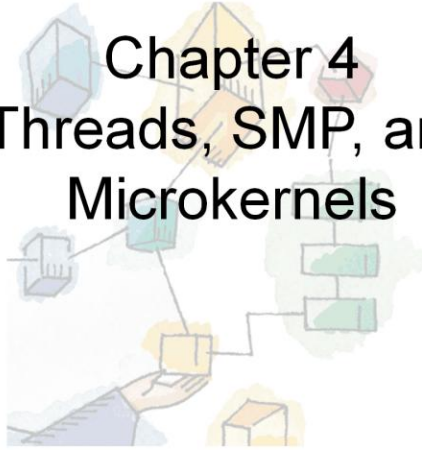


*Operating Systems:
Internals and Design Principles, 6/E*
William Stallings

Chapter 4 Threads, SMP, and Microkernels



Dave Bremer
Otago Polytechnic, N.Z.
©2008, Prentice Hall

These slides are intended to help a teacher develop a presentation. This PowerPoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.

Roadmap



→ Threads: Resource ownership and execution

- Symmetric multiprocessing (SMP).
- Microkernel
- Case Studies of threads and SMP:
 - Windows
 - Solaris
 - Linux





Processes and Threads

- Processes have two characteristics:
 - **Resource ownership** - process includes a virtual address space to hold the process image
 - **Scheduling/execution** - follows an execution path that may be interleaved with other processes
- These two characteristics are treated independently by the operating system





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.

Figure 4.1. Threads and Processes [ANDE97]

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

Single Thread Approaches

- MS-DOS supports a single user process and a single thread.
- Some UNIX, support multiple user processes but only support one thread per process

Figure 4.1 Threads and Processes [ANDE97]

Animated Slide

Onload Enlarges top-left to discuss DOS

Click1: Enlarges bottom-left for Unix

Single Threaded approach: The traditional approach of a single thread of execution per process, in which the concept of a thread is not recognized, examples are

- MS DOS (single process, single thread)
- Unix (multiple, single threaded processes)

Multithreading

- Java run-time environment is a single process with multiple threads
- Multiple processes *and* threads are found in Windows, Solaris, and many modern versions of UNIX

Figure 4.1 Threads and Processes [ANDE97]

Animated Slide

Onload: Emphasis on top-right and JRE (single process, multiple thread),

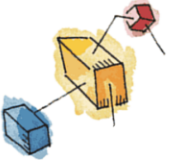
Click 1: Emphasis on multiple processes with multiple threads – this is the main topic of this chapter

JRE is an example of a system of one process with multiple threads.

Of main interest in this chapter is the use of multiple processes, each of which support multiple threads.


Examples include:

- Windows,
- Solaris,
- and many modern versions of UNIX.

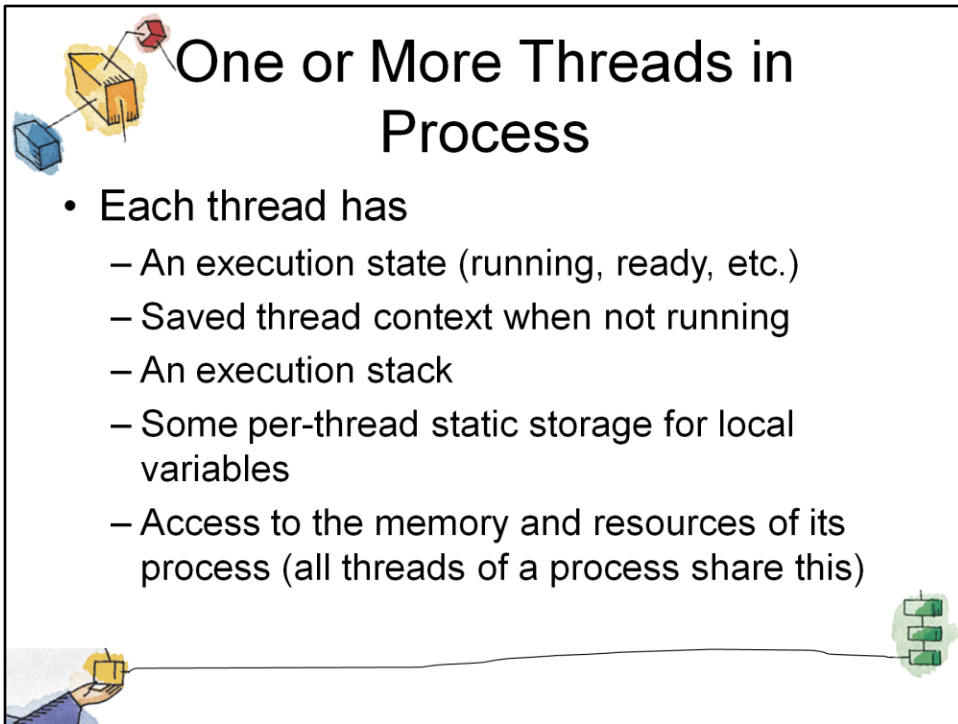


Processes

- A virtual address space which holds the process image
- Protected access to
 - Processors,
 - Other processes,
 - Files,
 - I/O resources




In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection.



One or More Threads in 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)



Within a process, there may be one or more threads, each with the following:

- A thread execution state (Running, Ready, etc.).
- A saved thread context when not running;

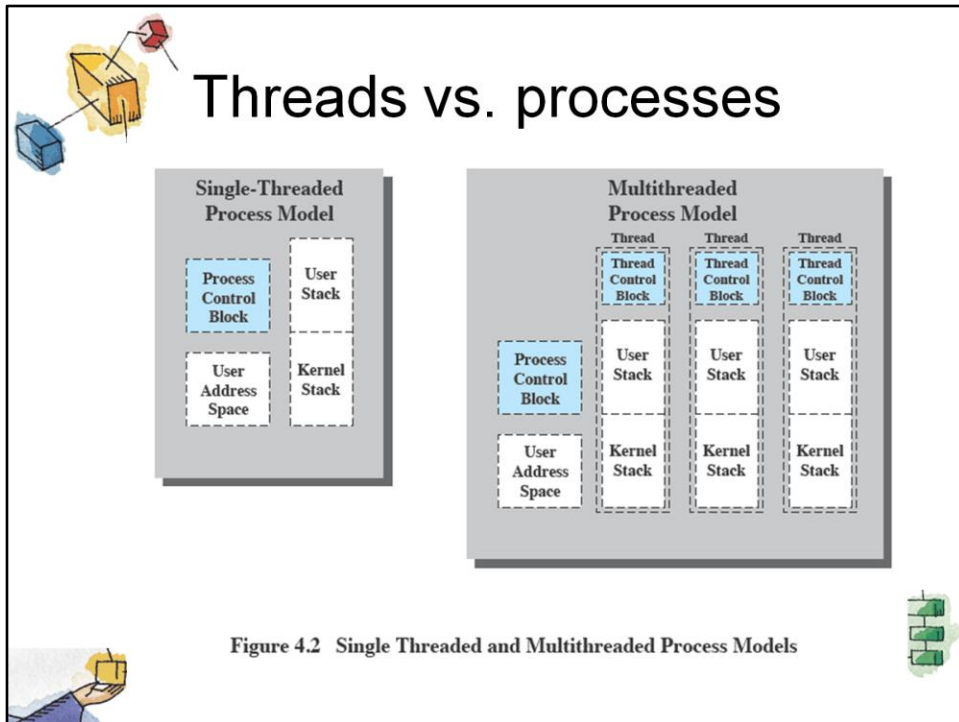
one way to view a thread is as an independent program counter operating within a process.



One view...

- *One way to view a thread is as an independent program counter operating **within** a process.*





Distinction between threads and processes from the point of view of process management.

In a single-threaded process model, the representation of a process includes

- its process control block
- user address space,
- user and kernel stacks to manage the call/return behaviour of the execution of the process.

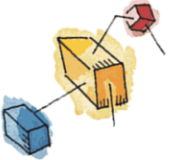
While the process is running, it controls the processor registers. The contents of these registers are saved when the process is not running.

In a multithreaded environment,

- there is still a single process control block and user address space associated with the process,
- **but** separate stacks for each thread,
- as well as a separate control block for each thread containing register values, priority, and other thread-related state information.


Thus, all of the threads of a process share the state and resources of that process.

- They reside in the same address space and have access to the same data.
- When one thread alters an item of data in memory, other threads see the results if and when they access that item.
- If one thread opens a file with read privileges, other threads in the same process can also read from that file.



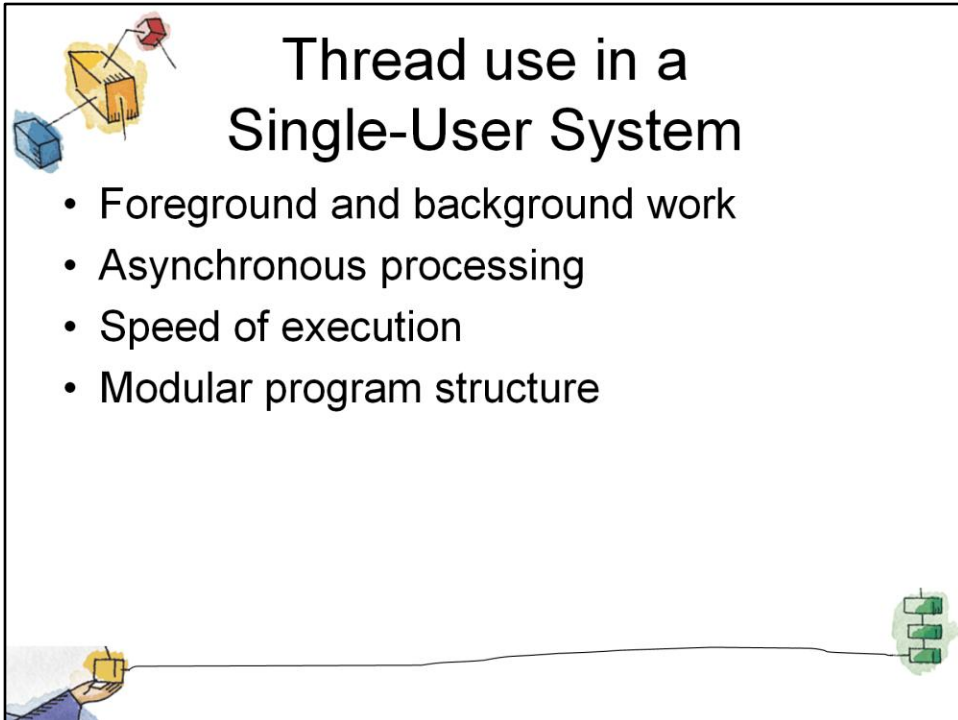
Benefits of Threads

- 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 processes
- Threads can communicate with each other
 - without invoking the kernel



If there is an application or function that should be implemented as a set of related units of execution,

it is far more efficient to do so as a collection of threads - rather than a collection of separate processes.



Thread use in a Single-User System

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

Foreground and background work

e.g. Spreadsheet

- one thread looking after display
- Another thread updating results of formulae

Asynchronous processing

E.G. protection against power failure within a word processor,

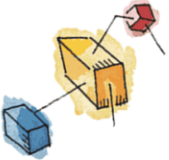
- A thread writes random access memory (RAM) buffer to disk once every minute.
- this thread schedules itself directly with the OS;
- no need for fancy code in the main program to provide for time checks or to coordinate input and output.

Speed of execution

- On thread can compute one batch of data while another thread reading the next batch from a device.
- On a multiprocessor system, multiple threads from the same process may be able to execute simultaneously.
- Even though one thread may be blocked for an I/O operation to read in a batch of data, another thread may be executing.


Modular program structure

- Threads make it easier to design programs which involve a variety of activities or a variety of sources and destinations of input and output.



Threads

- Several actions that affect all of the threads in a process
 - The OS must manage these at the process level.
- Examples:
 - Suspending a process involves suspending all threads of the process
 - Termination of a process, terminates all threads within the process



Suspension involves swapping the address space of one process out of main memory to make room for the address space of another process.

Because all threads in a process share the same address space, all threads are suspended at the same time.

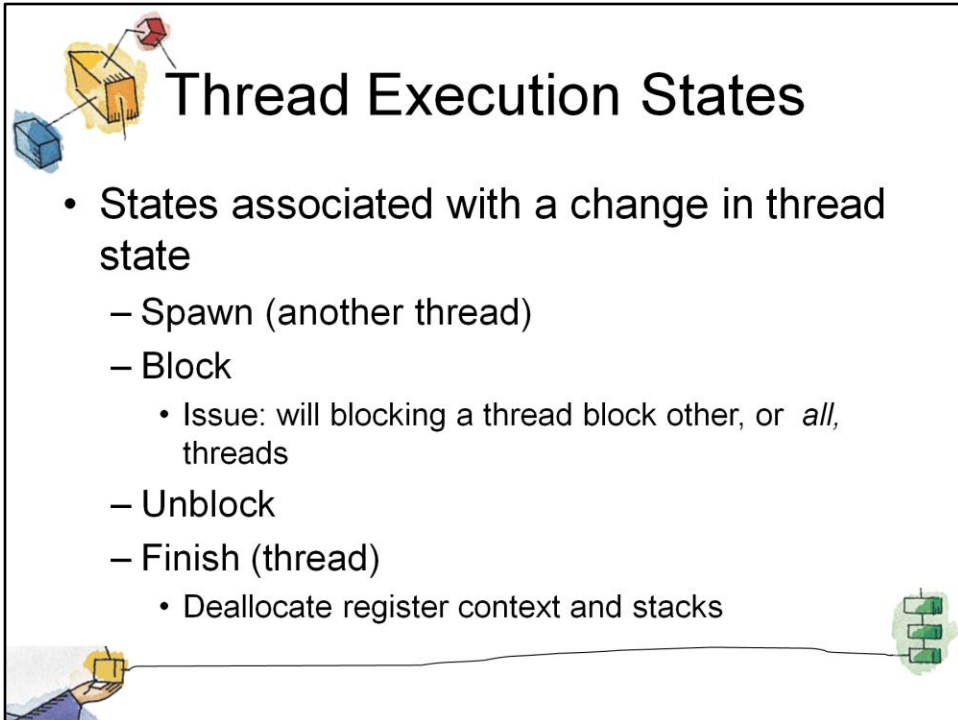
Similarly, termination of a process terminates all threads within that process.



Activities similar to Processes

- Threads have execution states and may synchronize with one another.
 - Similar to processes
- We look at these two aspects of thread functionality in turn.
 - States
 - Synchronisation



The diagram is enclosed in a black rectangular border. In the top-left corner, there is a small illustration of a yellow box with a red dot on top, connected by lines to a blue box and a red box. In the bottom-left corner, there is an illustration of a hand holding a yellow box. In the bottom-right corner, there is an illustration of a green box with a white box inside it. The title "Thread Execution States" is written in a large, black, sans-serif font in the upper right portion of the diagram. Below the title is a bulleted list of thread execution states.

Thread Execution States

- States associated with a change in thread state
 - Spawn (another thread)
 - Block
 - Issue: will blocking a thread block other, or *all*, threads
 - Unblock
 - Finish (thread)
 - Deallocate register context and stacks

A significant issue is whether the blocking of a thread results in the blocking of the entire process.

If one thread in a process is blocked, does this prevent the running of any other thread in the same process even if that other thread is in a ready state?

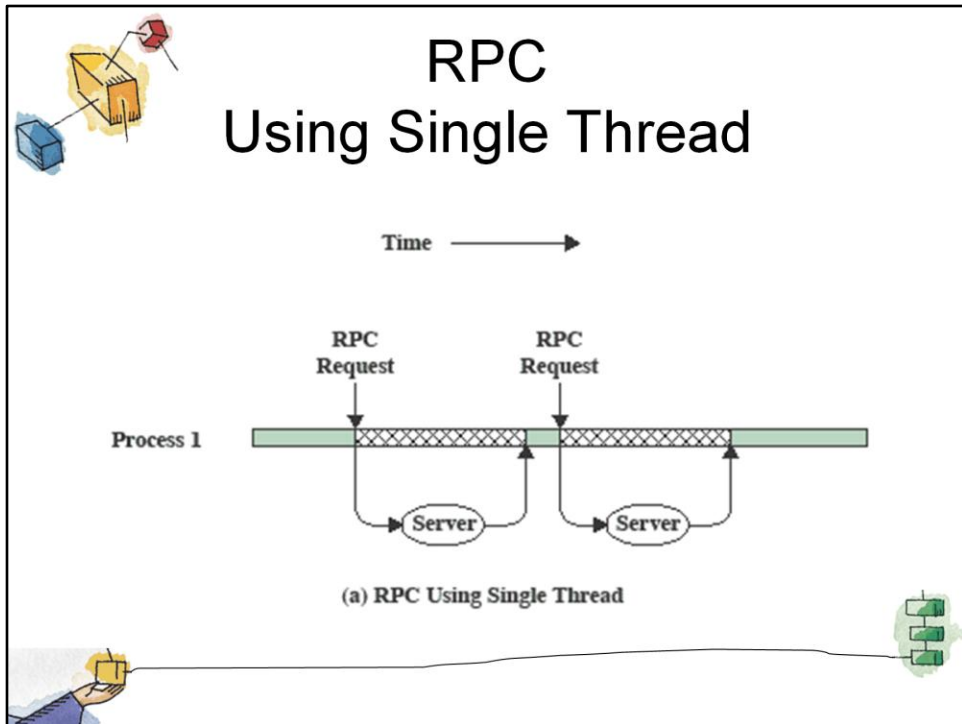
Clearly, some of the flexibility and power of threads is lost if the one blocked thread blocks an entire process.



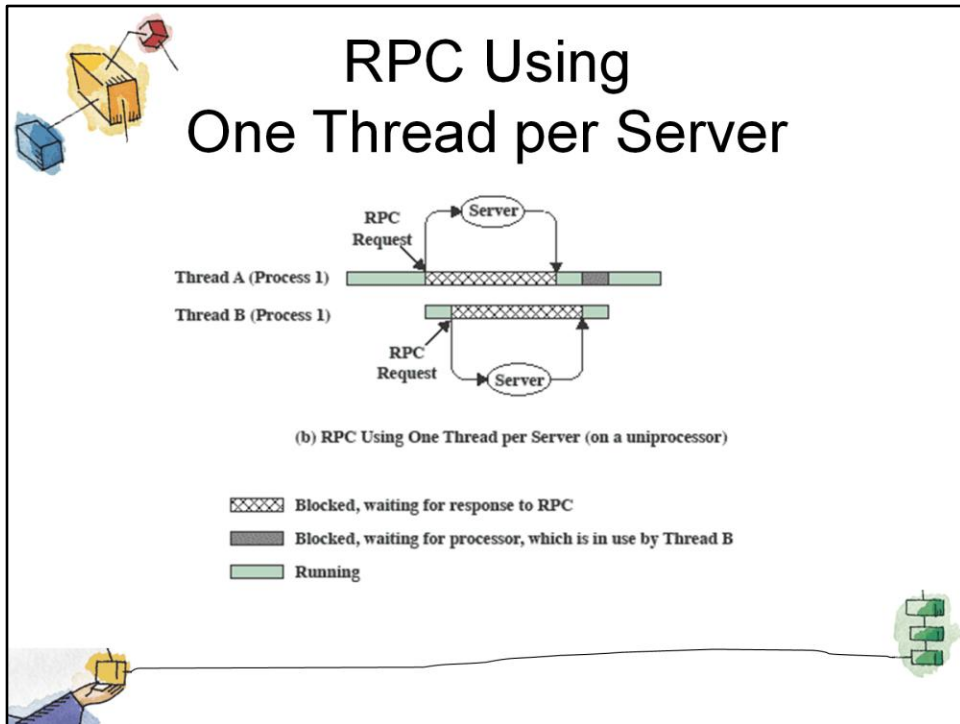
Example: Remote Procedure Call

- Consider:
 - A program that performs two remote procedure calls (RPCs)
 - to two different hosts
 - to obtain a combined result.





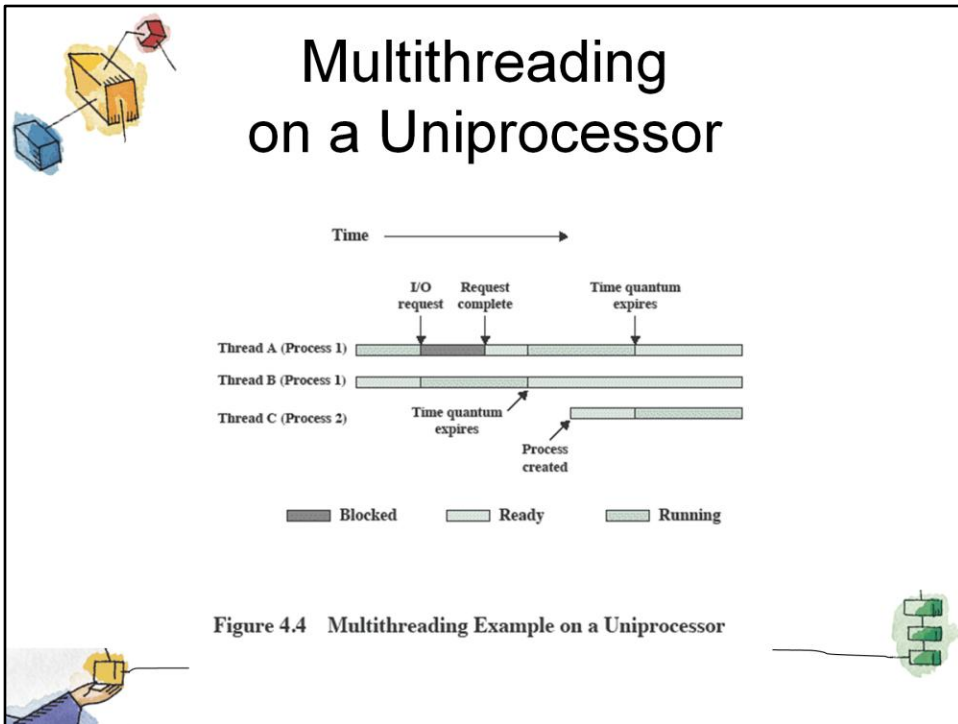
The results are obtained in sequence,
so that the program has to wait for a response from each server in turn.



Rewriting the program to use a separate thread for each RPC results in a substantial speedup.

Note that if this program operates on a uniprocessor, the requests must be generated sequentially and the results processed in sequence;

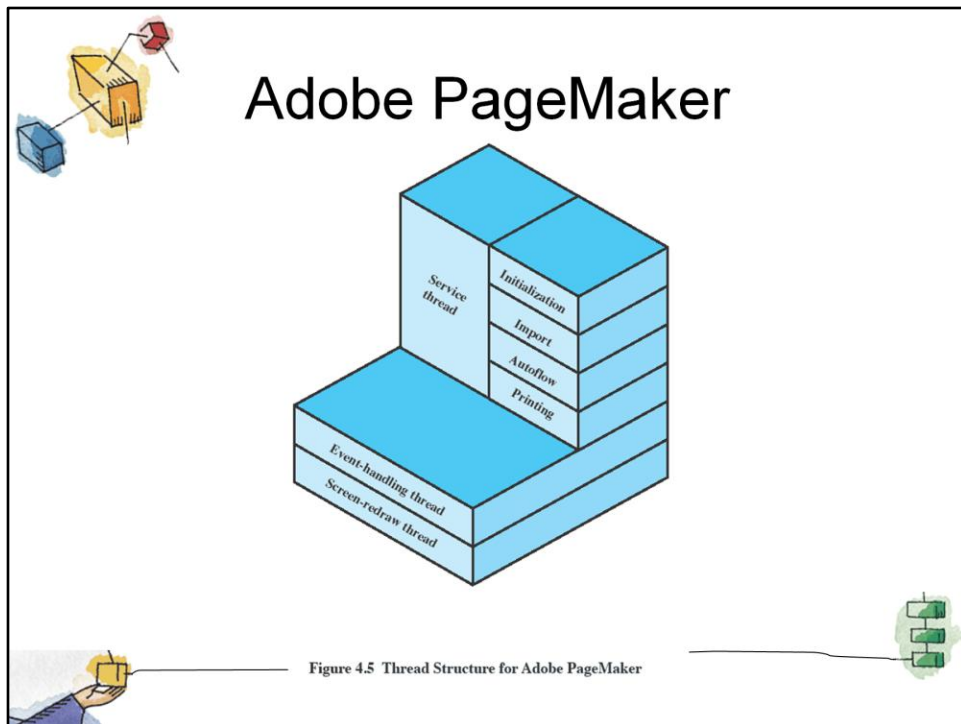
however, the program waits concurrently for the two replies.



On a uniprocessor, multiprogramming enables the interleaving of multiple threads within multiple processes.

In the example,

- three threads in two processes are interleaved on the processor.
- Execution passes from one thread to another either when the currently running thread is blocked or its time slice is exhausted.



An example of the use of threads is the Adobe PageMaker application running under a shared system.

Three threads are always active:

- an event-handling thread,
- a screen-redraw thread,
- a service thread.



Categories of Thread Implementation

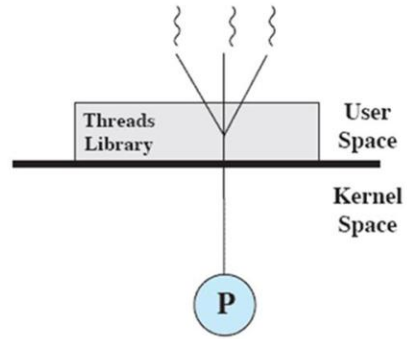
- User Level Thread (ULT)
- Kernel level Thread (KLT) also called:
 - kernel-supported threads
 - lightweight processes.





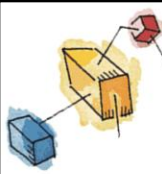
User-Level Threads

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

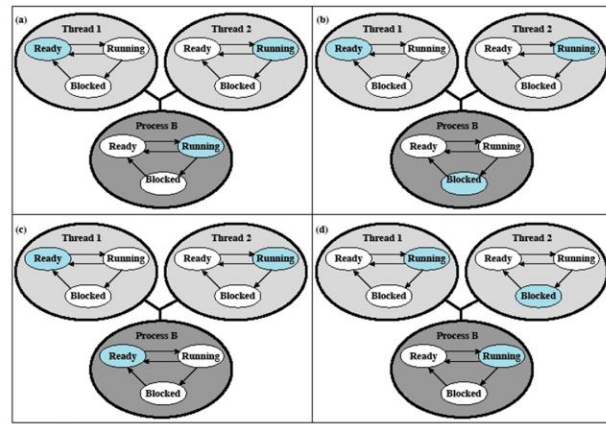


(a) Pure user-level





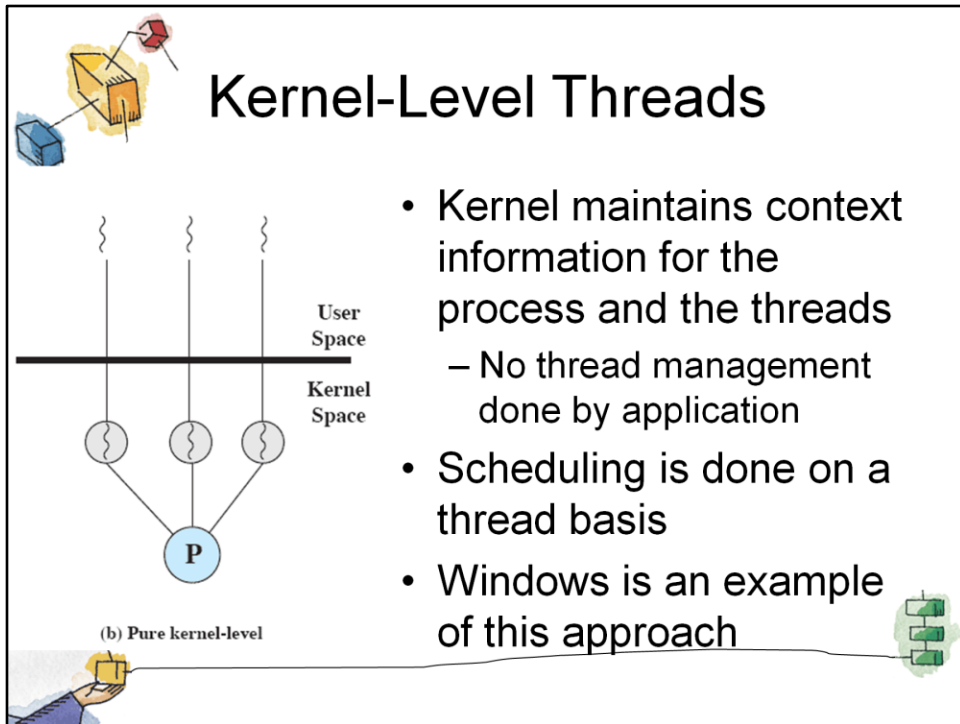
Relationships between ULT Thread and Process States



Colored state is current state

Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States





The kernel maintains context information for the process as a whole and for individual threads within the process.

Scheduling by the kernel is done on a thread basis.



Advantages of KLT

- 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.





Disadvantage of KLT

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




Combined Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads by the application
- Example is Solaris

The diagram illustrates a combined approach to thread execution. A horizontal line separates the 'User Space' (top) from the 'Kernel Space' (bottom). In the User Space, a grey box labeled 'Threads Library' contains four wavy lines representing threads. These threads are connected to two processes in the Kernel Space, labeled 'P'. The first process 'P' has two threads connected to it, and the second process 'P' has one thread connected to it. The caption below the diagram is '(c) Combined'.

In a combined approach, multiple threads within the same application can run in parallel on multiple processors,
and a blocking system call need not block the entire process.


If properly designed, this approach should combine the advantages of the pure ULT and KLT approaches while minimizing the disadvantages.



Relationship Between Thread 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



The concepts of resource allocation and dispatching unit have traditionally been embodied in the single concept of the process; that is, as a 1 : 1 relationship between threads and processes.

There has been much interest in providing for multiple threads within a single process, which is a many-to-one relationship

.

However, as the table shows, the other two combinations have also been investigated, namely,

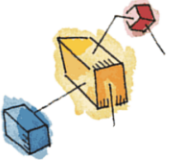
- a many-to-many relationship and
- a one-to-many relationship.



Roadmap



- Threads: Resource ownership and execution
- Symmetric multiprocessing (SMP).
- Microkernel
- Case Studies of threads and SMP:
 - Windows
 - Solaris
 - Linux





Traditional View

- Traditionally, the computer has been viewed as a sequential machine.
 - A processor executes instructions one at a time in sequence
 - Each instruction is a sequence of operations
- Two popular approaches to providing parallelism
 - Symmetric MultiProcessors (SMPs)
 - Clusters (ch 16)



Traditionally, the computer has been viewed as a sequential machine.

Most computer programming languages require the programmer to specify algorithms as sequences of instructions.

A processor executes programs by executing machine instructions in sequence and one at a time. Each instruction is executed in a sequence of operations


(fetch instruction, fetch operands, perform operation, store results).

This view of the computer has never been entirely true.

As computer technology has evolved and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to improve performance and, in some cases, to improve reliability.



the two most popular approaches to providing parallelism by replicating processors:

- symmetric multiprocessors (SMPs) and
- clusters.



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



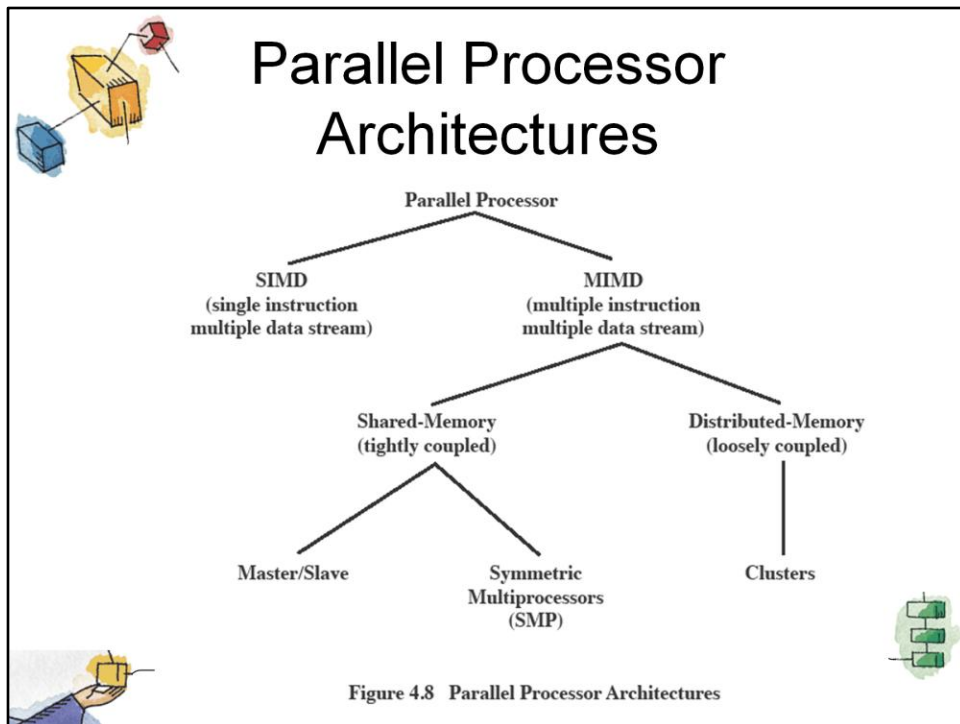
It is useful to see where SMP architectures fit into the overall category of parallel processors.



Categories of Computer Systems

- Multiple Instruction Single Data (MISD) stream (Never implemented)
 - A sequence of data is transmitted to a set of processors, each of execute a different instruction sequence
- Multiple Instruction Multiple Data (MIMD)
 - A set of processors simultaneously execute different instruction sequences on different data sets

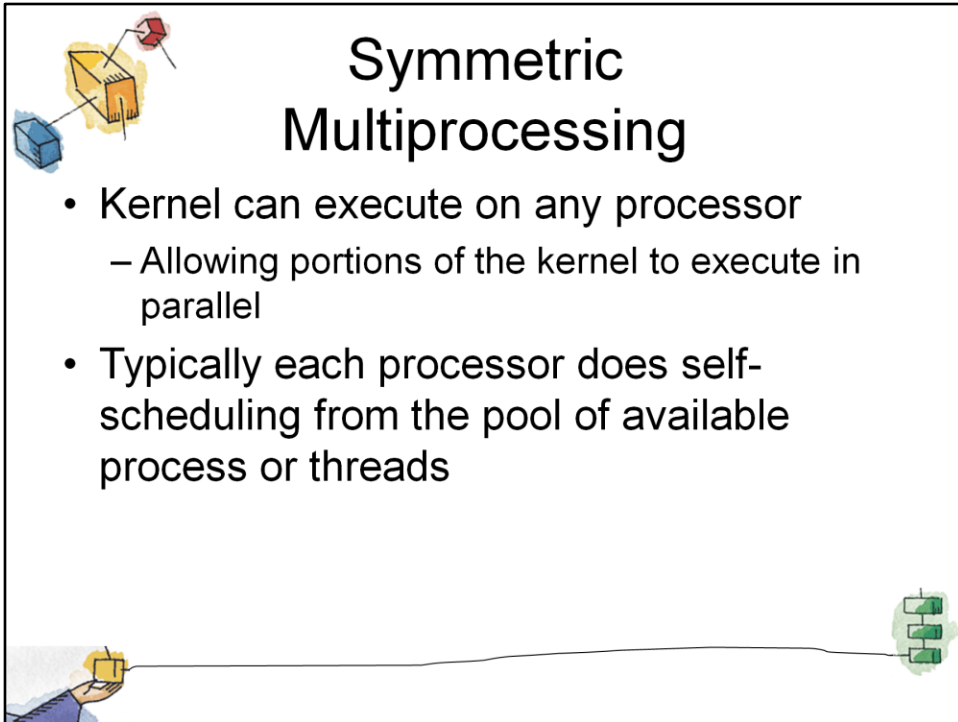




With the MIMD organization, the processors are general purpose, because they must be able to process all of the instructions necessary to perform the appropriate data transformation.

MIMDs can be subdivided by the means in which the processors communicate.

- If the processors each have a dedicated memory, then each processing element is a self-contained computer.
 - Such a system is known as a **cluster, or multicomputer.**
- If the processors share a common memory, then each processor accesses programs and data stored in the shared memory, and processors communicate with each other via that memory;
 - such a system is known as a **shared-memory multiprocessor.**



Symmetric Multiprocessing

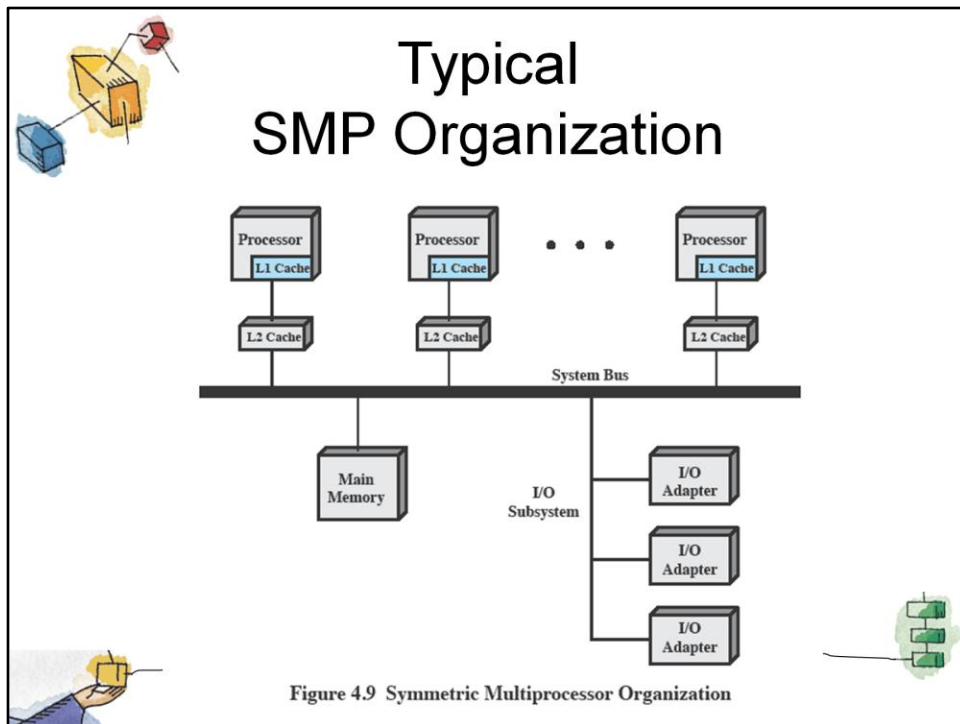
- Kernel can execute on any processor
 - Allowing portions of the kernel to execute in parallel
- Typically each processor does self-scheduling from the pool of available process or threads

In a symmetric multiprocessor (SMP),

- the kernel can execute on any processor, and
- typically each processor does self-scheduling from the pool of available processes or threads.

The kernel can be constructed as multiple processes or multiple threads, allowing portions of the kernel to execute in parallel. This complicates the OS.

- It must ensure that two processors do not choose the same process and that processes are not somehow lost from the queue.
- Techniques must be employed to resolve and synchronize claims to resources.




There are multiple processors, each of which contains its own control unit, arithmetic-logic unit, and registers.

Each processor has access to a shared main memory and the I/O devices through some form of interconnection mechanism; a shared bus is a common facility.



The processors can communicate with each other through memory (messages and status information left in shared address spaces).

It may also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible.



Multiprocessor OS Design Considerations

- The key design issues include
 - Simultaneous concurrent processes or threads
 - Scheduling
 - Synchronization
 - Memory Management
 - Reliability and Fault Tolerance

Talk through each of the issues

Simultaneous concurrent processes or threads:

- Kernel routines need to be re-entrant to allow several processors to execute the same kernel code simultaneously.
- With multiple processors executing the same or different parts of the kernel, kernel tables and management structures must be managed properly to avoid deadlock or invalid operations.

Scheduling:

- Scheduling may be performed by any processor, so conflicts must be avoided.
- If kernel-level multithreading is used, then the opportunity exists to schedule multiple threads from the same process simultaneously on multiple processors.

Synchronization:

- With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization.
- Synchronization is a facility that enforces mutual exclusion and event ordering.
- A common synchronization mechanism used in multiprocessor operating systems is locks

Memory management:

- Memory management on a multiprocessor must deal with all of the issues found on uniprocessor computers.
- The OS also needs to exploit the available hardware parallelism, such as multiport memories, to achieve the best performance.
- The paging mechanisms on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement.

Reliability and fault tolerance:

- The OS should provide graceful degradation in the face of processor failure.
- The scheduler and other portions of the OS must recognize the loss of a processor and restructure management tables accordingly.

Roadmap




- Threads: Resource ownership and execution
- Symmetric multiprocessing (SMP).

→ Microkernel



- Case Studies of threads and SMP:
 - Windows
 - Solaris
 - Linux





Microkernel

- A microkernel is a small OS core that provides the foundation for modular extensions.
- Big question is how small must a kernel be to qualify as a microkernel
 - *Must* drivers be in user space?
- In theory, this approach provides a high degree of flexibility and modularity.



Also, whether to run nonkernel operations in kernel or user space, and whether to keep existing subsystem code (e.g., a version of UNIX) or start from scratch.

The microkernel approach was popularized by its use in the Mach OS, which is now the core of the Macintosh Mac OS X operating system.

A number of products now boast microkernel implementations, and this general design approach is likely to be seen in most of the personal computer, workstation, and server operating systems developed in the near future.

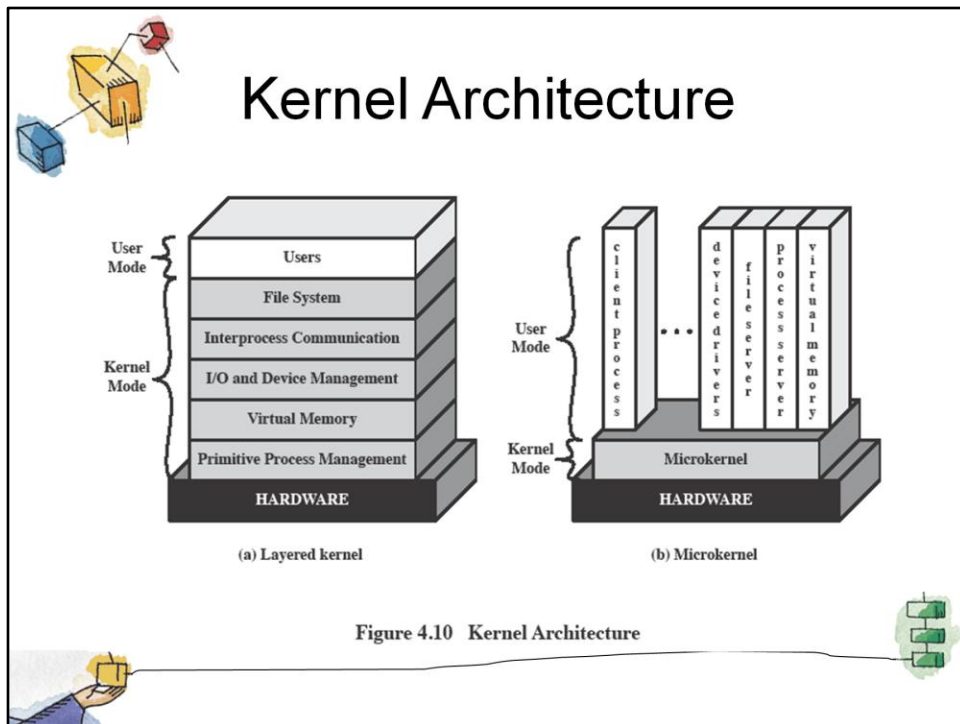


Figure A:

Operating systems developed in the mid to late 1950s were designed with little concern about structure.

- The problems caused by mutual dependence and interaction were grossly underestimated.
- In these monolithic operating systems, virtually any procedure can call any other procedure – the approach was unsustainable as operating systems grew to massive proportions.

Modular programming techniques were needed to handle this scale of software development.

- layered operating systems were developed in which functions are organized hierarchically and interaction only takes place between adjacent layers.
- Most or all of the layers execute in kernel mode.

PROBLEM:

- Major changes in one layer can have numerous effects on code in adjacent layers - many difficult to trace
- And security is difficult to build in because of the many interactions between adjacent layers.

Figure B

In a Microkernel - only absolutely essential core OS functions should be in the kernel.

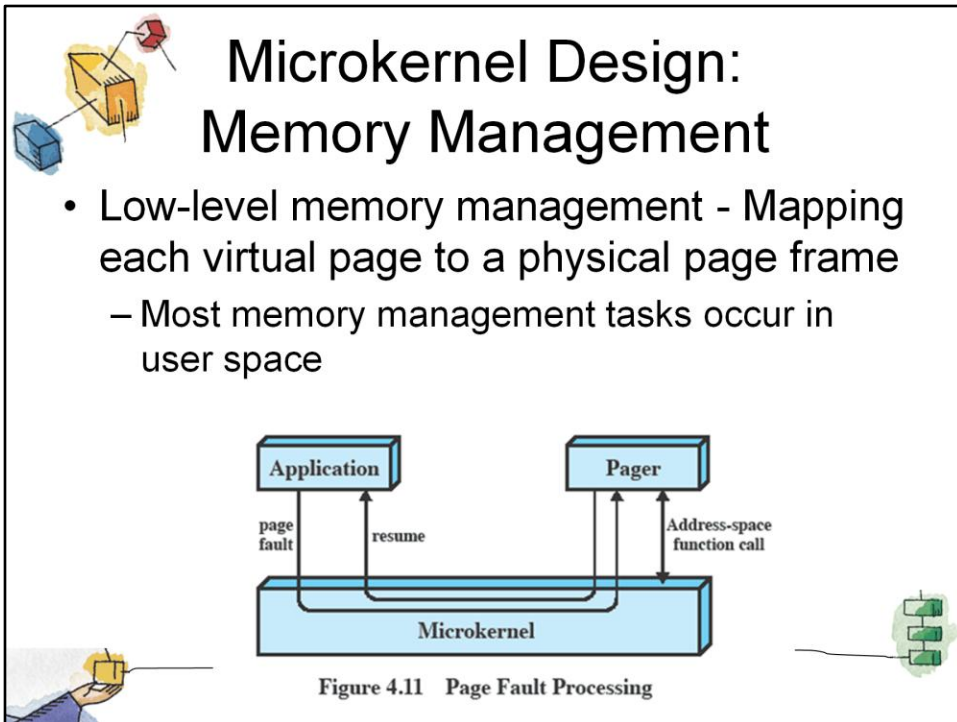
- Less essential services and applications are built on the microkernel and execute in user mode.
- Common characteristic is that many services that traditionally have been part of the OS are now external subsystems that interact with the kernel and with each other;
- these include device drivers, file systems, virtual memory manager, windowing system, and security services.

The microkernel functions as a message exchange:

- It validates messages,
- passes them between components,
- and grants access to hardware.

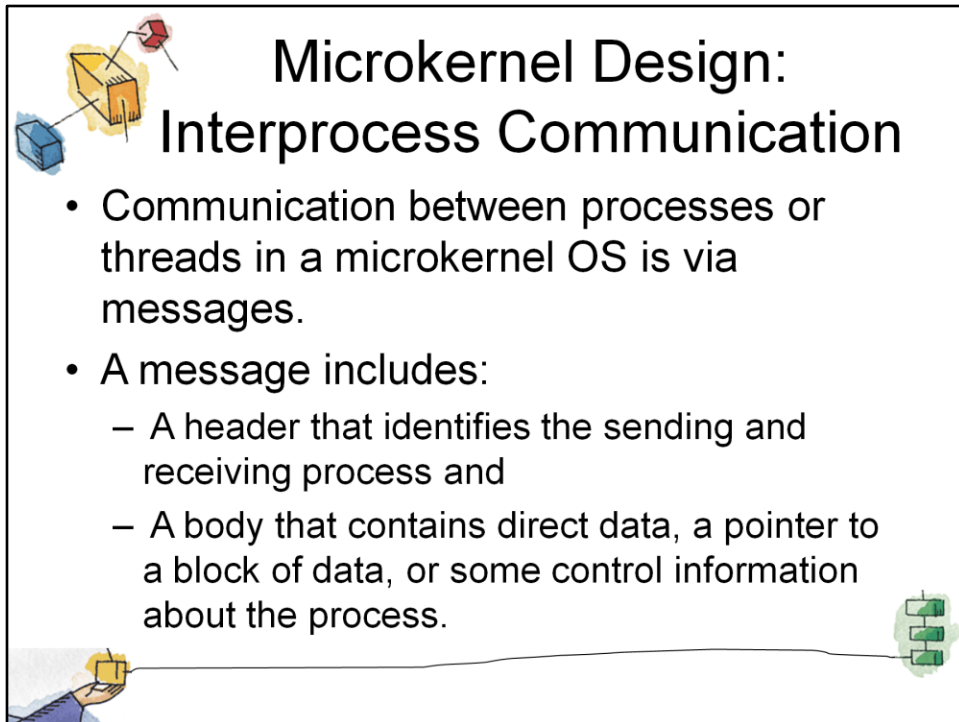
The microkernel also performs a protection function;

it prevents message passing unless exchange is allowed.



The microkernel has to control the hardware concept of address space to make it possible to implement protection at the process level.

Providing the microkernel is responsible for mapping each virtual page to a physical frame, the majority of memory management can be implemented outside the kernel (protection of the address space of one process from another and the page replacement algorithm and other paging logic etc)

The diagram is enclosed in a black rectangular border. In the top-left corner, there is an illustration of a yellow cube with a red dot on top, connected by lines to a blue cube and a red cube. In the bottom-left corner, a hand is shown holding a yellow cube. In the bottom-right corner, there is a small green icon consisting of three stacked rectangular blocks. The main text is centered at the top.

Microkernel Design: Interprocess Communication

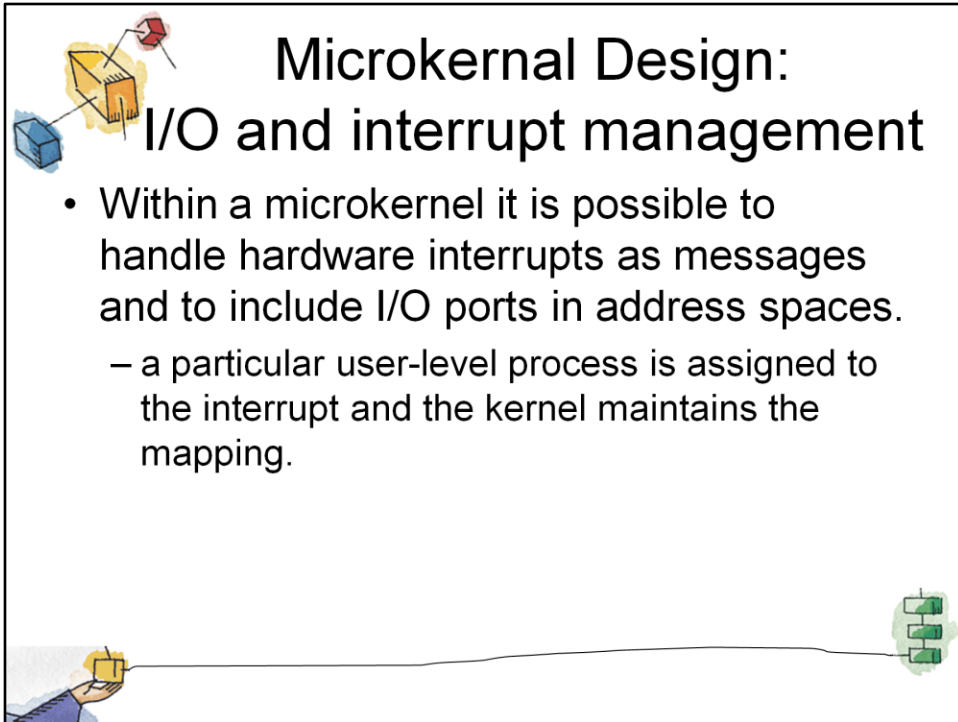
- Communication between processes or threads in a microkernel OS is via messages.
- A message includes:
 - A header that identifies the sending and receiving process and
 - A body that contains direct data, a pointer to a block of data, or some control information about the process.

The basic form of communication between processes or threads in a microkernel OS is messages.

A message includes:

- A header that identifies the sending and receiving process and
- A body that contains direct data, a pointer to a block of data, or some control information about the process.

Typically, we can think of IPC as being based on ports associated with processes.




The diagram is enclosed in a black rectangular border. In the top-left corner, there is a cluster of icons: a blue cube, a yellow cube with a red dot on top, and a red cube. In the top-right corner, the title "Microkernel Design: I/O and interrupt management" is written in a large, black, sans-serif font. Below the title, a bulleted list contains two items. The first item is a main bullet point, and the second is a sub-bullet point. In the bottom-left corner, there is an illustration of a hand holding a yellow cube. In the bottom-right corner, there is a small green icon consisting of three stacked rectangular blocks. A thin black line runs horizontally across the bottom of the diagram, starting from the hand and ending at the green icon.

Microkernel Design: I/O and interrupt management

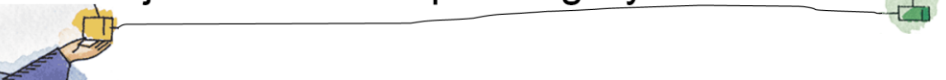
- Within a microkernel it is possible to handle hardware interrupts as messages and to include I/O ports in address spaces.
 - a particular user-level process is assigned to the interrupt and the kernel maintains the mapping.

With a microkernel architecture, it is possible to handle hardware interrupts as messages and to include I/O ports in address spaces.



Benefits of a Microkernel Organization

- Uniform interfaces on requests made by a process.
- Extensibility
- Flexibility
- Portability
- Reliability
- Distributed System Support
- Object Oriented Operating Systems



Uniform Interfaces

- Imposes a uniform interface on requests made by a process.
- Processes need not distinguish between kernel-level and user-level services because all such services are provided by means of message passing.

Extensibility is facilitated allowing the addition of new services as well as the provision of multiple services in the same functional area.

- when a new feature is added, only selected servers need to be modified or added.
- The impact of new or modified servers is restricted to a subset of the system.
- Modifications do not require building a new kernel.

Flexibility

- Existing features can be subtracted to produce a smaller, more efficient implementation.
- If features are made optional, the base product will appeal to a wider variety of users.

Portability

- All or at least much of the processor-specific code is in the microkernel.
- Thus, changes needed to port the system to a new processor are fewer and tend to be arranged in logical groupings.

Reliability

- A small microkernel can be rigorously tested.
- Its use of a small number of application programming interfaces (APIs) improves the chance of producing quality code for the OS services outside the kernel.

Distributed System Support

- When a message is sent from a client to a server process, the message must include an identifier of the requested service.
- If a distributed system (e.g., a cluster) is configured so that all processes and services have unique identifiers, then in effect there is a single system image at the microkernel level.
- A process can send a message without knowing on which computer the target service resides.

Object Oriented Operating Systems

- An object-oriented approach can lend discipline to the design of the microkernel and to the development of modular extensions to the OS.
-



Roadmap

- Threads: Resource ownership and execution
- Symmetric multiprocessing (SMP).
- Microkernel



Case Studies of threads and SMP:

- Windows
- Solaris
- Linux






Different Approaches to Processes


- Differences between different OS's support of processes include
 - How processes are named
 - Whether threads are provided
 - How processes are represented
 - How process resources are protected
 - What mechanisms are used for inter-process communication and synchronization
 - How processes are related to each other



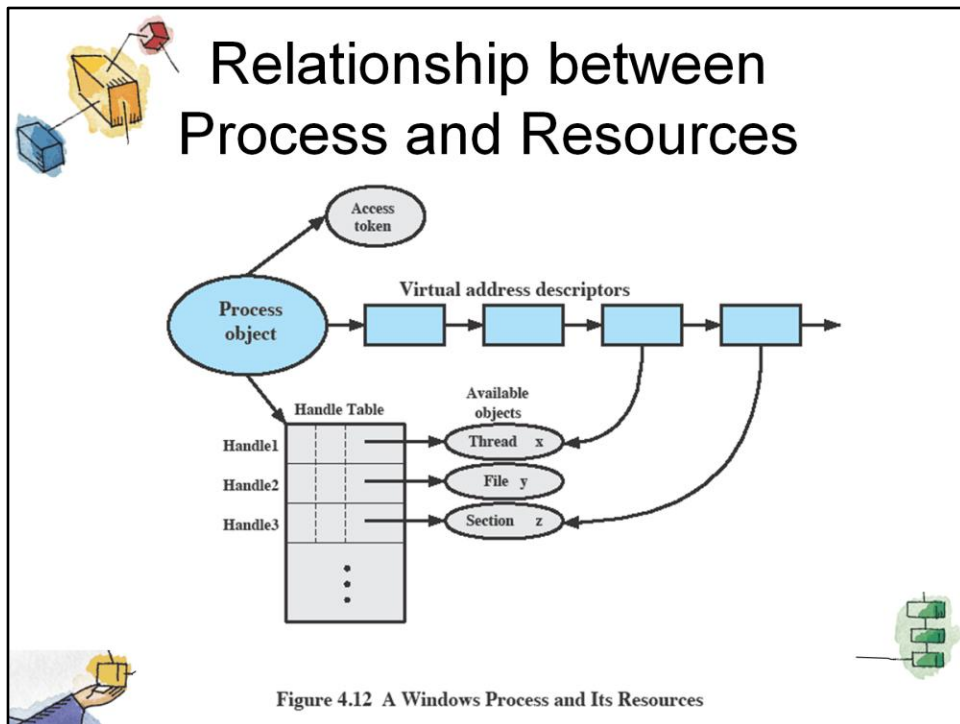


Windows Processes

- Processes and services provided by the Windows Kernel are relatively simple and general purpose
 - Implemented as objects
 - An executable process may contain one or more threads
 - Both processes and thread objects have built-in synchronization capabilities



The native process structures and services provided by the Windows Kernel are relatively simple and general purpose, allowing each OS subsystem to emulate a particular process structure and functionality.



This figure shows a single thread.

In addition, the process has access to a file object and to a section object that defines a section of shared memory.

TOKEN

Each process is assigned a **security access token**, called the primary token of the process.

When a user first logs on, Windows creates an access token that includes the security ID for the user.

Every process that is created by or runs on behalf of this user has a copy of this access token.

The token is used by windows to validate the user's ability to access secured objects or to perform restricted functions on the system and on secured objects.

The access token controls whether the process can change its own attributes.

In this case, the process does not have a handle opened to its access token. If the process attempts to open such a handle, the security system determines whether this is permitted and therefore whether the process may change its own attributes.

ADDRESS SPACE

A series of blocks that define the virtual address space currently assigned to this process.

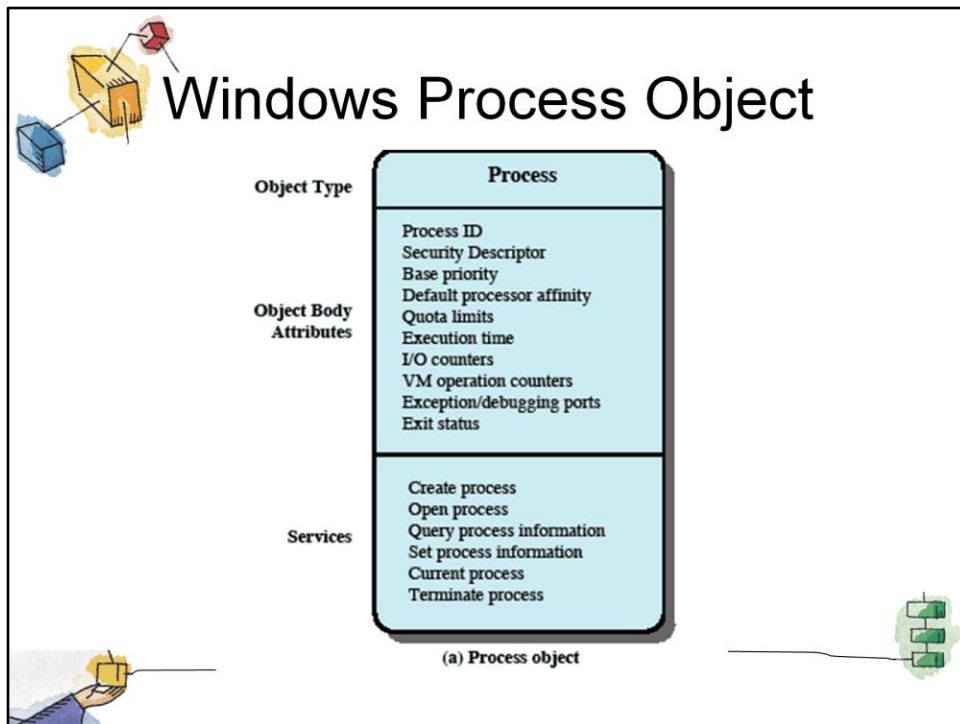
The process cannot directly modify these structures but must rely on the virtual memory manager, which provides a memory allocation

service for the process.

OBJECT TABLE

The process includes an object table, with handles to other objects known to this process.

One handle exists for each thread contained in this object.



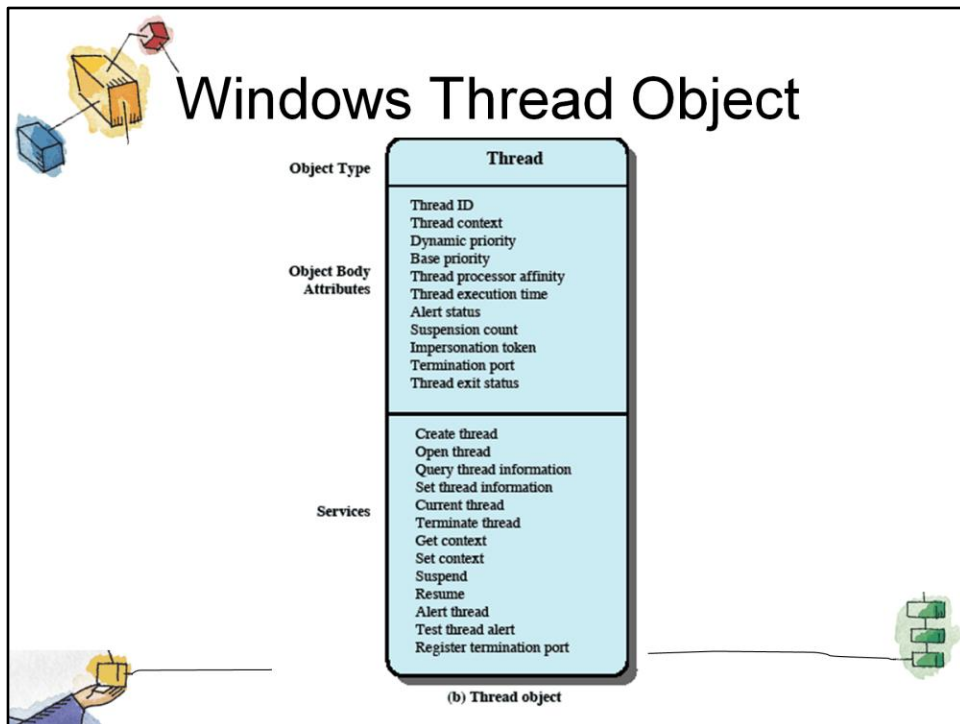
Each Windows process is represented by an object whose general structure is shown in this figure

The object-oriented structure of Windows facilitates the development of a general purpose process facility.

Windows makes use of two types of process-related objects: processes and threads.

A process corresponds to a user job or application that owns resources, such as memory, and opens files.

A thread is a dispatchable unit of work that executes sequentially and is interruptible, so that the processor can turn to another thread.

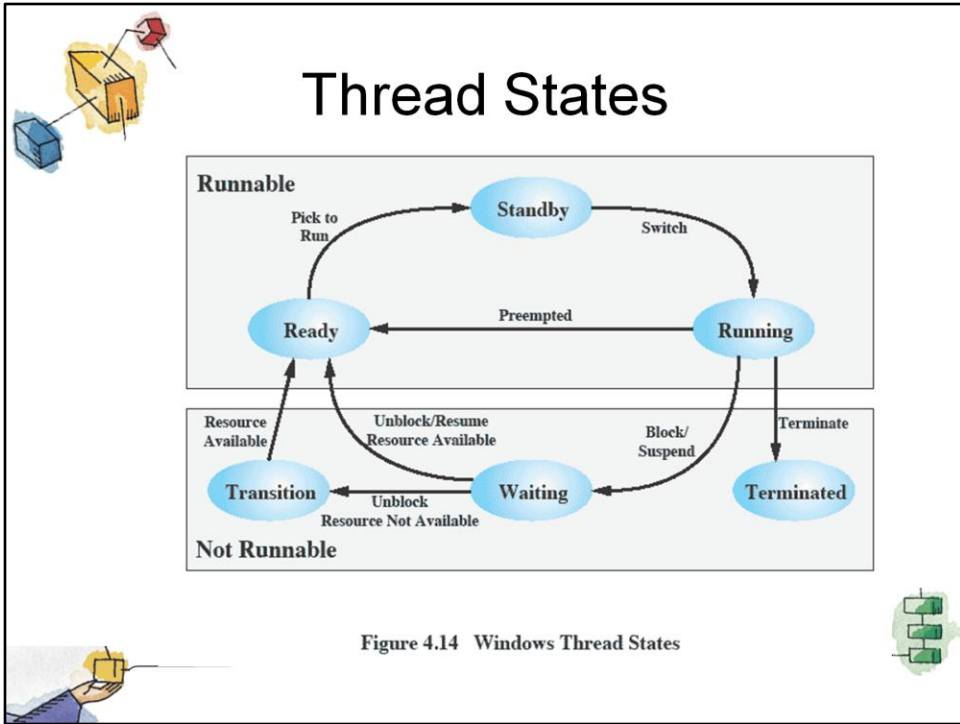


This figure depicts the object structure for a thread object.

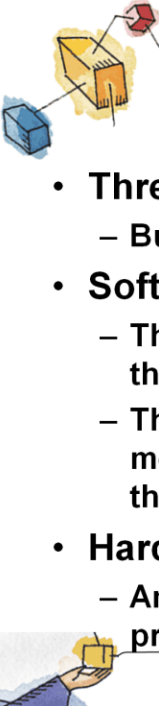
A Windows process must contain at least one thread to execute.

That thread may then create other threads.

In a multiprocessor system, multiple threads from the same process may execute in parallel.




An existing Windows thread is in one of six states



Windows SMP Support

- **Threads can run on any processor**
 - But an application can restrict affinity
- **Soft Affinity**
 - 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.
- **Hard Affinity**
 - An application restricts threads to certain processor



The threads of any process, including those of the executive, can run on any processor.

In the absence of affinity restrictions, the microkernel assigns a ready thread to the next available processor.

As a default, the microkernel 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 to certain processors (hard affinity).



Solaris

- Solaris implements multilevel thread support designed to provide flexibility in exploiting processor resources.
- Processes include the user's address space, stack, and process control block

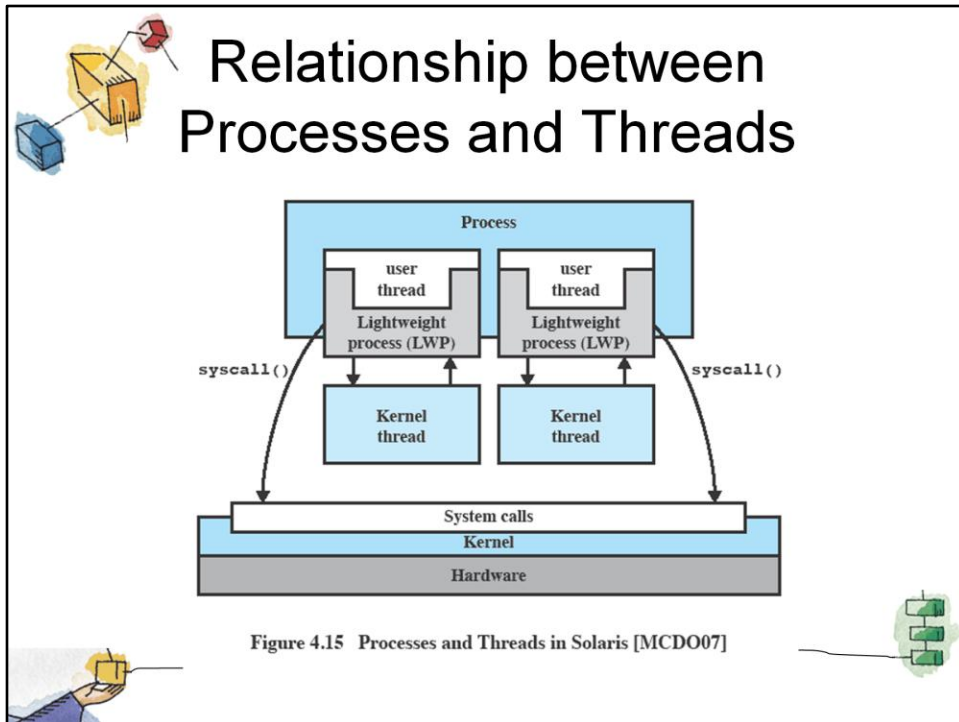


Solaris Process

- Solaris makes use of four separate 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: a mapping between ULTs and kernel threads.
 - Kernel threads

Solaris makes use of four separate thread-related concepts:

- Process:
 - includes the user's address space, stack, and process control block.
- User-level threads:
 - Implemented through a threads library in the address space of a process (invisible to the OS).
 - A user-level thread (ULT) is a user-created unit of execution within a process.
- Lightweight processes:
 - Can be viewed as a mapping between ULTs and kernel threads.
 - Each LWP supports ULT and maps to one kernel thread.
 - LWPs are scheduled by the kernel independently and may execute in parallel on multiprocessors.
- Kernel threads:
 - Fundamental entities that can be scheduled and dispatched to run on one of the system processors.



Note that there is always exactly one kernel thread for each LWP.

A process may consist of a single ULT bound to a single LWP.

In this case, there is a single thread of execution, corresponding to a traditional UNIX process.

When concurrency is not required within a single process, an application uses this process structure.

If an application requires concurrency,

- its process contains multiple threads,
- each bound to a single LWP,
- which in turn are each bound to a single kernel thread

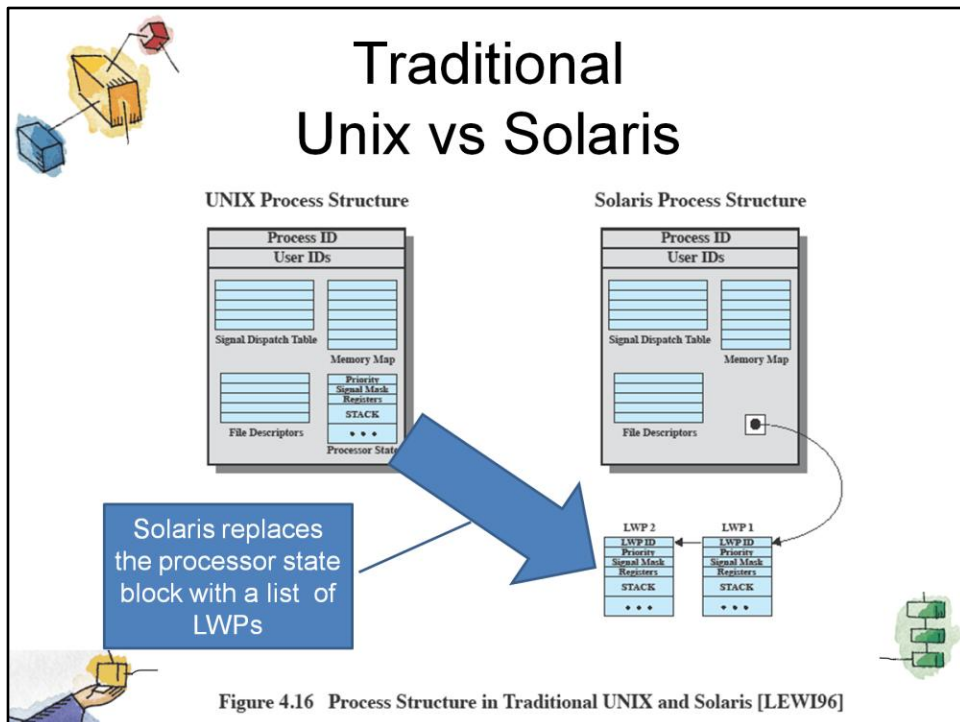


Figure 4.16 Process Structure in Traditional UNIX and Solaris [LEWI96]

Animated Slide


Point out the traditional unix structure – CLICK to emphasise the change

This figure compares, in general terms, the process structure of a traditional UNIX system with that of Solaris.

Typical UNIX implementation of a process includes



- the process ID;
- the user IDs;
- a signal dispatch table, which the kernel uses to decide what to do when sending a signal to a process;
- file descriptors, which describe the state of files in use by this process;
- a memory map, which defines the address space for this process;
- and a processor state structure, which includes the kernel stack for this process.

Solaris retains this basic structure but replaces the processor state block with a list of structures containing one data block for each LWP.



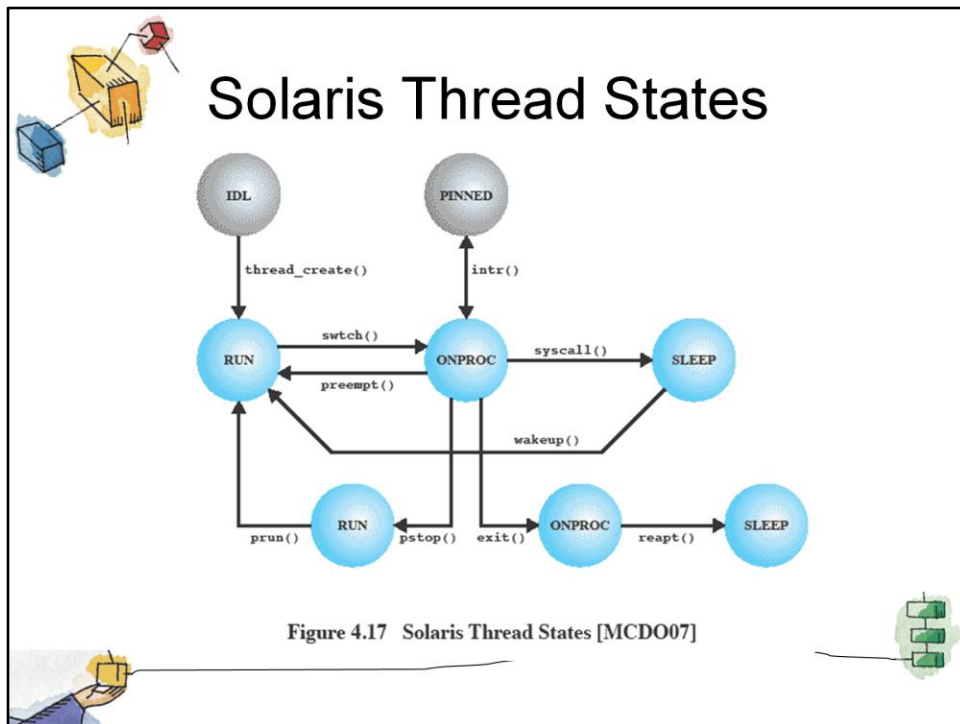
LWP Data Structure

- 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



The LWP data structure includes the following elements:

- 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 (when the LWP is not running)
- 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




A **simplified** view of both thread execution states.

These states reflect the execution status of both a kernel thread and the LWP bound to it.

- Some kernel threads are not associated with an LWP;
- the same execution diagram applies.



The states are as follows:

- **RUN:**
 - The thread is runnable; that is, the thread is ready to execute.
- **ONPROC:**
 - The thread is executing on a processor.
- **SLEEP:**
 - The thread is blocked.
- **STOP:**
 - The thread is stopped.
- **ZOMBIE:**
 - The thread has terminated.
- **FREE:**
 - Thread resources have been released
 - and** the thread is awaiting removal from the OS thread data structure.



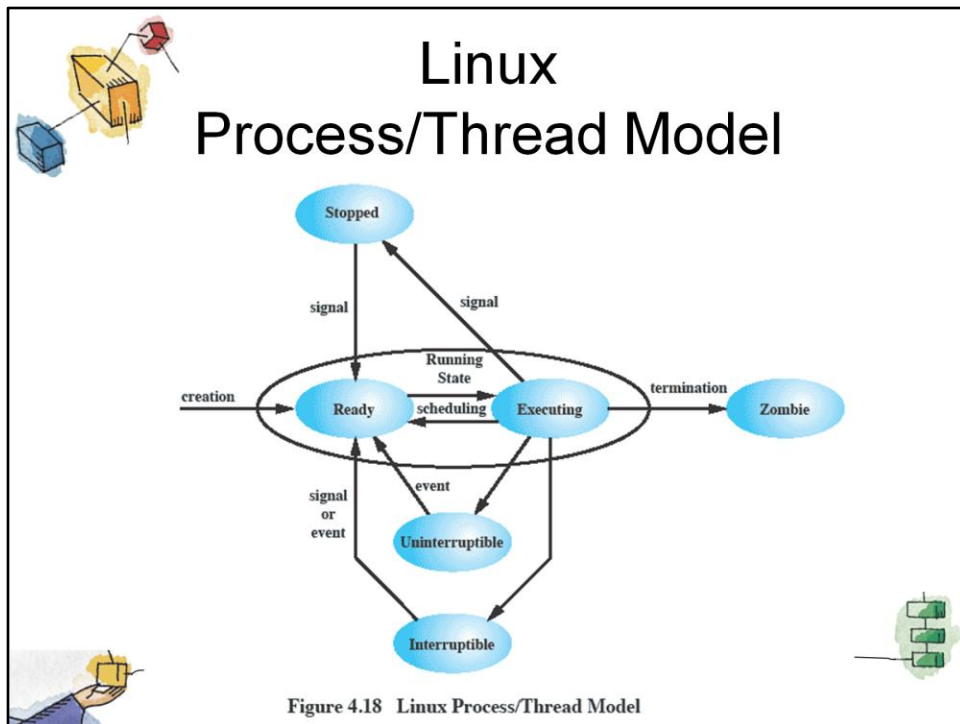
Linux Tasks

- A process, or task, in Linux is represented by a `task_struct` data structure
- This contains a number of categories including:
 - State
 - Scheduling information
 - Identifiers
 - Interprocess communication
 - And others



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

The `task_struct` data structure contains information in a number of categories:



Running:

- Corresponds to two states.
 - A Running process is either executing or
 - it is ready to execute.

Interruptible:

- A blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.

Uninterruptible:

- Another blocked state.
- The difference between the Interruptible state is that in this state, a process is waiting directly on hardware conditions and therefore will not handle any signals.

Stopped:

- The process has been halted and can only resume by positive action from another process.
- E.G., a process that is being debugged can be put into the Stopped state.

Zombie:

- The process has been terminated but, for some reason, still must have its task structure in the process table.