Operating Systems

"Threads, SMP, and Microkernels"

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 dispatching unit is referred to as a ***thread*** or lightweight process
- The entity that owns a resource is referred to as a process or ***task***
- One process / task can have one or more threads

Multithreading

- The ability of an OS to support multiple, concurrent paths of execution within a single process.

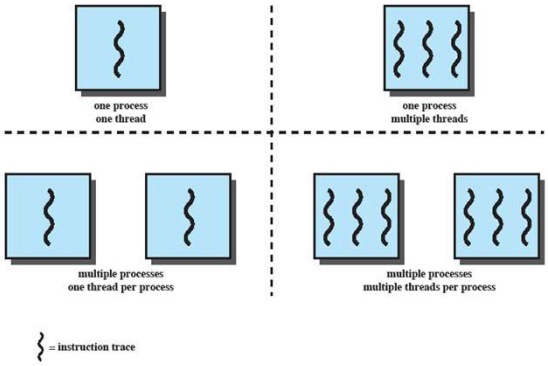


Figure 4.1 Threads and Processes [ANDE97]

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

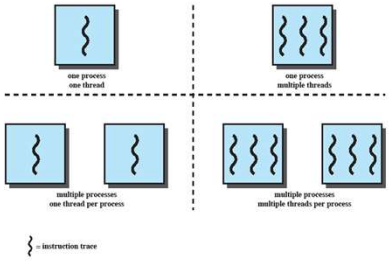


Figure 4.1 Threads and Processes [ANDE97]

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

  →**the main topic**

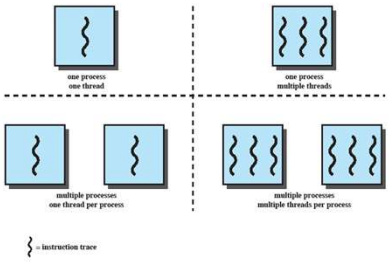


Figure 4.1 Threads and Processes [ANDE97]

Processes

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

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)

One view…

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

Threads vs. processes

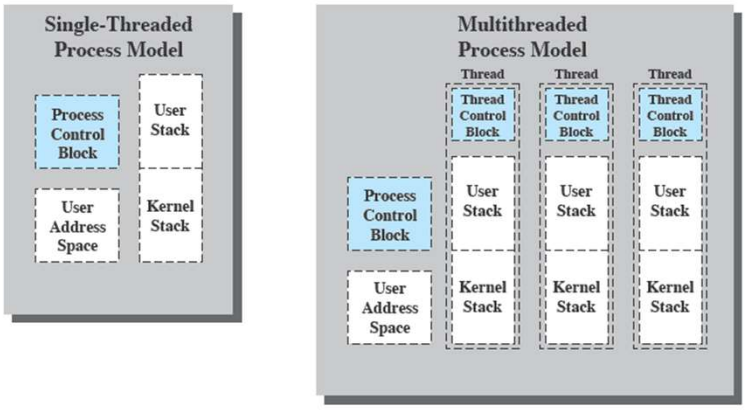


Figure 4.2 Single Threaded and Multithreaded Process Models

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

Thread use in a Single-User System

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

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

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

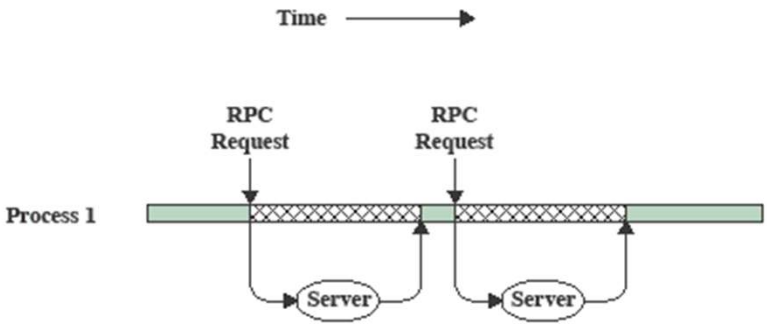
Thread Execution States

- 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

Example: Remote Procedure Call

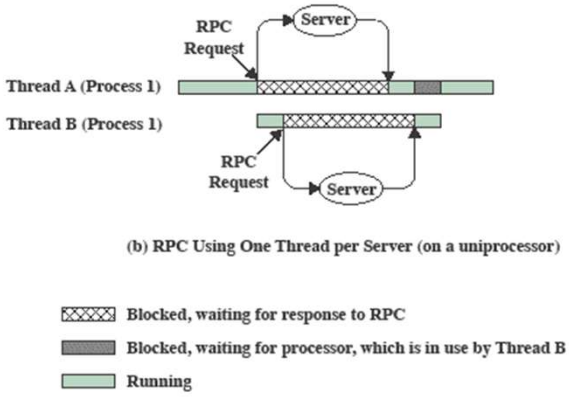
- Consider:
  - A program that performs two remote procedure calls (RPCs)
  - to two different hosts
  - to obtain a combined result.

RPC Using Single Thread



(a) RPC Using Single Thread

RPC Using One Thread per Server



Multithreading on a Uniprocessor

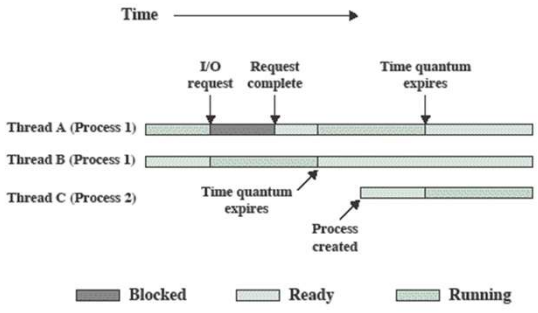


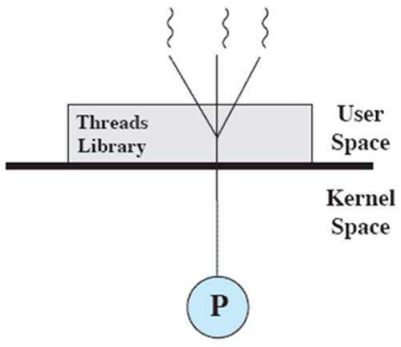
Figure 4.4 Multithreading Example on a Uniprocessor

Categories of Thread Implementation

- User Level Thread (ULT)
  - thread that are managed entirely by user-level code
  - Not requiring any support from the operating system kernel.
- Kernel level Thread (KLT) also called:
  - kernel-supported threads
  - lightweight processes.

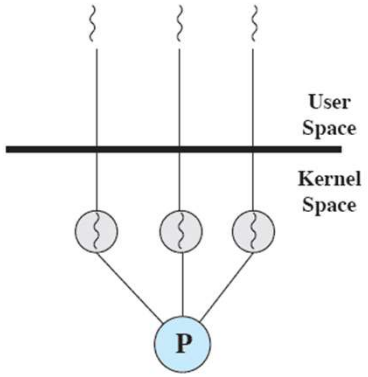
User-Level Threads

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



(a) Pure user-level

Kernel-Level Threads



(b) Pure kernel-level

- Kernel maintains context information for the process and the threads
  - No thread management done by application
- Scheduling is done on a thread basis
- Windows is an example of this approach

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

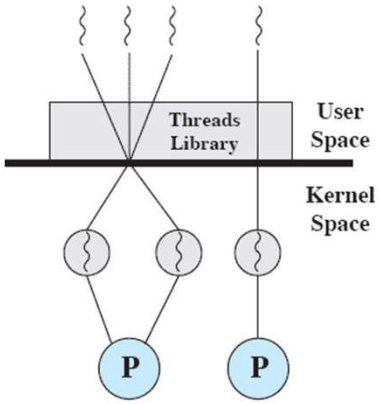
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Comparison ULT and KLT

Criteria	User-Level Threads (ULTs)	Kernel-Level Threads (KLTs)
Management	Managed entirely by user-level code, without kernel support	Managed by the operating system kernel
Scheduling	Scheduled by a user-level thread library or application, without kernel intervention	Scheduled by the operating system kernel, which may provide advanced scheduling algorithms and policies
Context Switching	Context switching occurs entirely in user space, without requiring a system call or trap into kernel mode	Context switching requires a system call or trap into kernel mode, which can incur overhead
Resources	ULTs share the same process address space and resources as the parent process	KLTs have their own kernel-level data structures, which can result in higher overhead and memory usage
Scalability	ULTs are limited to a single processor core and cannot take full advantage of multicore systems	KLTs can be assigned to different processor cores and can take full advantage of multicore systems
Synchronization	ULTs must use user-level synchronization mechanisms, which can be more efficient but may be subject to priority inversion and other issues	KLTs can use both user-level and kernel-level synchronization mechanisms, providing more flexibility and better enforcement of policies
Portability	ULTs are highly portable across different operating systems, as long as a compatible user-level thread library is available	KLTs may be less portable, as different operating systems may have different thread APIs and scheduling mechanisms

Combined Approaches

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



(c) Combined

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

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 Multi Processors (SMPs)
  - Clusters (ch 16)

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

Categories of Computer Systems

- Multiple Instruction Single Data (MISD) stream
  - 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

Parallel Processor Architectures

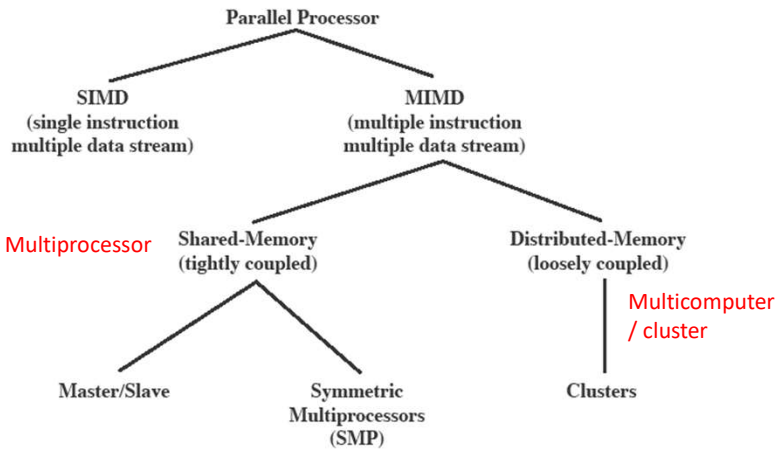


Figure 4.8 Parallel Processor Architectures

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

Typical SMP Organization

symmetricity

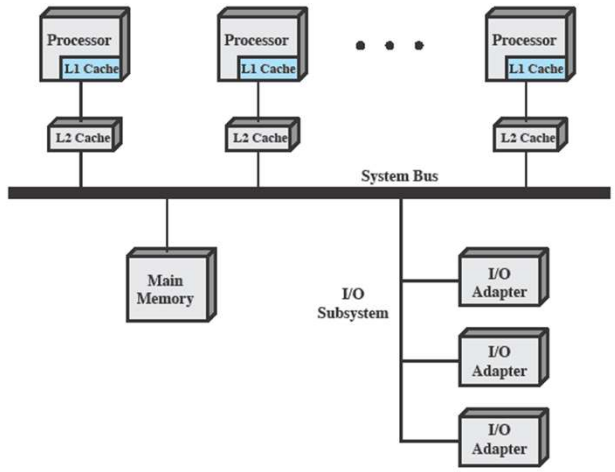


Figure 4.9 Symmetric Multiprocessor Organization

Multiprocessor OS Design Considerations

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

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.

Kernel Architecture

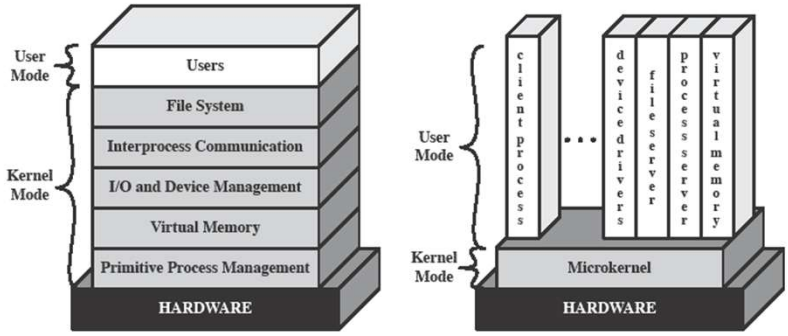


Figure 4.10 Kernel Architecture

Microkernel Design: Memory Management

- Low-level memory management - Mapping each virtual page to a physical page frame
  - Most memory management tasks occur in user space

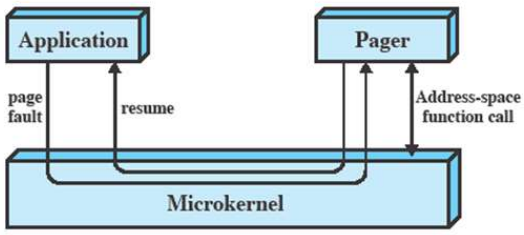


Figure 4.11 Page Fault Processing

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.

Microkernal 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.

Benefits of a Microkernel Organization

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

Comparison Microkernel vs Monolitic kernel

Criteria	Microkernel	Monolithic Kernel
Architecture	Minimalistic kernel that provides basic services, with additional services implemented as user-space processes	Entire kernel including device drivers, system calls, and other services are executed in kernel space
Security	More secure due to smaller attack surface, as most operating system services are run in user space	Exposes entire kernel to potential security vulnerabilities
Flexibility	More flexible and easier to customize and maintain as new services can be added without modifying the kernel itself	Requires modifications to kernel code to add new functionality
Performance	Generally slower due to increased reliance on interprocess communication and message passing	Better performance due to all system services being executed in kernel space, avoiding the overhead of interprocess communication
Debugging and Maintenance	Easier to debug and maintain due to modular design, where different services run as independent processes	Debugging and maintenance can be more difficult due to all services running in the same kernel space

Roadmap

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

Which OS uses microkernel / monolithic kernel

- Windows:
  - Windows NT and later versions use a hybrid kernel that combines elements of a microkernel and a monolithic kernel.
  - Earlier versions of Windows (such as Windows 95, 98, and ME) use a monolithic kernel.
- Linux:
  - a monolithic kernel, but it can also support loadable kernel modules that can be dynamically loaded and unloaded at runtime.
- macOS:
  - a hybrid kernel that combines elements of a microkernel and a monolithic kernel.
- iOS:
  - a hybrid kernel that combines elements of a microkernel and a monolithic kernel.
- Android:
  - a monolithic kernel, but it also includes a user-level component called the Binder IPC mechanism that provides some microkernel-like functionality.

Which OS uses microkernel / monolithic kernel

- some research and niche operating systems that use microkernels, such as
  - QNX, Minix, and L4 microkernel.
- Minix can be downloaded from the official website: http://www.minix3.org/download/.
  - select the version of Minix you want to download (e.g., stable or development) and the architecture (e.g., x86, ARM, or PowerPC), and the format (e.g., ISO or USB image).
- Install Minix on virtual machine (VM) such as VirtualBox and VMware

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

Relationship between Process and Resources

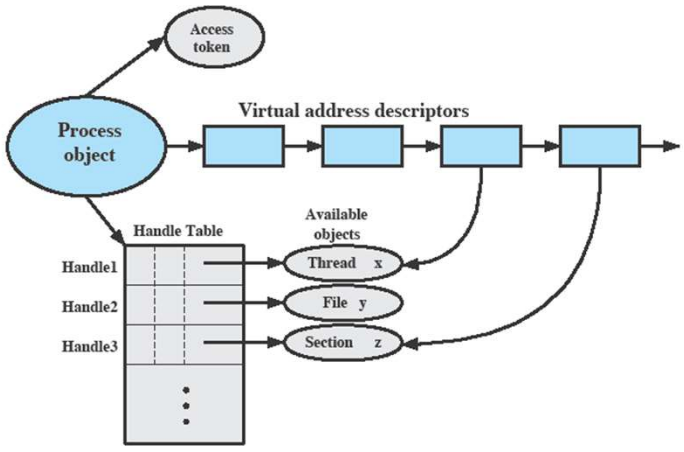
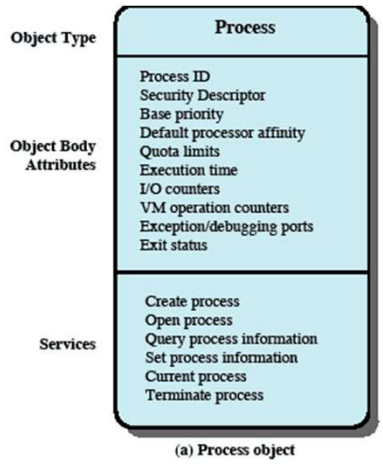


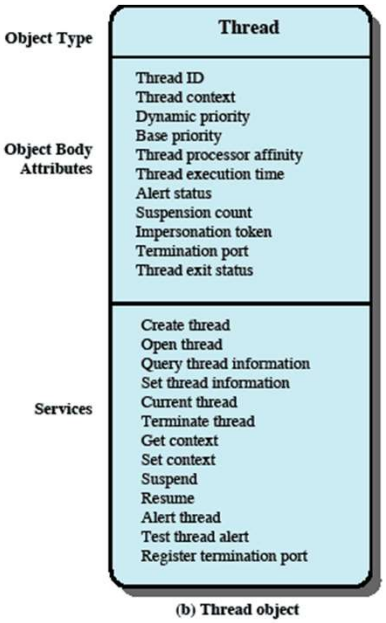
Figure 4.12 A Windows Process and Its Resources

Windows Process Object



(a) Process object

Windows Thread Object



(b) Thread object

Thread States

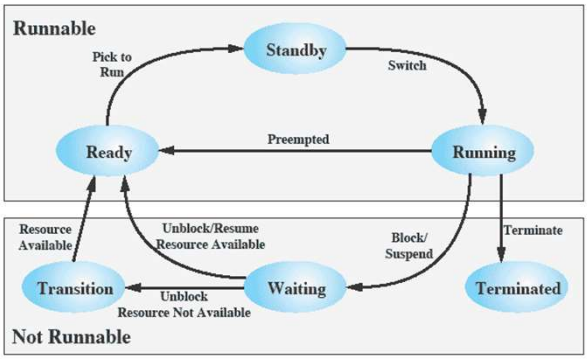


Figure 4.14 Windows Thread States

Windows SMP Support

- Threads can run on any processor
  - But an application can restrict affinity / natural liking
- 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

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

Relationship between Processes and Threads

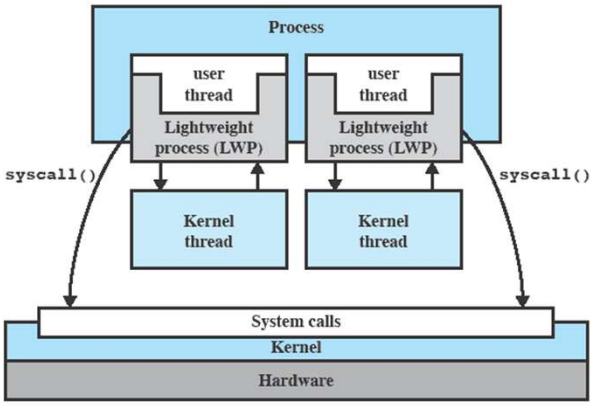


Figure 4.15 Processes and Threads in Solaris [MCDO07]

Traditional Unix vs Solaris

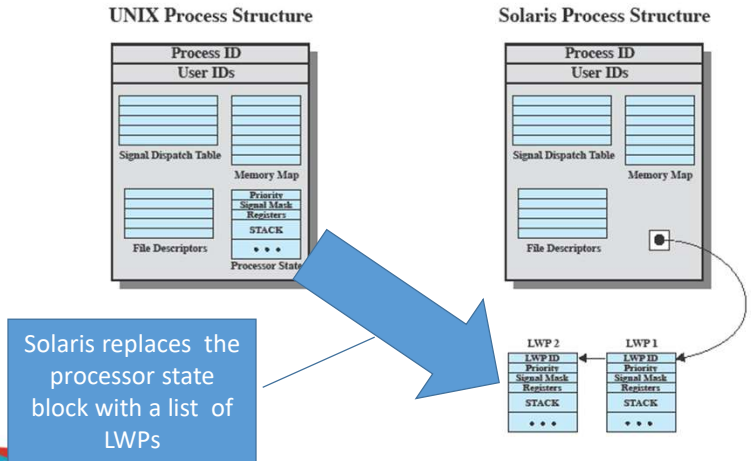


Figure 4.16 Process Structure in Traditional UNIX and Solaris [LEWI96]

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

Solaris Thread States

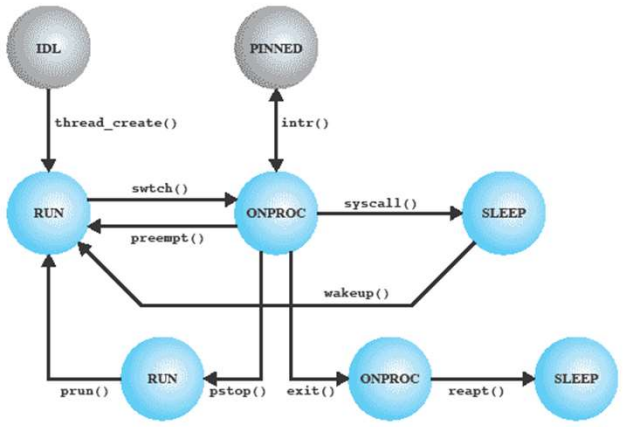


Figure 4.17 Solaris Thread States [MCDO07]

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

Linux Process/Thread Model

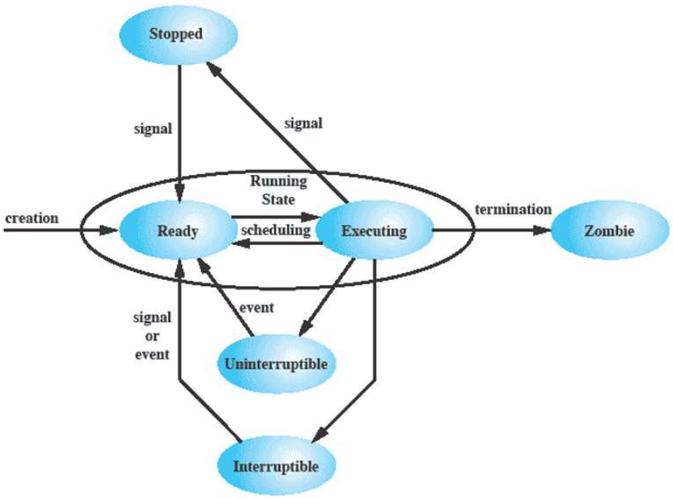


Figure 4.18 Linux Process/Thread Model

Thank You