Chapter 4
Threads, SMP, and Microkernels

(based on original slides by Pearson)

Concepts about Processes so far

- 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

  → how?

Process

- Dispatching is referred to as a thread or lightweight process
- Resource of ownership is referred to as a process or task

Multithreading

- 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
- Windows, Solaris, Linux, Mach, and OS/2 support multiple threads

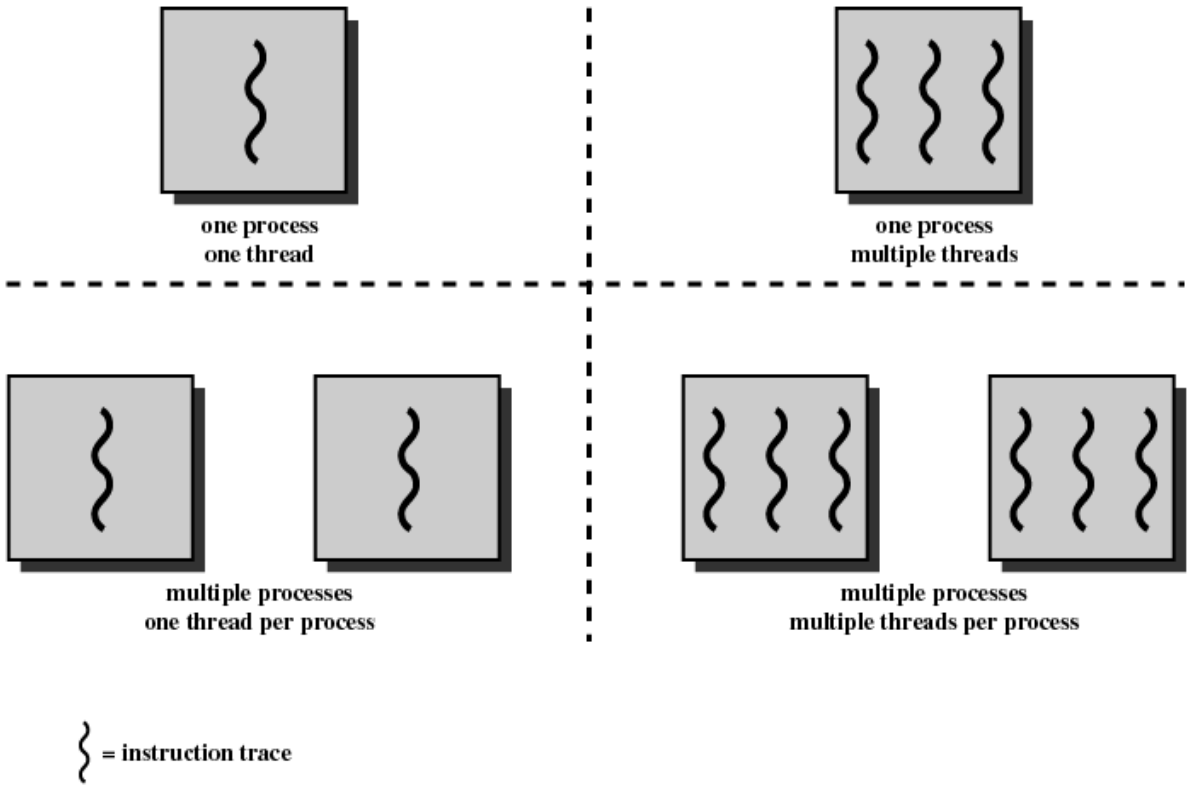


Figure 4.1 Threads and Processes [ANDE97]

Processes vs Threads

- Processes:
  - Have a virtual address space which holds the process image
  - Protected access to processors, other processes, files, and I/O resources

- Threads (within a process, each has…):
  - 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

    Hint: All threads within a process share this!

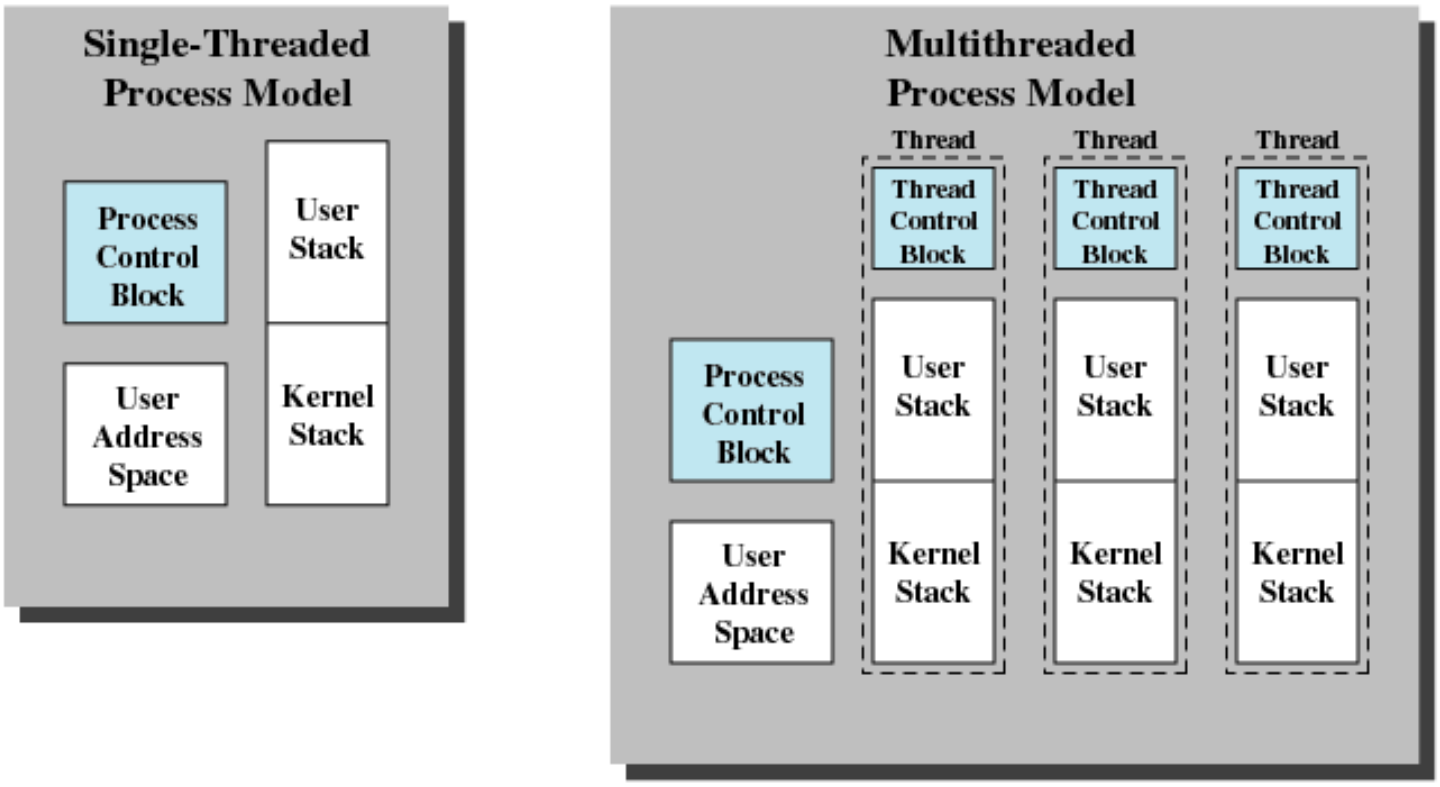


Figure 4.2 Single Threaded and Multithreaded Process Models

We break up threads into individual process blocks and allocate to each a user stack and kernel stack. Wee need to maintain a thread control block in the process image that keeps track of its status.

Benefits of Threads

- Takes less time to create a new thread than a process
  - Can skip resource allocation through the kernel
  - Factor 10 times faster
- Less time to terminate a thread than a process
  - Don’t have to release resources through the kernel
- Less time to switch between two threads within the same process
- More efficient communication between multiple execution entities
  - Threads within the same process share memory

Threads are faster to create than processes. We dont need to allocate resources since we are just using the ones allocated to the process. Similarly we can easily terminate, all we need to do is free any local memory. Also easier to switch between the two (dont even need a system call). Communicate between threads is much easier since they share resources (global variables rock).

Uses of Threads in a Single-User Multiprocessing System

- Foreground to background work
  - T1 handles sampling, T2 background checks, T3 data processing
- Asynchronous processing
  - T1 computes everything, T2 passes through RS232
- Speed of execution
  - Multiple threads on multiple CPUs/cores; I/O blocking doesn't stop the app, only one thread (e.g., printing and Word 6, webserver, DB server)
- Modular program structure

These are examples of threads in a single users multiprocessing system. Wee can break up threads by the resources that they use to make them more efficient like with foreground to background work. For asynchronous processing we have a computation thread and a communication thread.

Thread Behavior in Processes

- 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

Since everything in the process resides in the same regieon we need to suspend all threads when the process is suspended, similarly for termination.

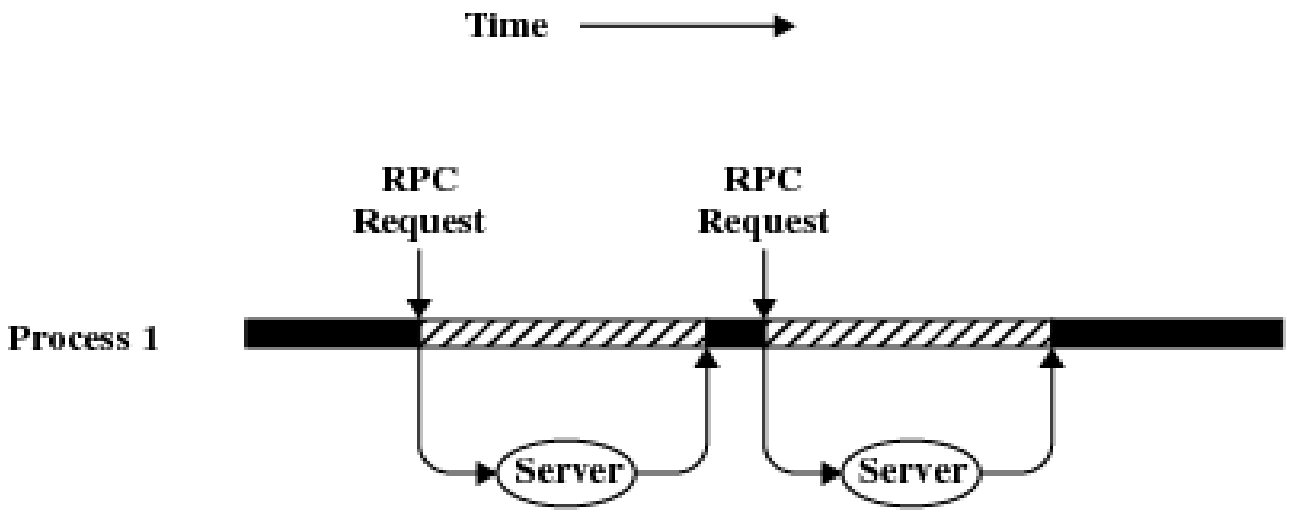
Thread States

- States associated with a change in thread state
  - Spawn
    - Spawn another thread
  - Block
  - Unblock
  - Finish
    - Deallocate register context and stacks

trhead states:

- spawning: the creation of threads
- blocking and unblocking: waiting on resources
- finish: free all used memory

Remote Procedure Call Using Single Thread



(a) RPC Using Single Thread

Remote Procedure Call Using Threads

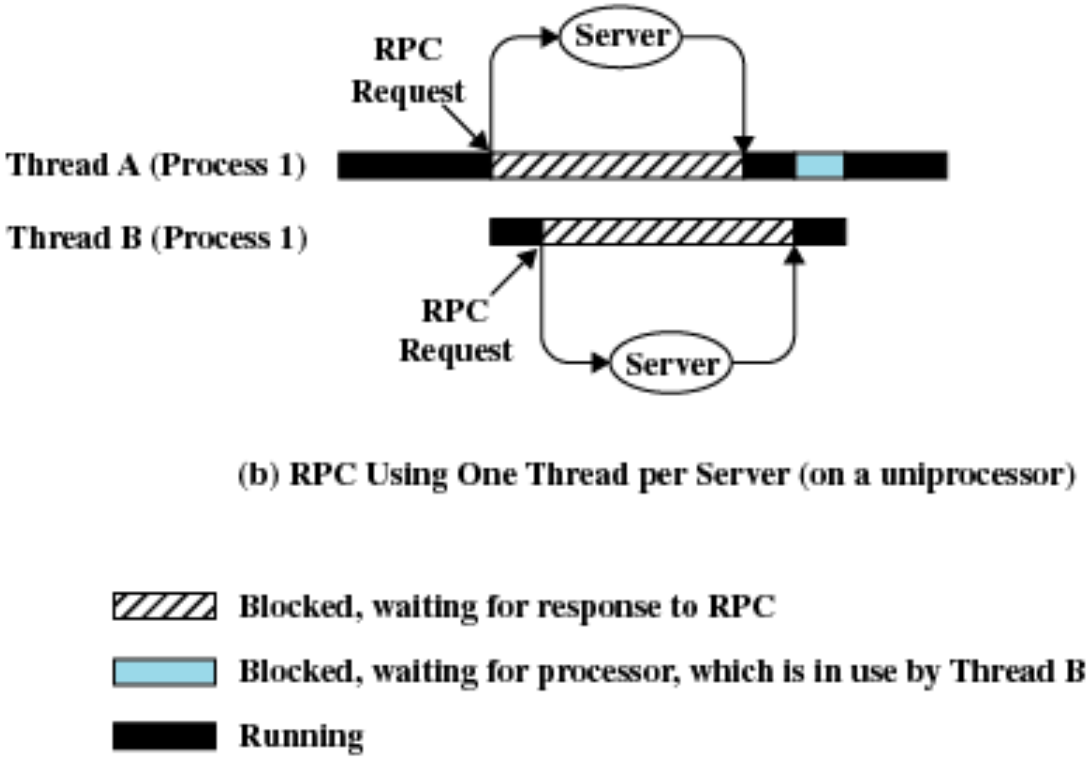


Figure 4.3 Remote Procedure Call (RPC) Using Threads

Assuming independence of the function calls!

We use multithreading to get around network delays when making RPCs from the web.

Multithreading

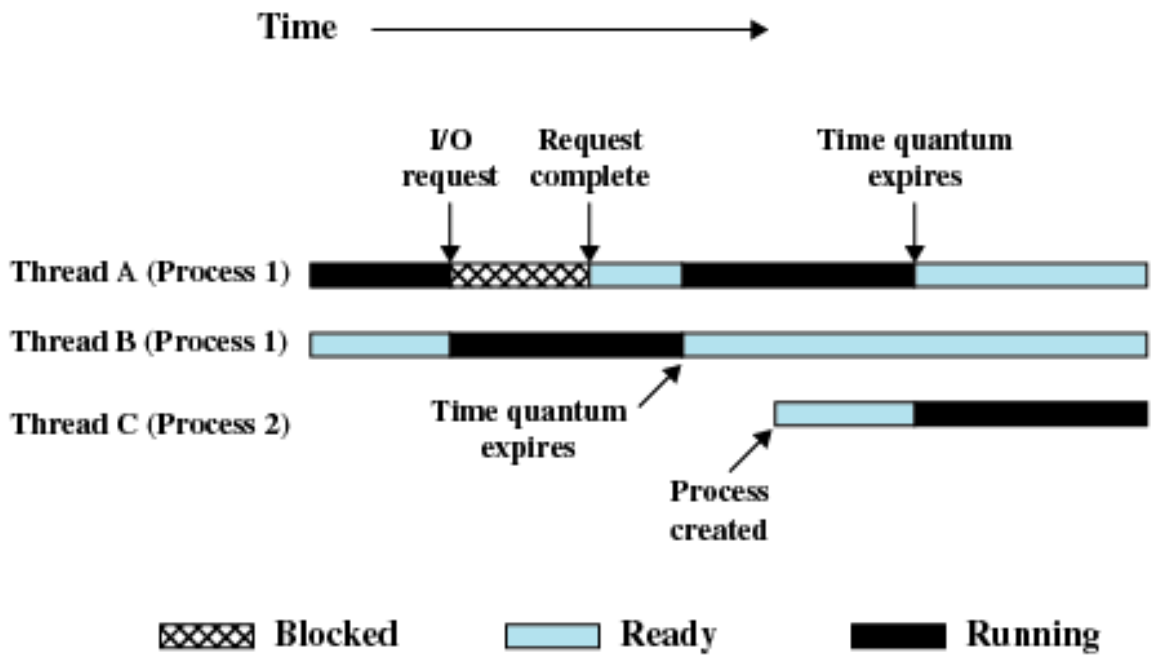


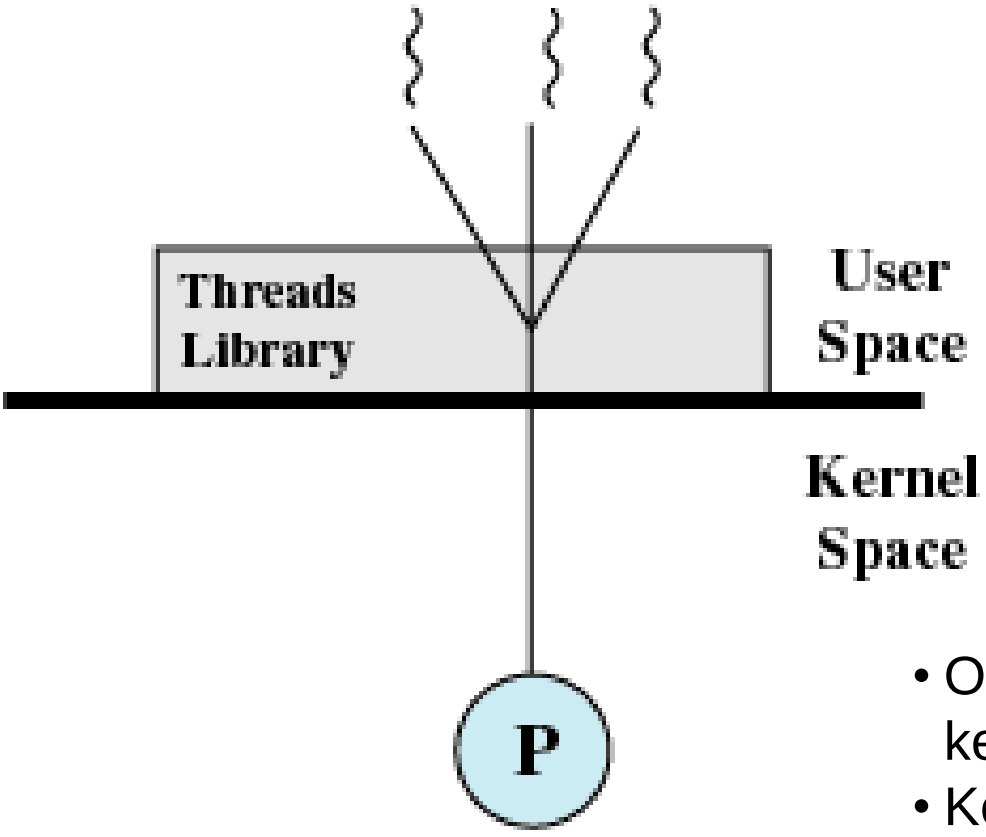
Figure 4.4 Multithreading Example on a Uniprocessor

User-Level Threads

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

This is a paradigm that the user should now know about threats to threads.

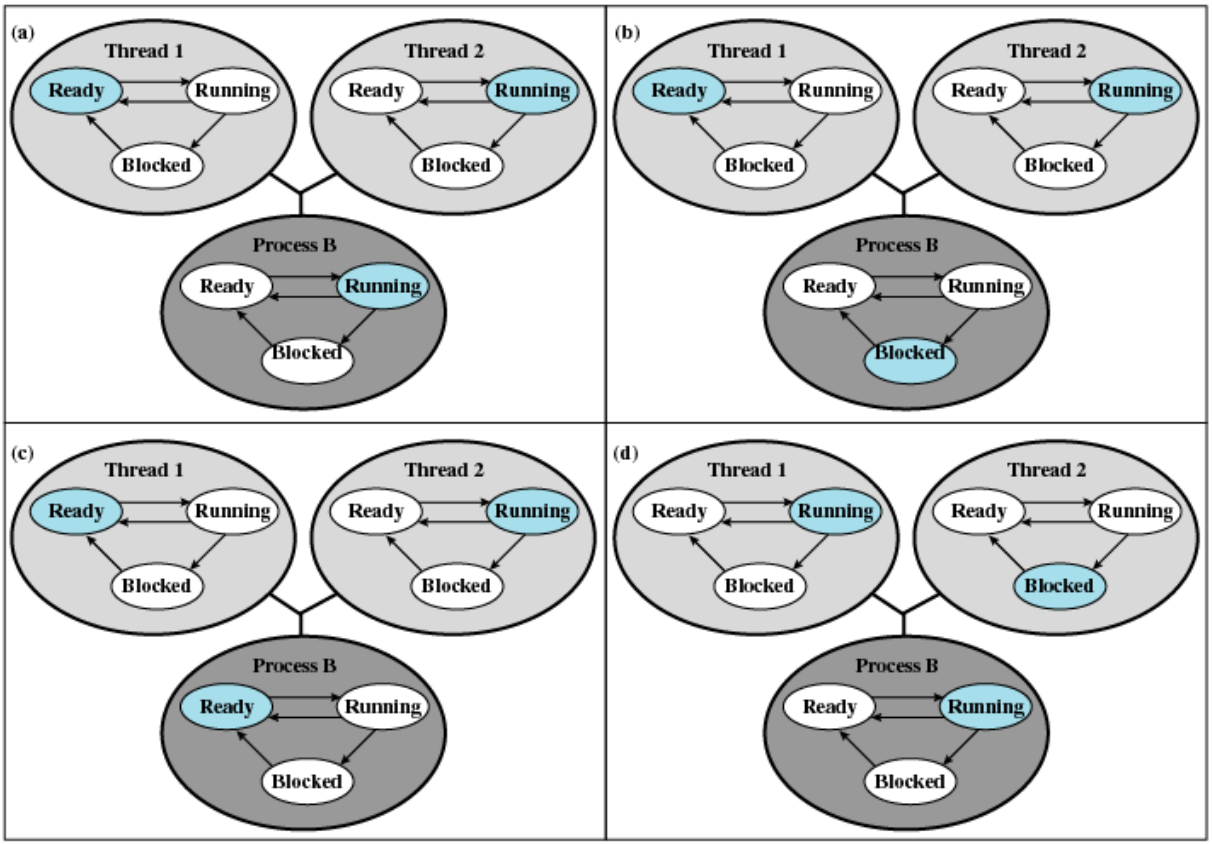
User-Level Threads



- Many threads in the application.
- Application schedules execution.

- One process in the kernel.
- Kernel schedules processes.

Load a threading library that makes threads and handles their states. We can augment OS's without threading by using threading libraries that manage things for them as programs.



Colored state
is current state

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

T2 calls a kernel I/O function; kernel blocks the process.

T2 waits for something other than kernel (e.g., T1)

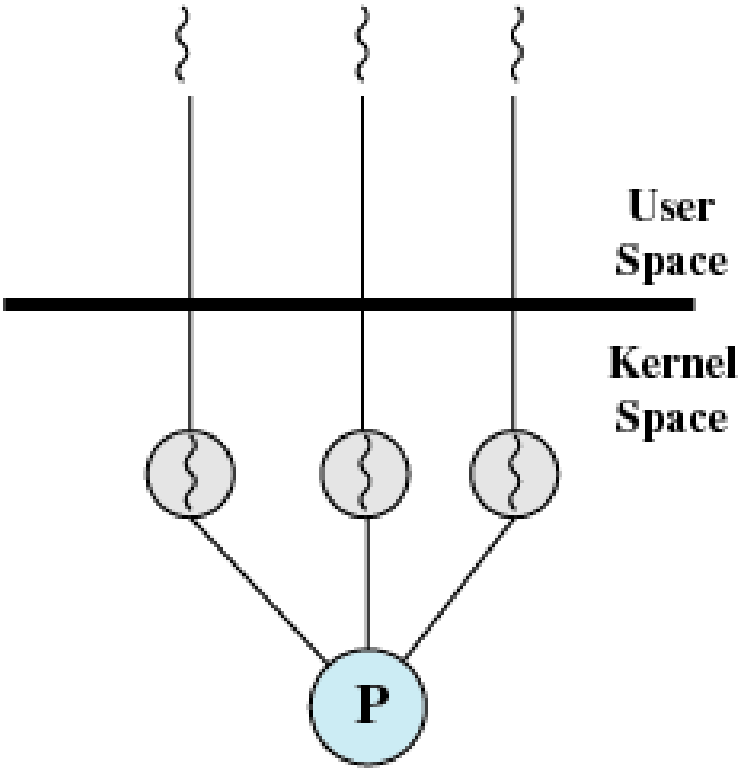
The threading library manages the states of threads. When a thread makes a system call (at b) the threading library doesnt know about that) the process gets blocked, but since the OS doesnt manage threads it doesnt know that the threads are associated with the process the thread is left as running while the process is blocked. There can be confusion between how the threading library and OS see execution. The biggest problem between the threading libraries and OS's is that when the OS doesnt support threading multicores are rendered useless for optimizing the process. We like kernel level threads (ones that the OS knows about). More housekeeping is done by the OS.

Kernel-Level Threads

- Kernel maintains context information for the process and the threads
- Scheduling is done on a thread basis

Now that the OS knows about it we can schedule threads more efficiently (woooo multicores).

Kernel-Level Threads



(b) Pure kernel-level

- Application consists of sets of threads.

- Kernel knows processes & threads.
- Kernel schedules processes & threads.

We have multiple threads in the user space and all of the resource management is done by the process still, the only thing that has chagned is that the thread context and its management has been moved to the kernel stack to be managed by the OS.

Comparison

- ULT:
  - Less switching overhead (save 2 mode switches)
  - Scheduling is app specific
  - ULT can run on any OS
- KLT:
  - OS calls are blocking only the thread
  - Can schedule threads simultaneously on multiple processors

VAX Running UNIX-Like Operating System

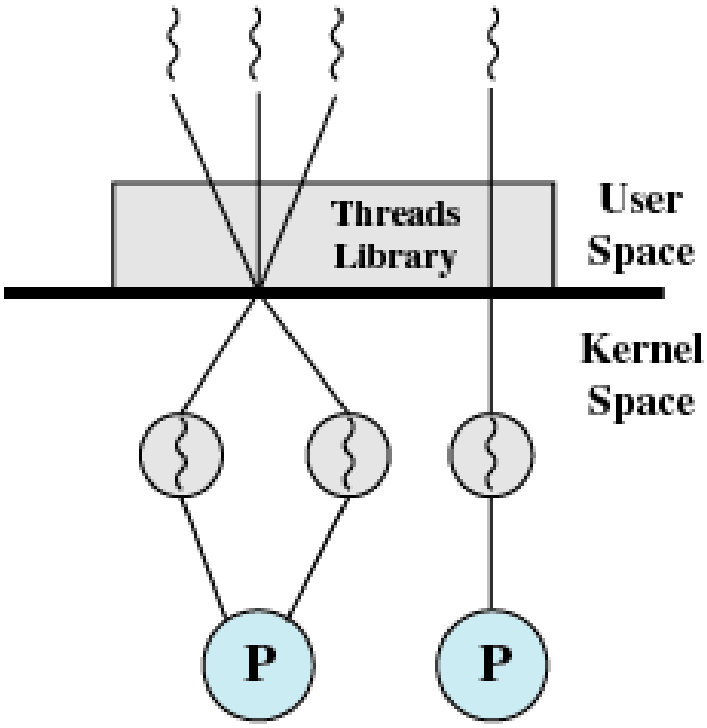
Table 4.1 Thread and Process Operation Latencies (μs) [ANDE92]

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

Null Fork measures the system overhead of the fork() call.
Signal Wait measures the system overhead of the signal() call.

User level threads are much faster than kernel level threads, but this image is suuuuuuuuper old. Now adays we are much better at managing threads and taking advantage of multicores so kernel level threads are actually faster now.

Combined Approaches



(c) Combined

Example: IO processes.

- Applications consists of multiple threads.
- Threads can be grouped to kernel threads.

- Kernel knows processes & threads.
- Kernel schedules processes & threads.

Really this is the best approach. User level knows about all threads, but they are grouped together as kernel threads. This means that the kernel has less threads to manage but still has ones to distribute across all cores (we create x kernel threads where x is the number of cores) while still allowing the user to have as many user level threads as they'd like so that they can make their application as modular and efficient as they can. here we get the best of both worlds yaaay compromise. Note: this requires a compliant OS (doesnt work on all OS's).

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

We can have multiple processes associated with the same thread. And really any many/1 relationship between threads and processors.

HOMEWORK: look at **pthreads**

Microkernels

- 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
- How many system calls in L4?

Instead of …

Use this …

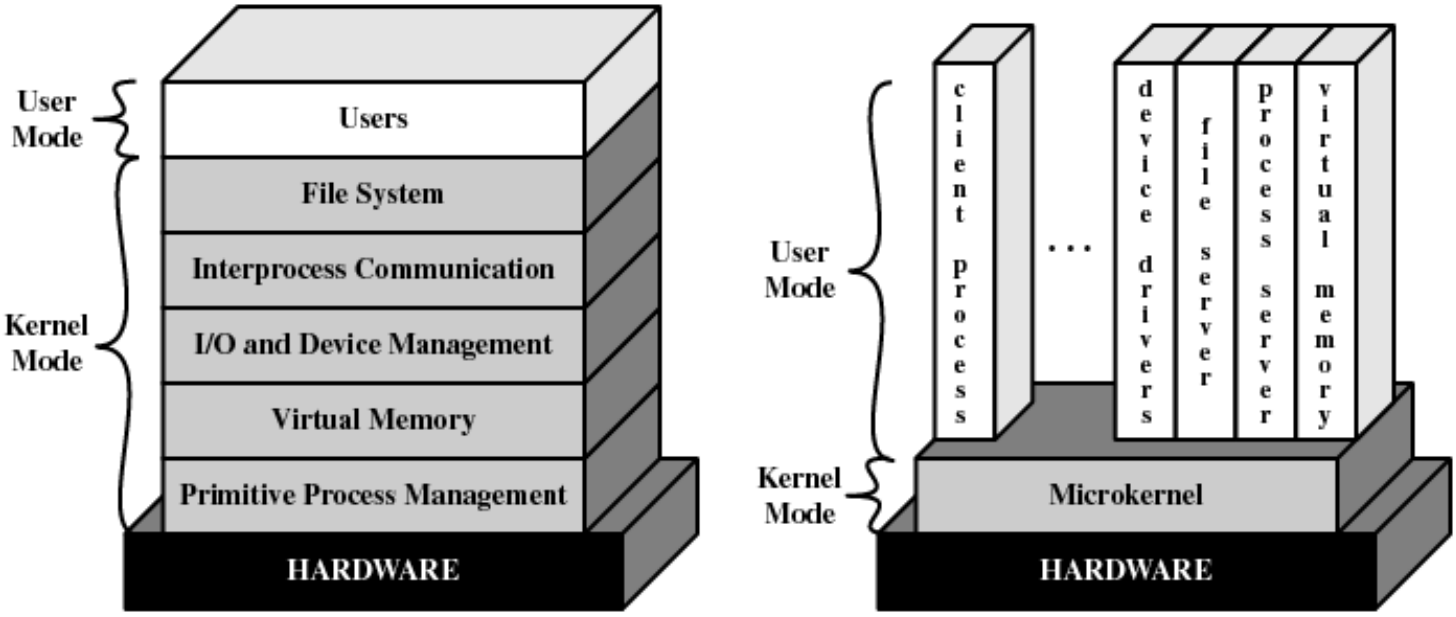


Figure 4.10 Kernel Architecture

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
- Flexibility
  - New features added
  - Existing features can be subtracted

Benefits of a Microkernel Organization

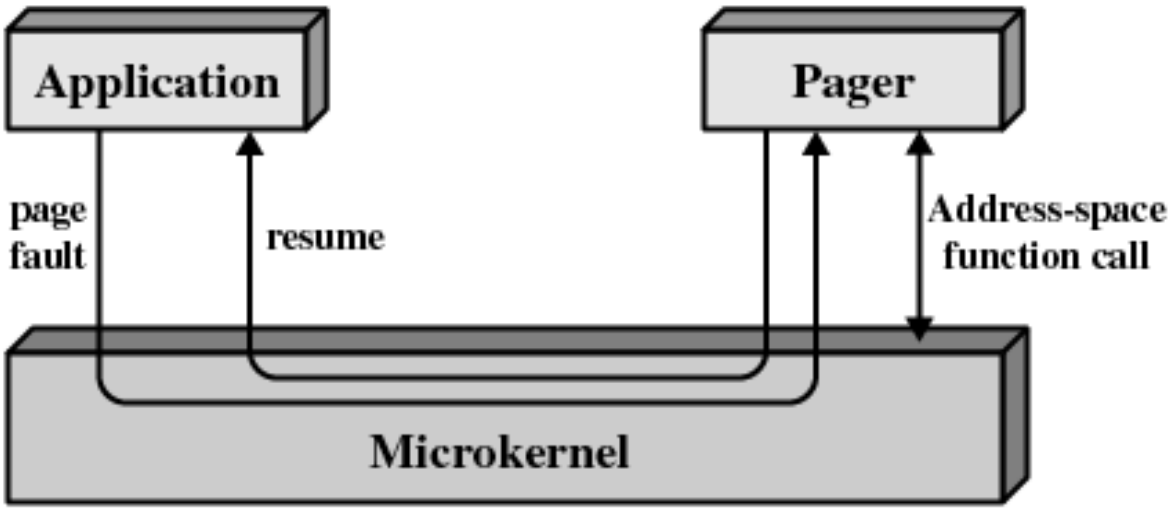
- Portability
  - Changes needed to port the system to a new processor is changed in the microkernel - not in the other services
- Reliability
  - Modular design
  - Small microkernel can be rigorously tested
  - Crashing a module does not crash the kernel

Benefits of Microkernel Organization

- Distributed system support
  - Message are sent without knowing what the target machine is
  - Opens up new applications types
- Object-oriented operating system
  - Components are objects with clearly defined interfaces that can be interconnected to form software

Microkernel Design

- Low-level memory management
  - Mapping each virtual page to a physical page frame



Handling page faults

Grant/map/flush calls

Microkernel Design

- Interprocess communication
  - Concepts: messages, ports, capabilities
  - Microkernel requires copying of messages; remapping pages may be faster
- I/O and interrupt management
  - Interrupts are messages sent to processes
  - Microkernel doesn’t need to know anything about the IRQ handling function

Subsequent slides are for private study

Windows Processes

- Implemented as objects
- An executable process may contain one or more threads
- Both processes and thread objects have built-in synchronization capabilities

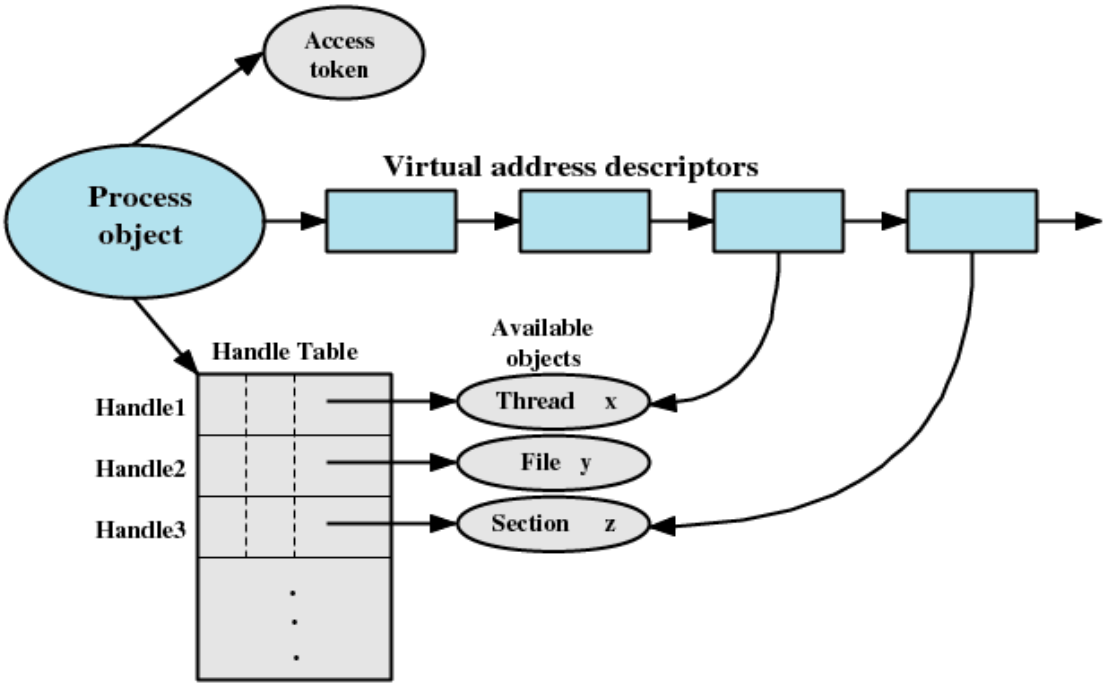
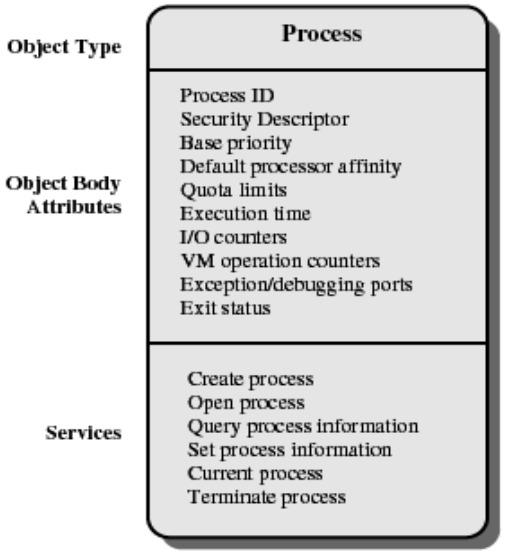


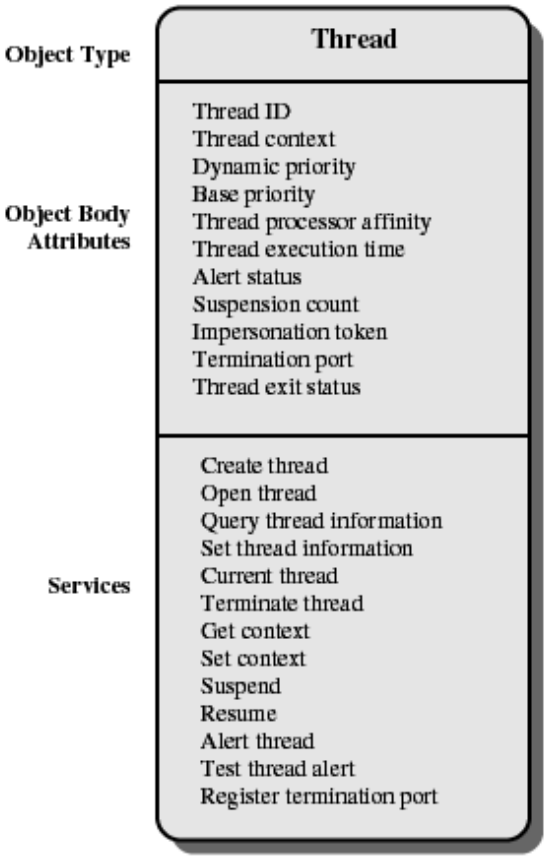
Figure 4.12 A Windows Process and Its Resources

Windows Process Object



(a) Process object

Windows Thread Object



(b) Thread object

Windows 2000 Thread States

- Ready
- Standby
- Running
- Waiting
- Transition
- Terminated

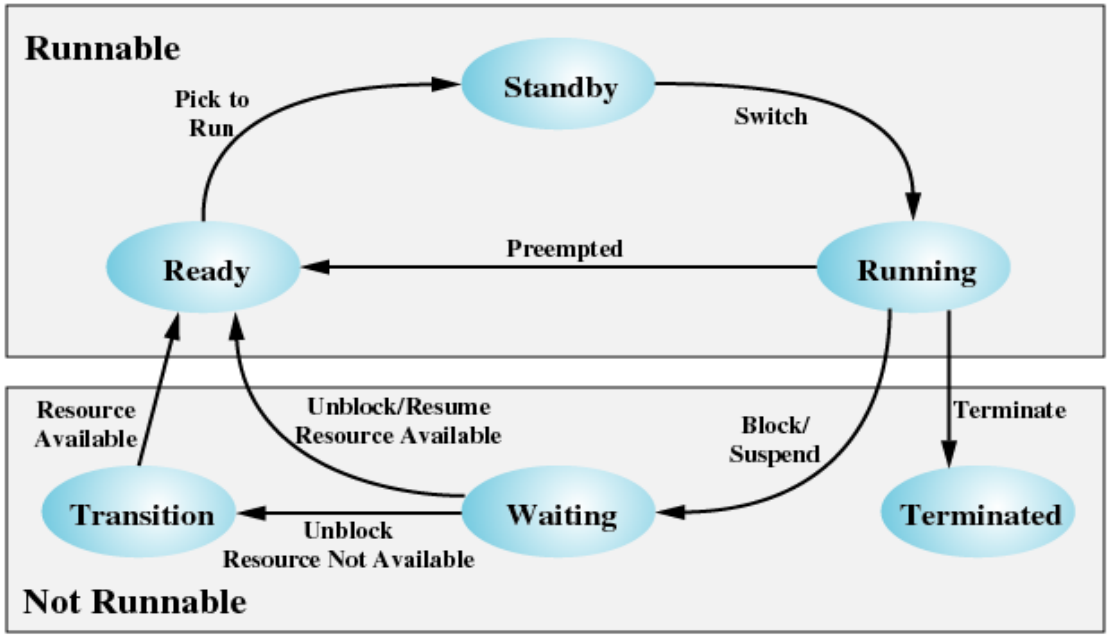


Figure 4.14 Windows Thread States

Solaris

- Process includes the user's address space, stack, and process control block
- User-level threads
- Lightweight processes (LWP)
- Kernel threads

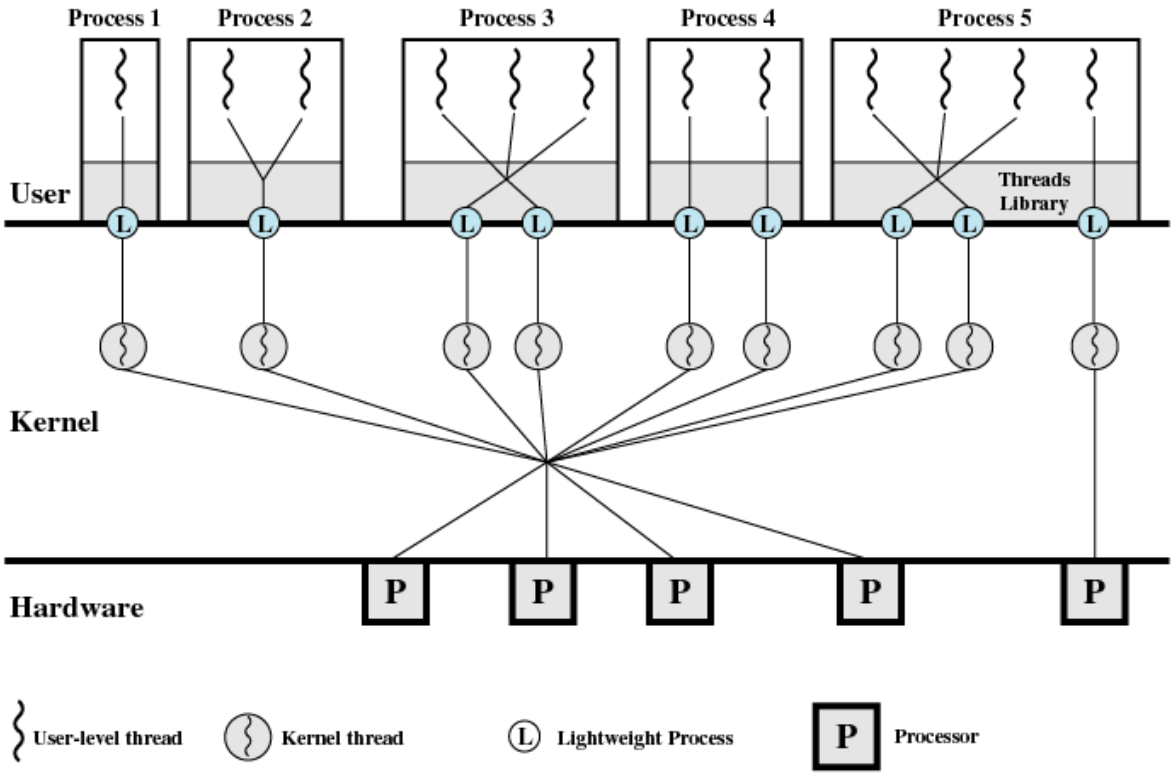


Figure 4.15 Solaris Multithreaded Architecture Example

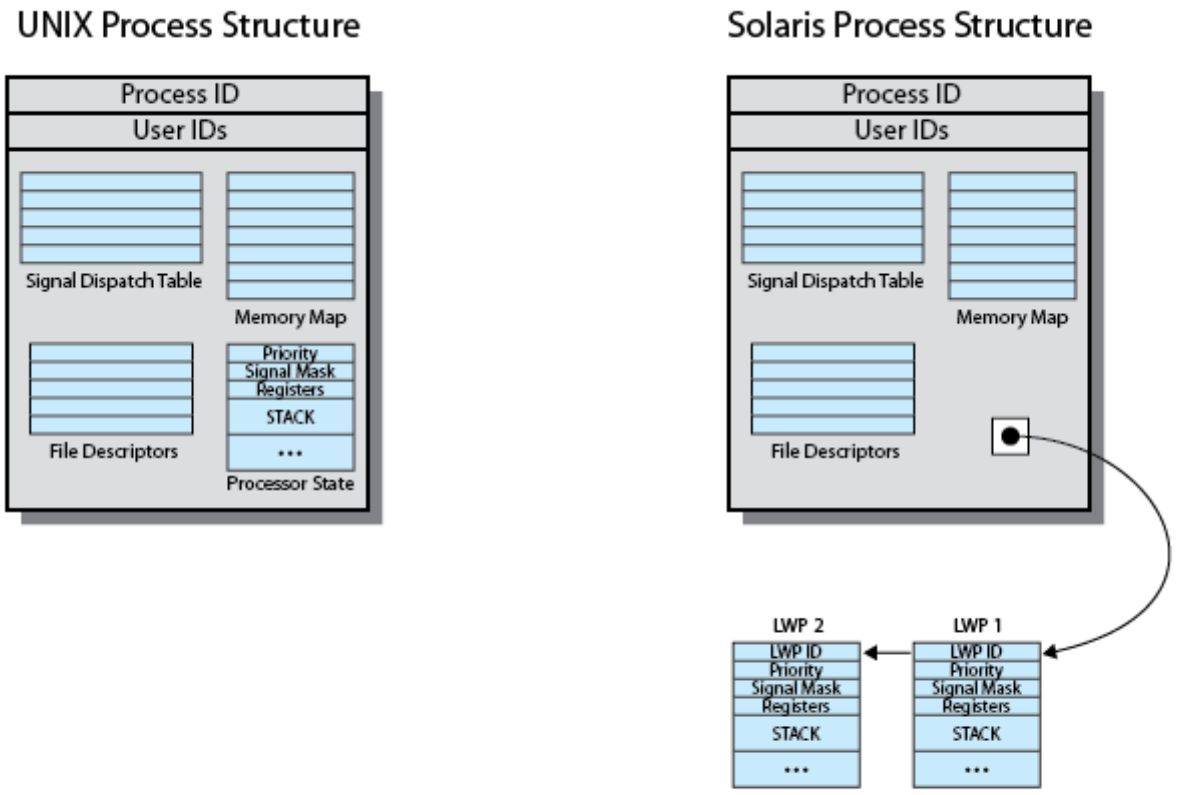


Figure 4.16 Process Structure in Traditional UNIX and Solaris [LEWI96]

Solaris Lightweight Data Structure

- Identifier
- Priority
- Signal mask
- Saved values of user-level registers
- Kernel stack
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

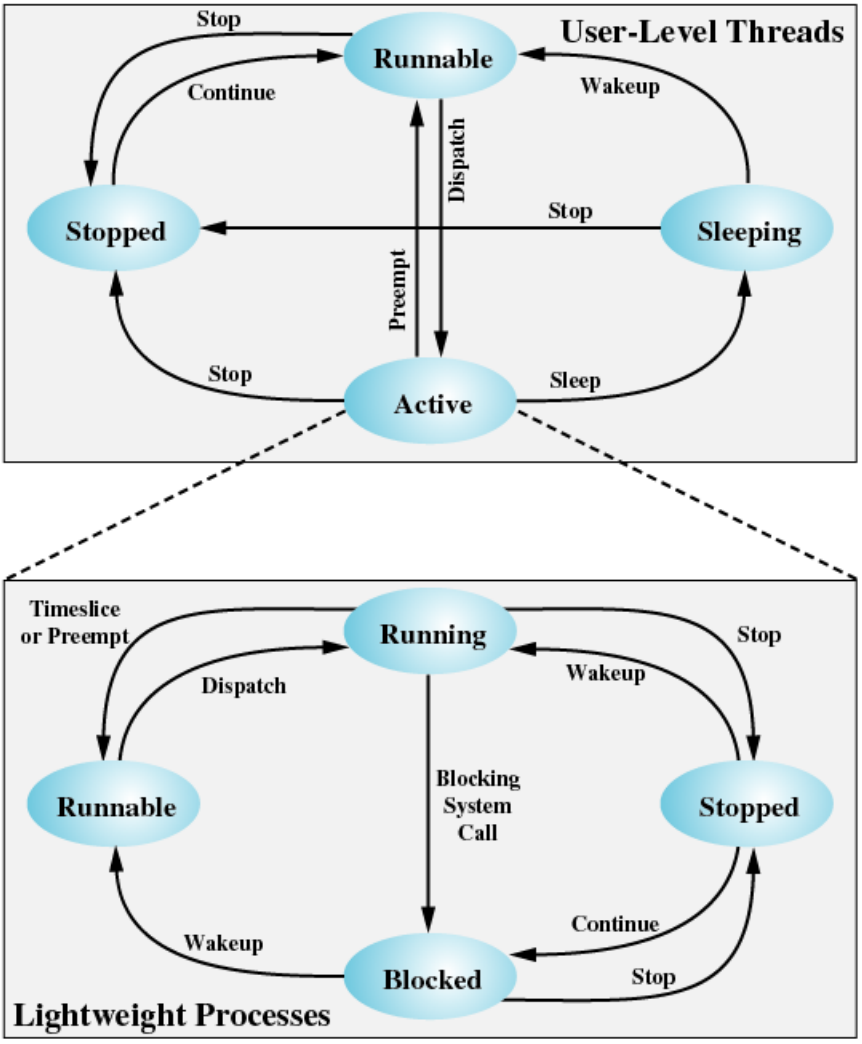


Figure 4.17 Solaris User-Level Thread and LWP States

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

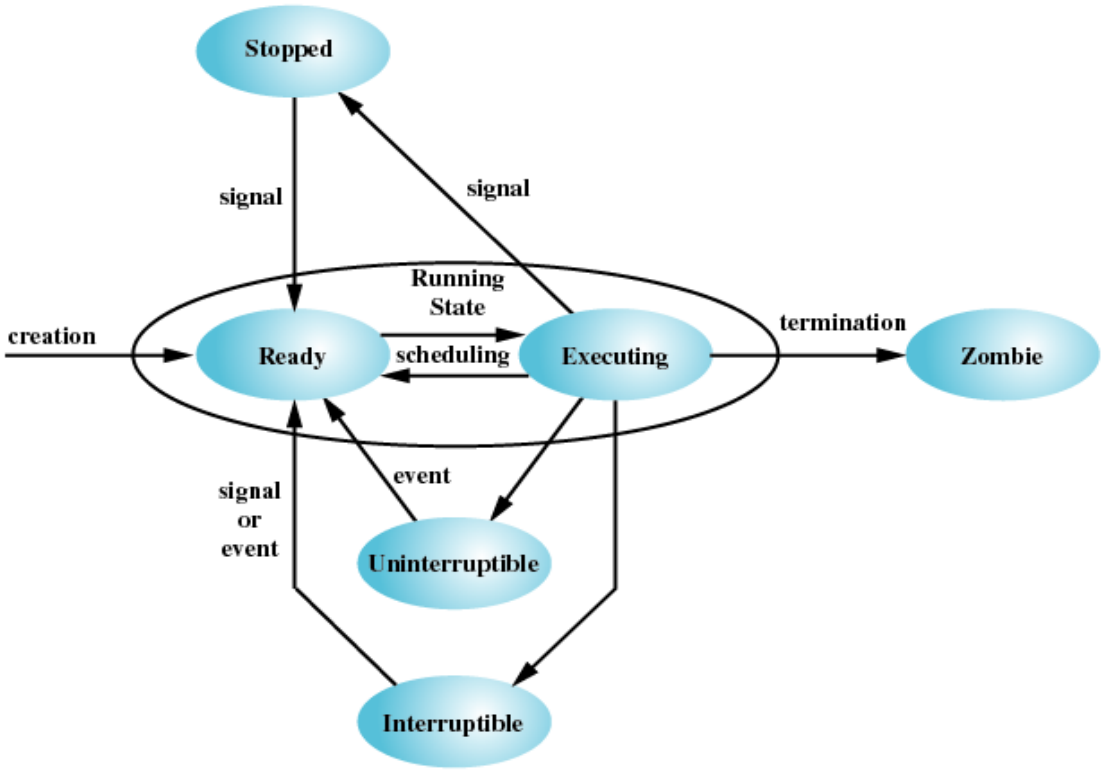


Figure 4.18 Linux Process/Thread Model