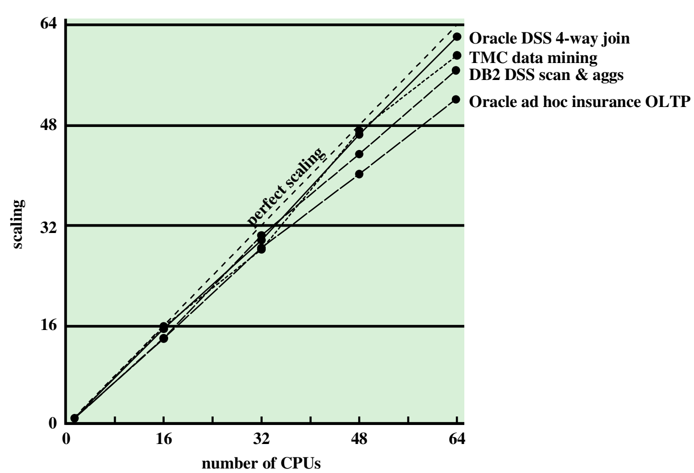


Figure 4.8 Scaling of Database Workloads on Multiple-Processor Hardware

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

Applications That Benefit

- Java applications
  - All applications that use a Java 2 Platform, Enterprise Edition application server can immediately benefit from multicore technology
- Multi-instance applications
  - Multiple instances of the application in parallel

Valve Game Software

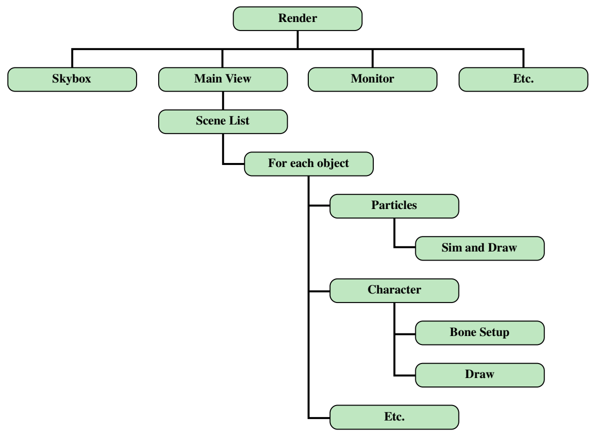


Figure 4.9 Hybrid Threading for Rendering Module

Windows Process and Thread Management

- An ***application*** consists of one or more processes
- Each ***process*** provides the resources needed to execute a program
- A ***thread*** is the entity within a process that can be scheduled for execution
- A ***job object*** allows groups of process to be managed as a unit

Windows Process and Thread Management

- A ***thread pool*** is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application
- A ***fiber*** is a unit of execution that must be manually scheduled by the application
- ***User-mode scheduling (UMS)*** is a lightweight mechanism that applications can use to schedule their own threads

Management of Background Tasks and Application Lifetimes

- Beginning with Windows 8, and carrying through to Windows 10, developers are responsible for managing the state of their individual applications
- Previous versions of Windows always give the user full control of the lifetime of a process

Management of Background Tasks and Application Lifetimes

- In the new Metro interface Windows takes over the process lifecycle of an application
  - A limited number of applications can run alongside the main app in the Metro UI using the SnapView functionality
  - Only one Store application can run at one time

Management of Background Tasks and Application Lifetimes

- Live Tiles give the appearance of applications constantly running on the system
  - In reality they receive push notifications and do not use system resources to display the dynamic content offered

Metro Interface

- Foreground application in the Metro interface has access to all of the processor, network, and disk resources available to the user
  - All other apps are suspended and have no access to these resources
- When an app enters a suspended mode, an event should be triggered to store the state of the user's information
  - This is the responsibility of the application developer

Metro Interface

- Windows may terminate a background app
  - You need to save your app's state when it's suspended, in case Windows terminates it so that you can restore its state later
  - When the app returns to the foreground another event is triggered to obtain the user state from memory

Windows Process

Important characteristics of Windows processes are:

- Windows processes are implemented as objects
- A process can be created as a new process or 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

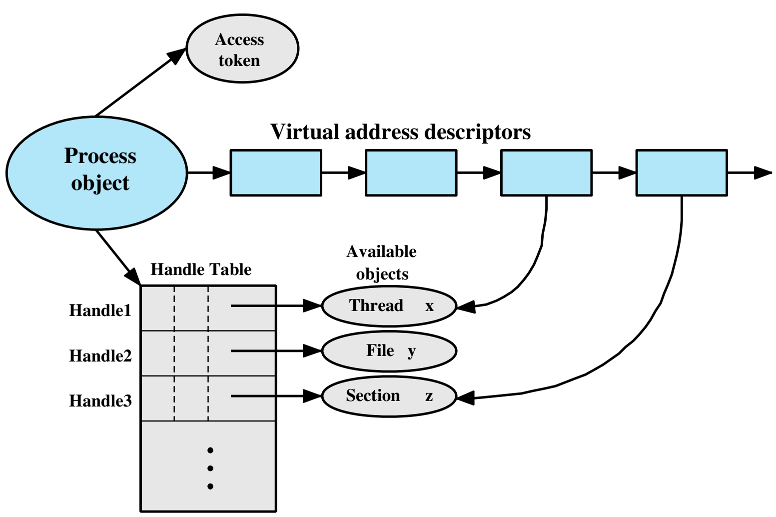


Figure 4.10 A Windows Process and Its Resources

Process and Threads Objects

Windows makes use of two types of process-related objects:

Processes	Threads
• An entity corresponding to a user job or application that owns resources	• A dispatchable unit of work that executes sequentially and is interruptible

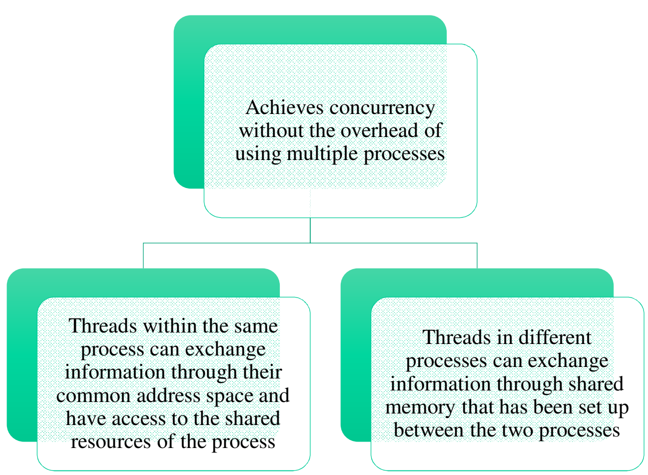
Process ID	A unique value that identifies the process to the operating system.
Security descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that the process's threads have performed.
Exception/debugging ports	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.
Exit status	The reason for a process's termination.

Windows Process Object Attributes

Windows Process Thread Attributes

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

Multithreading



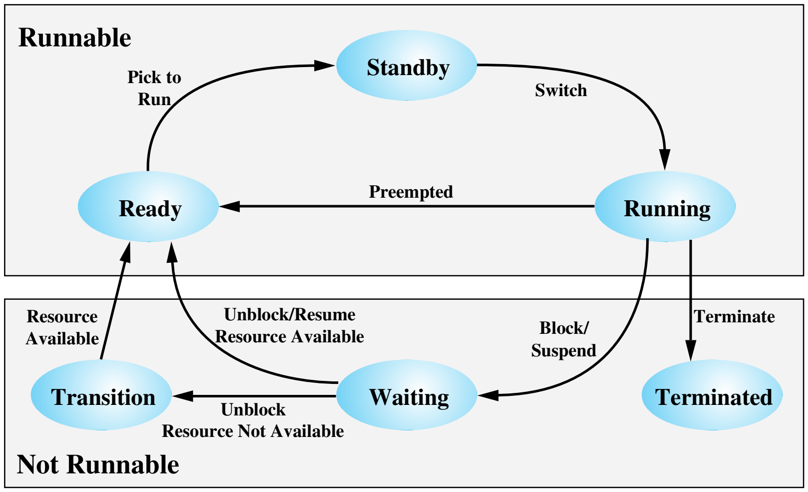


Figure 4.11 Windows Thread States

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

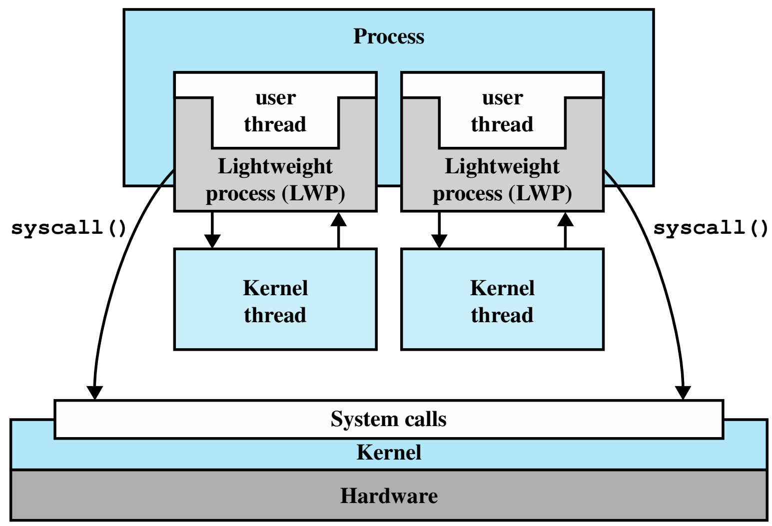


Figure 4.12 Processes and Threads in Solaris

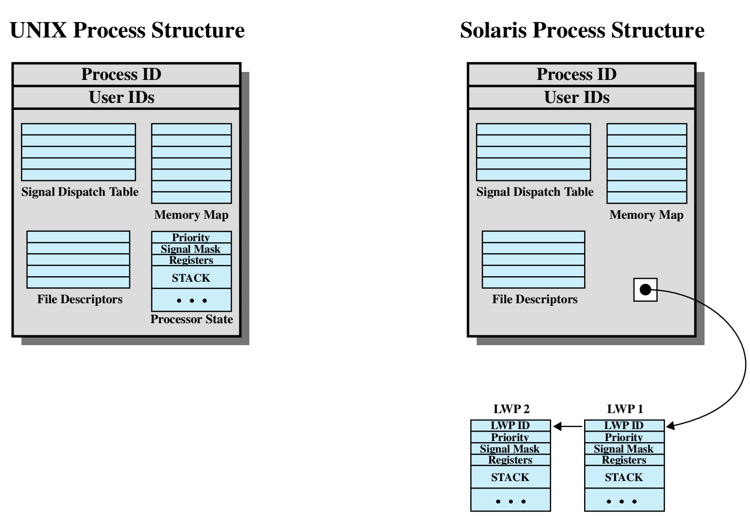


Figure 4.13 Process Structure in Traditional UNIX and Solaris [LEWI96]

A Lightweight Process (LWP) Data Structures Includes:

- An LWP identifier
- The priority of this LWP and hence the kernel thread that supports it
- A signal mask that tells the kernel which signals will be accepted
- Saved values of user-level registers

A Lightweight Process (LWP) Data Structures Includes:

- The kernel stack for this LWP, which includes system call arguments, results, and error codes for each call level
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

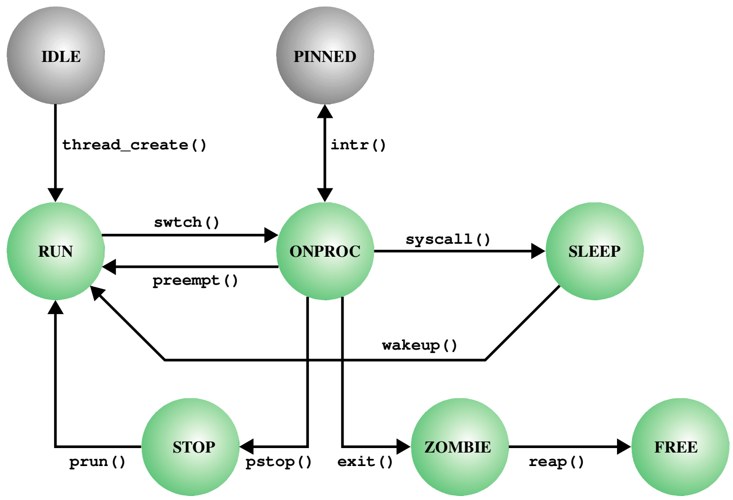


Figure 4.14 Solaris Thread States

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 unifies these two concepts into a single model, namely kernel threads, and the mechanisms for scheduling and executing kernel threads
- To do this, interrupts are converted to kernel threads

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

Linus Tasks

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

This structure contains information in a number of categories

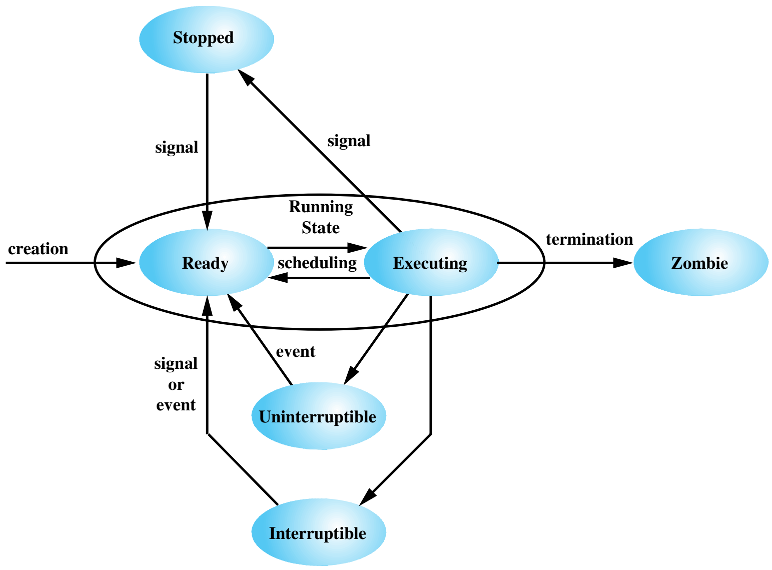


Figure 4.15 Linux Process/Thread Model

Linux Threads

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

Linux Namespaces

- A namespace enables a process to have a different view of the system than other processes that have other associated namespaces
- There are currently six namespaces in Linux
  - mnt - pid
  - net - ipc
  - uts -user

Android Process and Thread Management

- An Android application is the software that implements an app
- Each Android application consists of one or more instance of one or more of four types of application components

Android Process and Thread Management

- Each component performs a distinct role in the overall application behavior, and each component can be activated independently within the application and even by other applications
- Four types of components:
  – Activities
  – Services
  – Content providers
  – Broadcast receivers

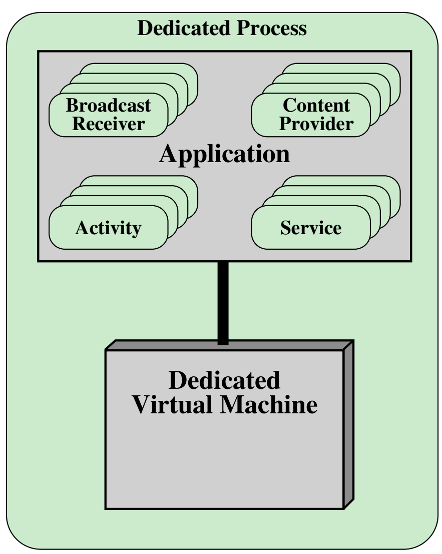


Figure 4.16 Android Application

Activities

- An Activity is an application component that provides a screen with which users can interact in order to do something
- Each Activity is given a window in which to draw its user interface
- The window typically fills the screen, but may be smaller than the screen and float on top of other windows

Activities

- An application may include multiple activities
- When an application is running, one activity is in the foreground, and it is this activity that interacts with the user
- The activities are arranged in a last-in-first-out stack in the order in which each activity is opened

Activities

- If the user switches to some other activity within the application, the new activity is created and pushed on to the top of the back stack, while the preceding foreground activity becomes the second item on the stack for this application

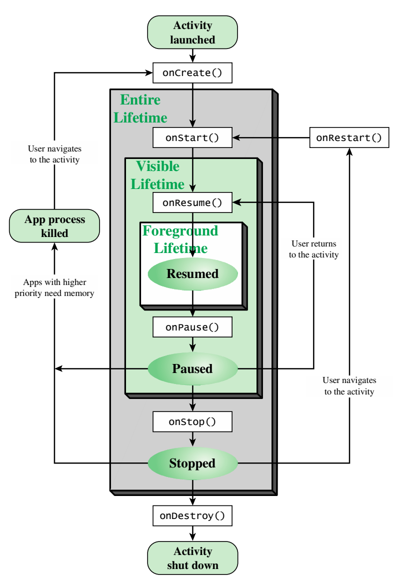
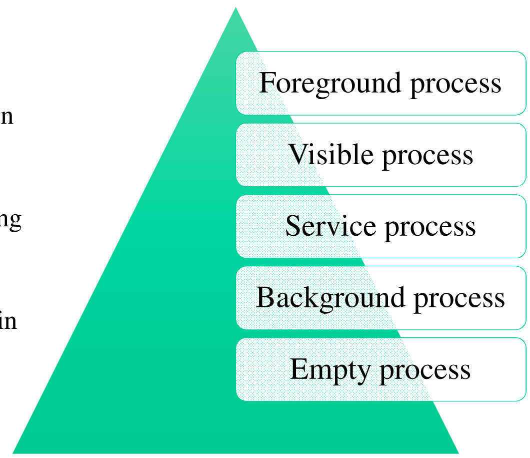


Figure 4.17 Activity State Transition Diagram

Processes and Threads

- A precedence hierarchy is used to determine which process or processes to kill in order to reclaim needed resources
- Processes are killed beginning with the lowest precedence first
- The levels of the hierarchy, in descending order of precedence are:



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

- Processes and threads
  - Multithreading
  - Thread functionality
- Types of threads
  - User level and kernel level threads
- Multicore and multithreading
  - Performance of Software on Multicore

Summary

- Windows process and thread management
  - Management of background tasks and application lifecycles
  - Windows process
  - Process and thread objects
  - Multithreading
  - Thread states
  - Support for OS subsystems

Summary

- Solaris thread and SMP management
  - Multithreaded architecture
  - Motivation
  - Process structure
  - Thread execution
  - Interrupts as threads

Summary

- Linux process and thread management
  - Tasks/threads/namespaces
- Android process and thread management
  - Android applications
  - Activities
  - Processes and threads
- Mac OS X grand central dispatch