

# Threads

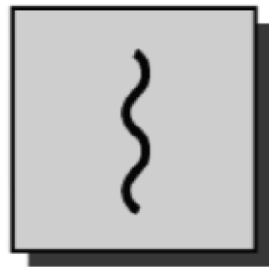
Gaurav Somani  
LNMIIT, Jaipur

References: Operating Systems- Internals and Design Principals, Fifth Edition,  
William Stallings. <http://williamstallings.com/OS/OS5e.html>

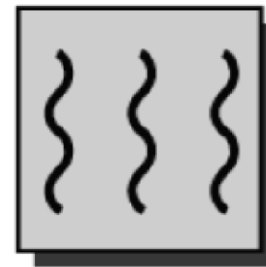
# Multithreading

Operating system supports multiple threads of execution within a single process

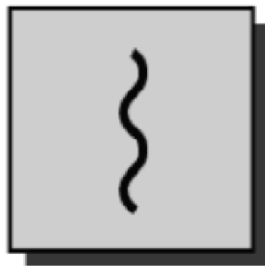
- 1) MS-DOS supports a single thread
- 2) JRE (Java Run Time) – A single process and multiple threads.
- 3) UNIX supports multiple user processes but only supports one thread per process
- 4) Windows, Solaris, Linux, Mach, and OS/2 support multiple threads



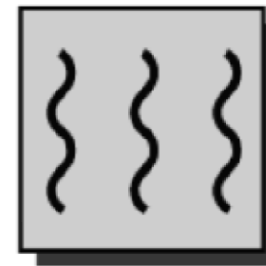
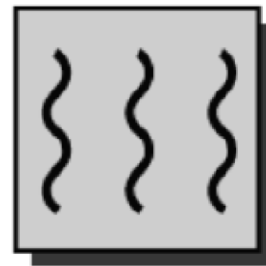
one process  
one thread



one process  
multiple threads



multiple processes  
one thread per process

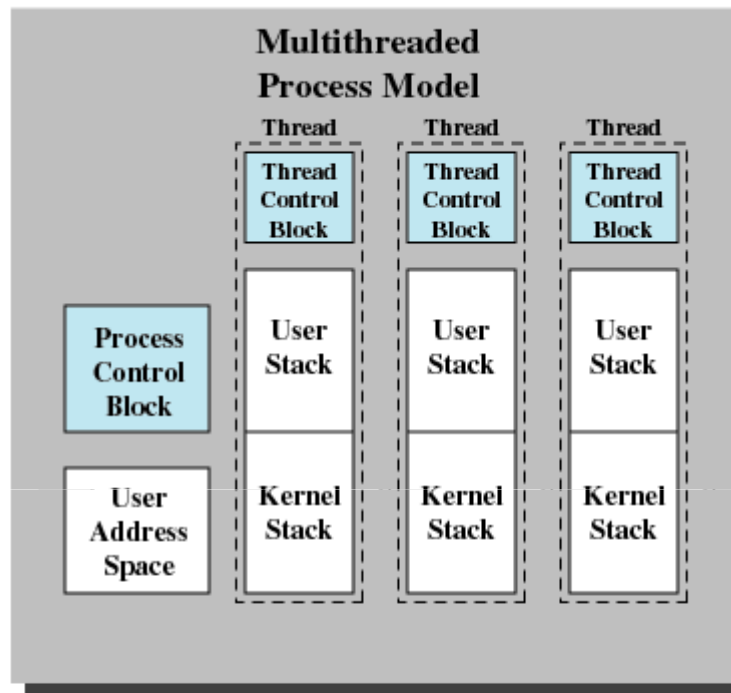
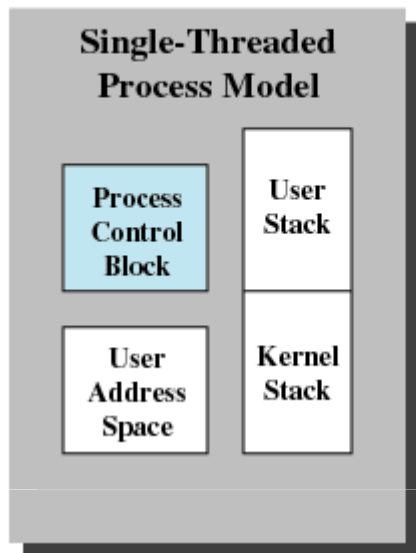


multiple processes  
multiple threads per process

} = instruction trace

# Thread

- An execution state (running, ready, etc.)
- Saved thread context when not running
- Has an execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process
  - all threads of a process share this



# Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

# Uses of Threads in a Single-User Multiprocessing System

- Foreground to background work - Spreadsheet
- Asynchronous processing – Word Processor
- Speed of execution -
- Modular program structure

# Threads

## Process level decisions

- **Suspension** of 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



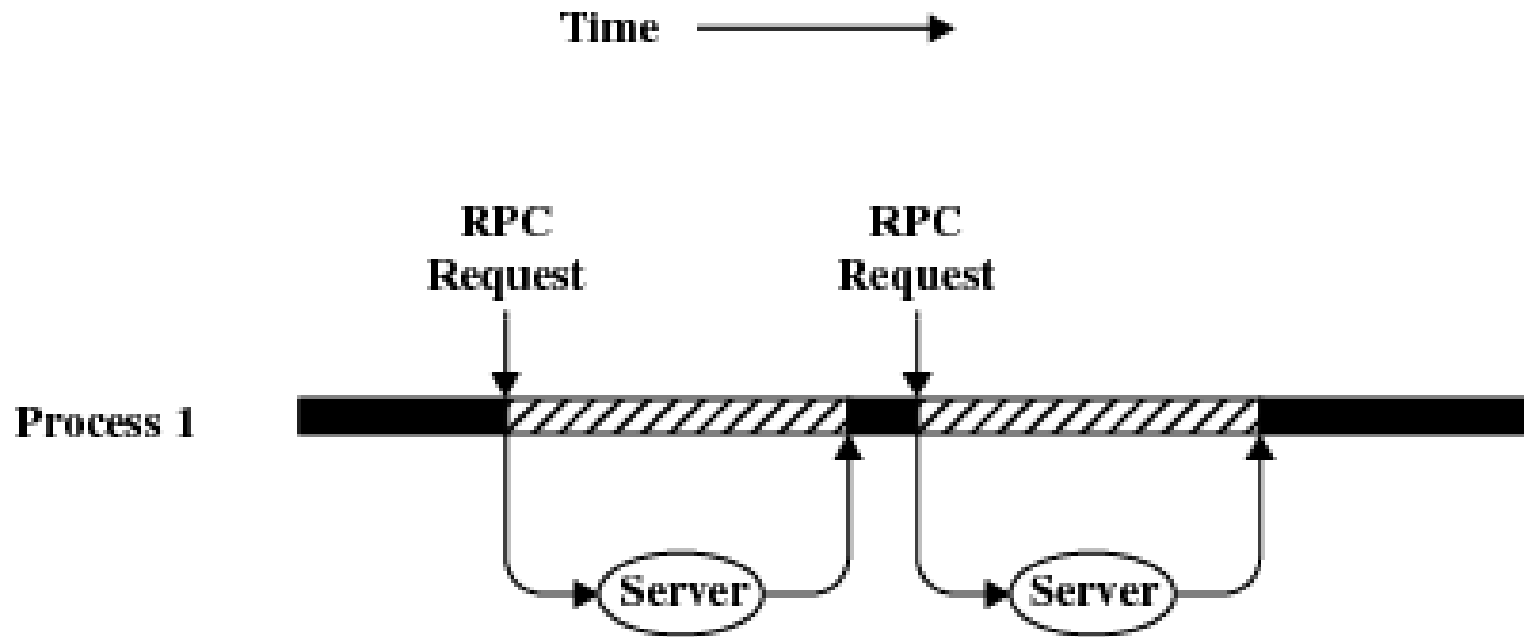
# Thread Operations

1. Spawn
  1. Spawn another thread
2. Block
3. Unblock
4. Finish
  1. Deallocate register context and stacks

Do we block the whole process when a thread blocks?

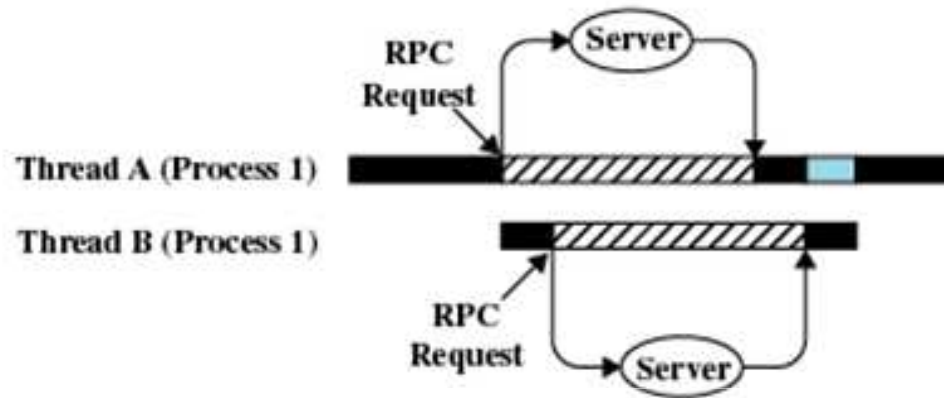
- Check an example where having no process block is beneficial.

# Remote Procedure Call Using Single Thread






(a) RPC Using Single Thread

# Remote Procedure Call Using Threads

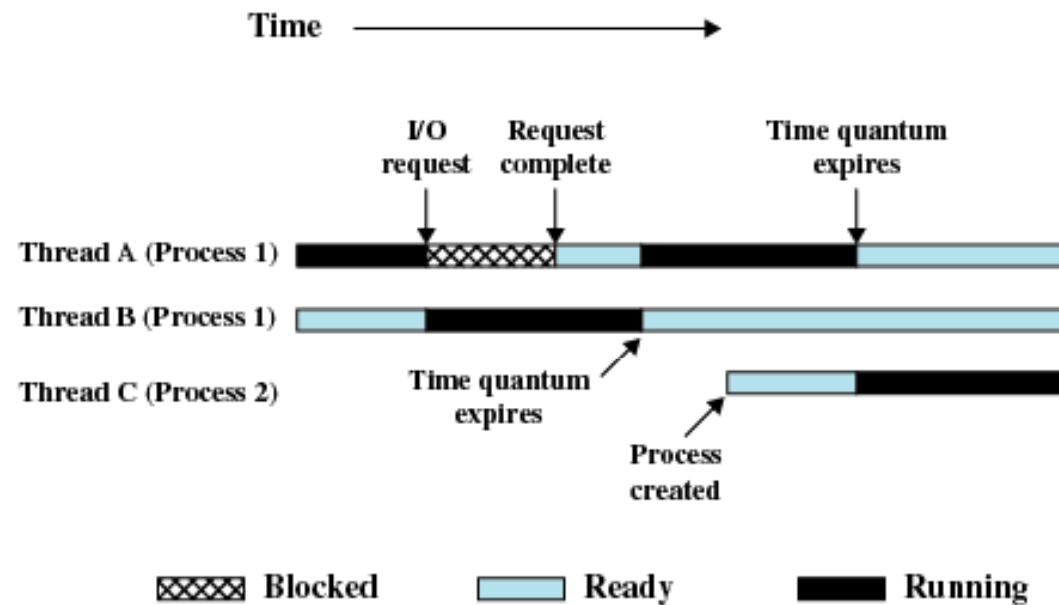


(b) RPC Using One Thread per Server (on a uniprocessor)

-  Blocked, waiting for response to RPC
-  Blocked, waiting for processor, which is in use by Thread B
-  Running

## Remote Procedure Call (RPC) Using Threads

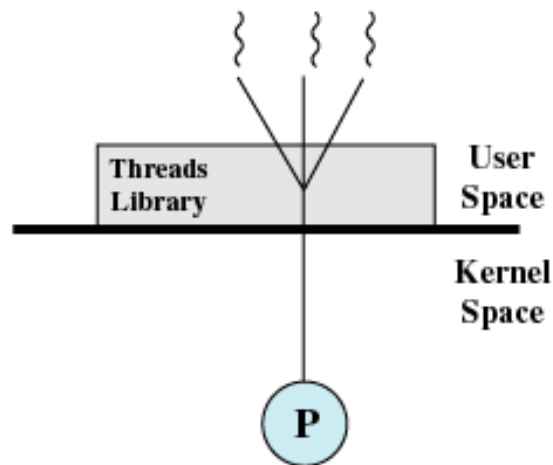
# Multithreading

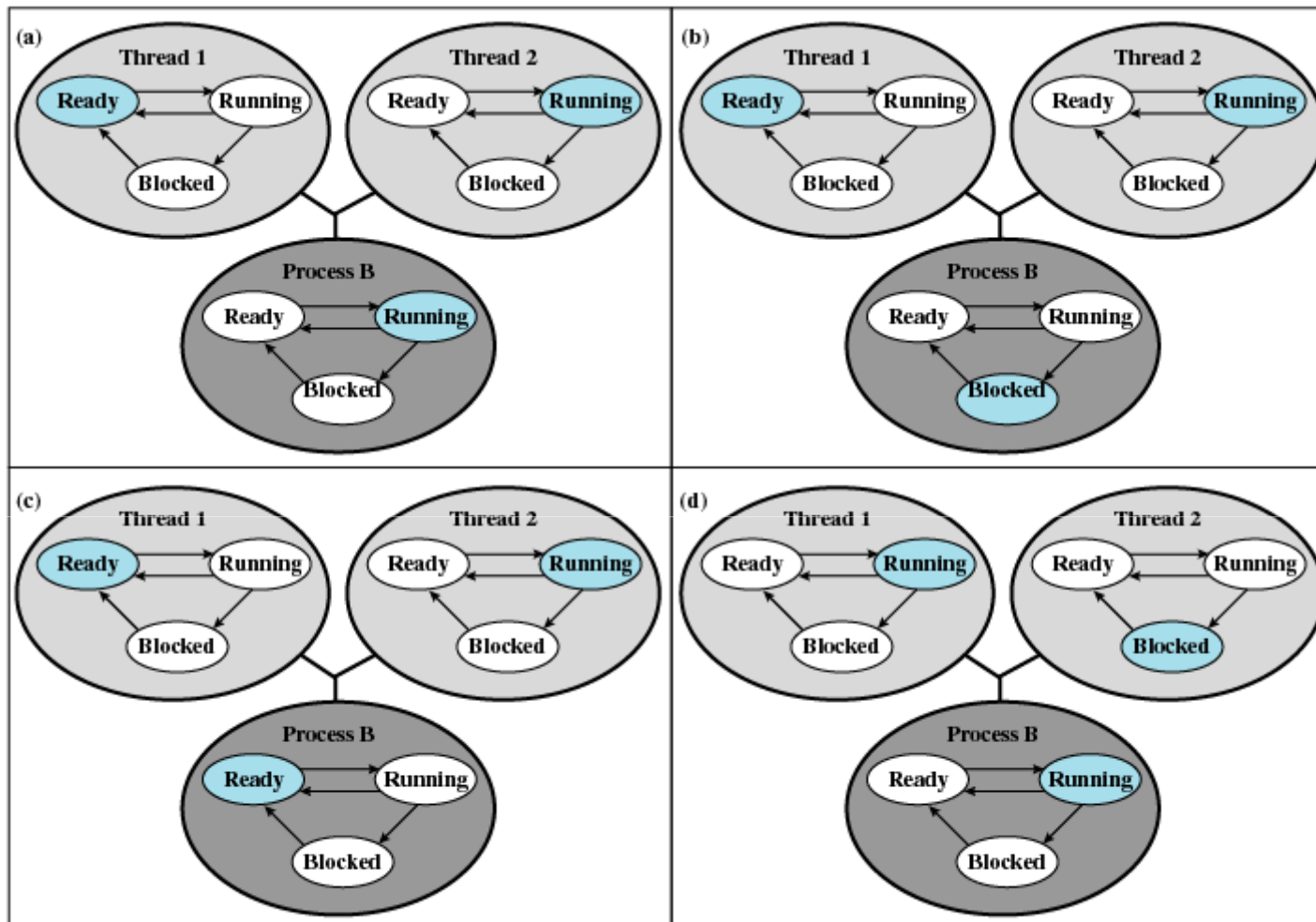


**Multithreading Example on a Uniprocessor**

# User-Level Threads

- All thread management is done by the application
- The kernel is not aware of the existence of thread
- Thread library is used to create/terminate, data passing and scheduling threads.





Colored state  
is current state

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

# User-Level Threads

1. Thread 2 makes a system call – block B
2. Clock interrupt – slice exhausted for B (Ready)
3. Thread 2 blocks – waiting for thread 1's event to happen.  
(Running)

# Advantages of ULTs vs. KLTs

1. Thread switching does not require kernel mode privileges. This saves the overhead of two mode switches (user to kernel; kernel back to user).
2. Scheduling can be application specific.
3. ULTs can run on any OS. No changes are required to the underlying kernel to support ULTs.



# Disadvantages of ULTs vs. KLTs

1. 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 also all of the threads within the process are blocked.
2. 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. In effect, we have application-level multiprogramming within a single process.

Solutions?

# Solutions?

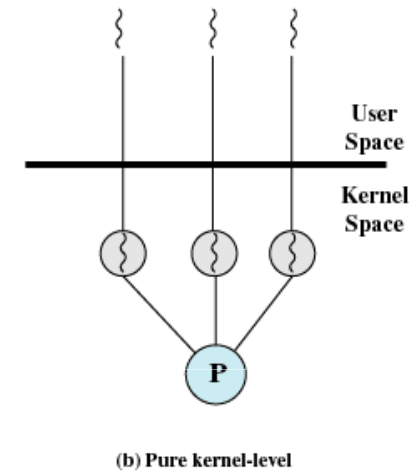
1. **Jacketing** The purpose of jacketing is to convert a blocking system call into a non-blocking system call.
  1. For example, instead of directly calling a system I/O routine, a thread calls an application-level I/O jacket routine. Within this jacket routine is code that checks to determine if the I/O device is busy.
  2. If it is, the thread enters the Blocked state and passes control (through the threads library) to another thread. When this thread later is given control again, the jacket routine checks the I/O device again.
2. Do multiprocessing – No threading advantage.

# Kernel-Level Threads

- No thread management in application area. (Windows)
- Kernel maintains context information for the process and the threads
- Scheduling is done on a thread basis
- Overcomes drawbacks of ULTs
  1. Scheduling multiple threads from a process to multiple processors.
  2. If one thread is blocking than schedule another thread from the same process.

# Kernel-Level Threads

Main disadvantage is context switch cost.



# VAX running UNIX-Like Operating System

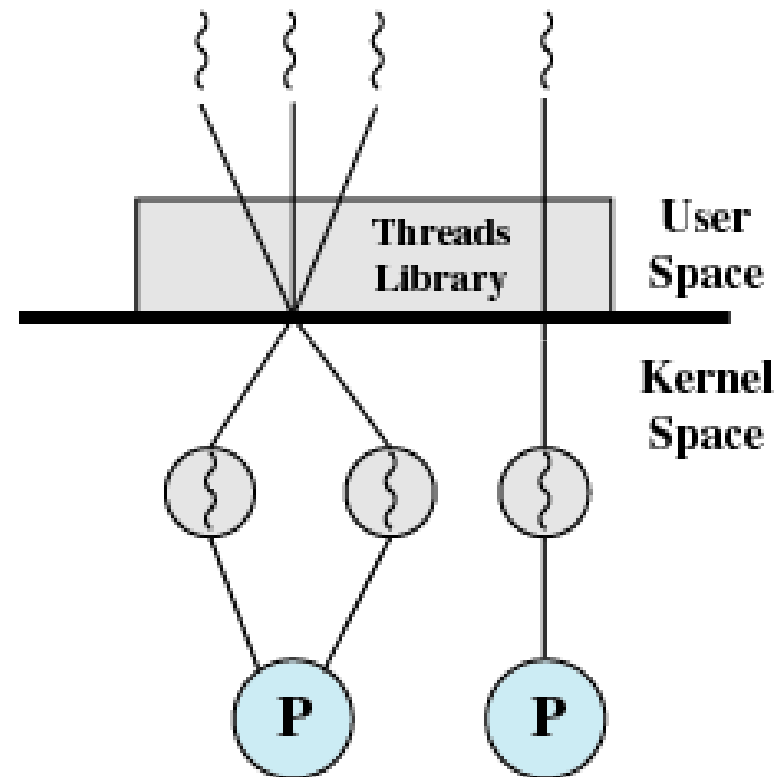
**Thread and Process Operation Latencies ( $\mu$ s)**

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

# Combined Approaches

- Example is Solaris
- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads within application
  - Multiprocessor support.
  - No process blocking on thread blocking

# Combined Approaches



(c) Combined

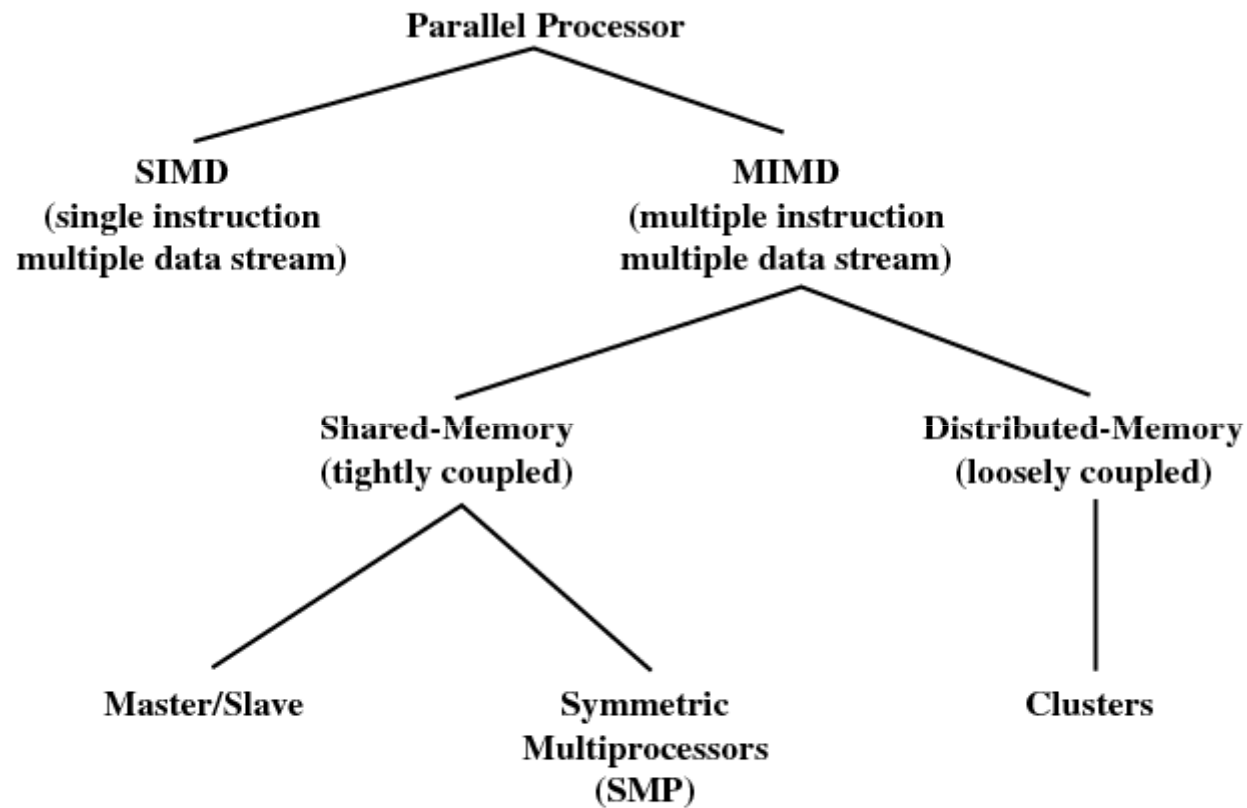
# Relationship Between Threads and Processes

**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



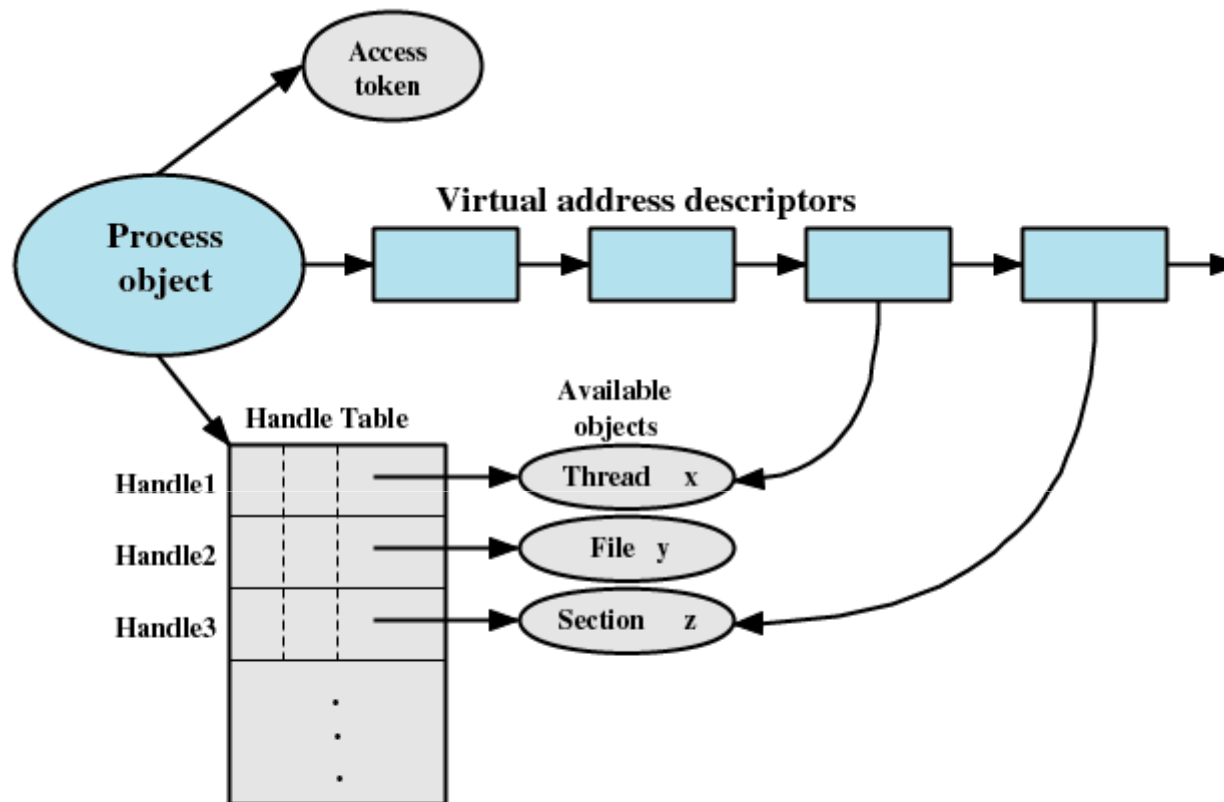
# Parallel Processor Architecture



Parallel Processor Architectures

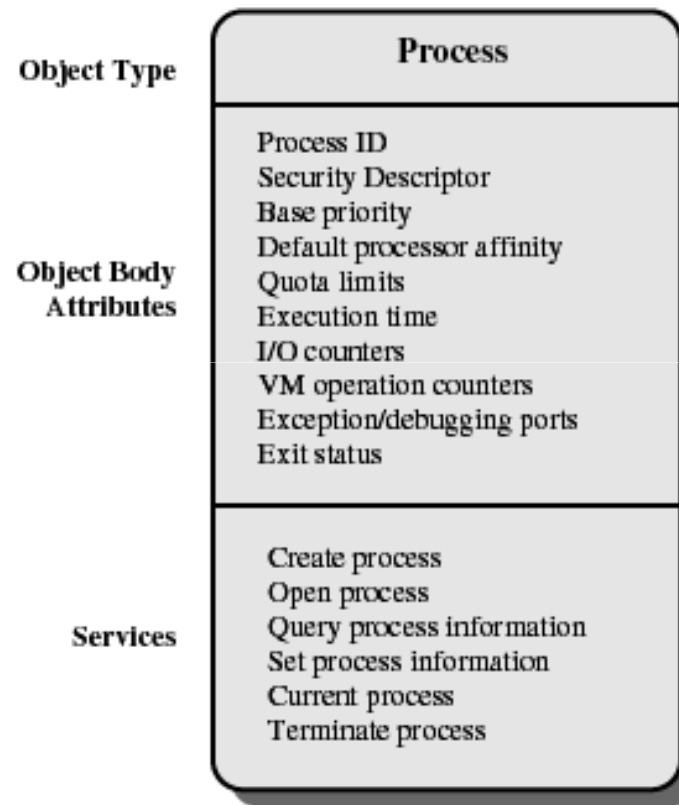
# Windows Processes

- Implemented as objects
- An executable process may contain one or more threads
- Both processes and thread objects have built-in synchronization capabilities
- Multiple threads from a single process can run on multiple processors concurrently.



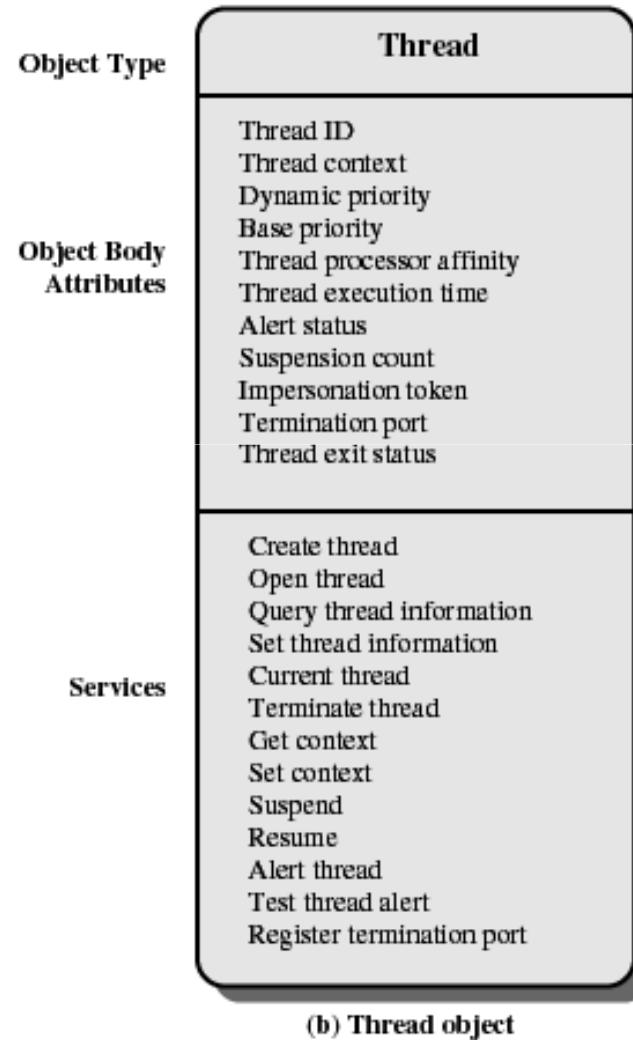
**A Windows Process and Its Resources**

# Windows Process Object



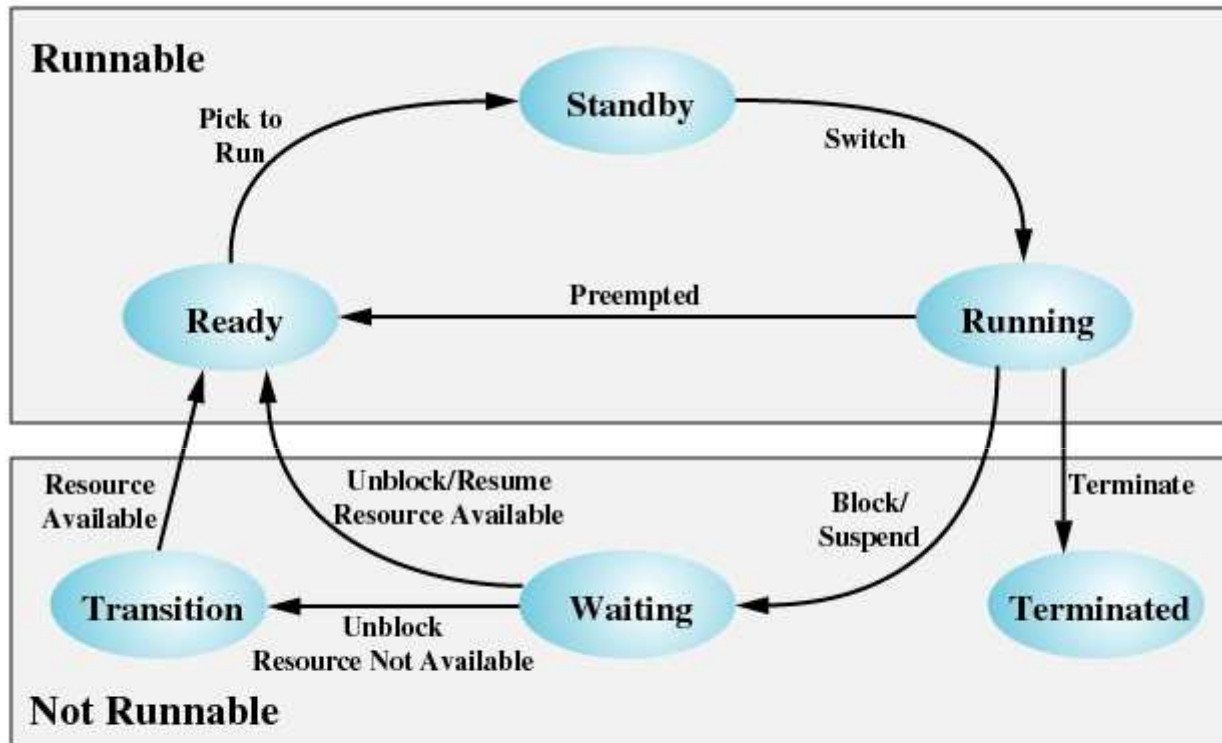
(a) Process object

# Windows Thread Object



# Windows 2000 Thread States

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

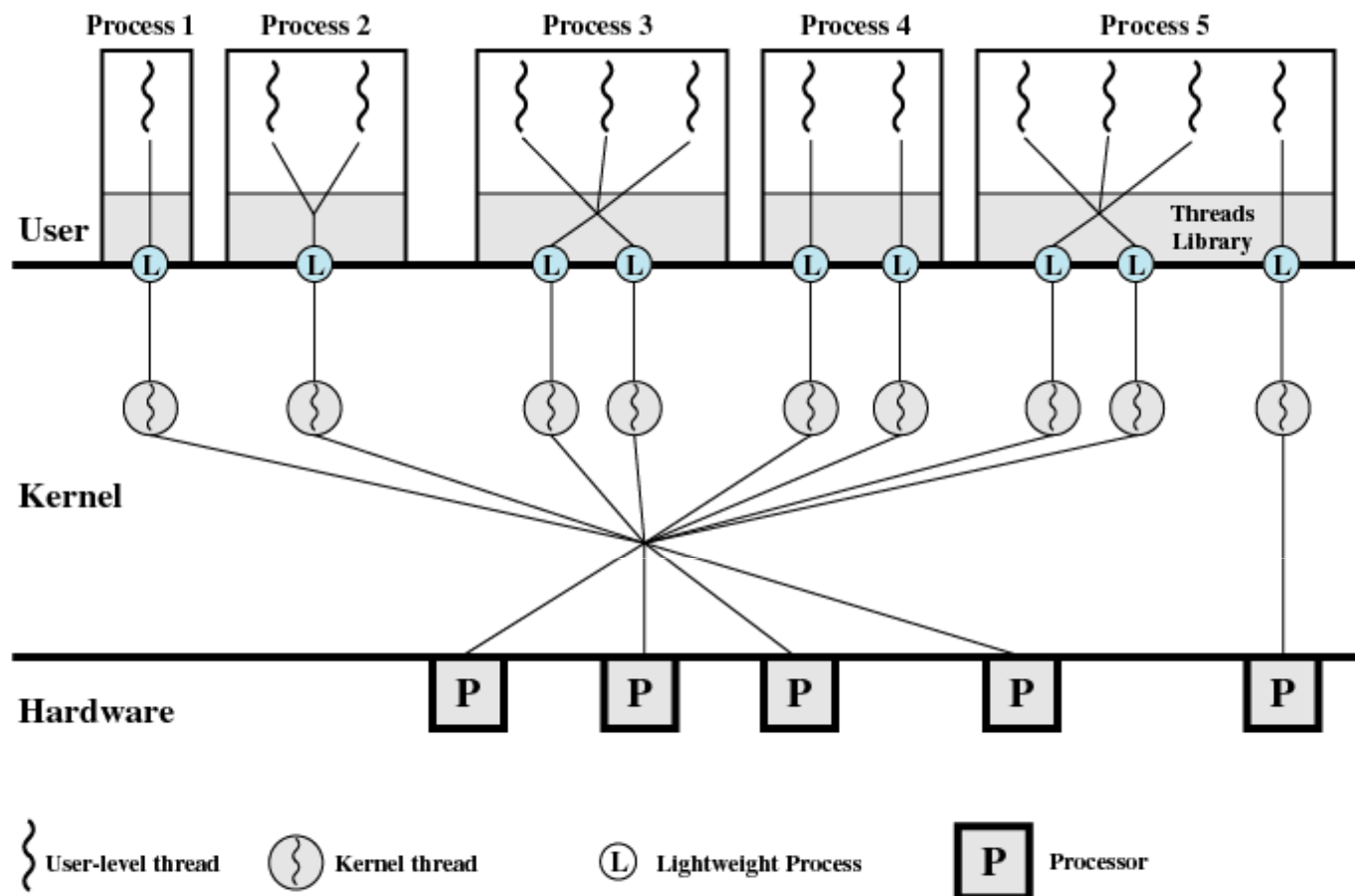


**Windows Thread States**

# Solaris

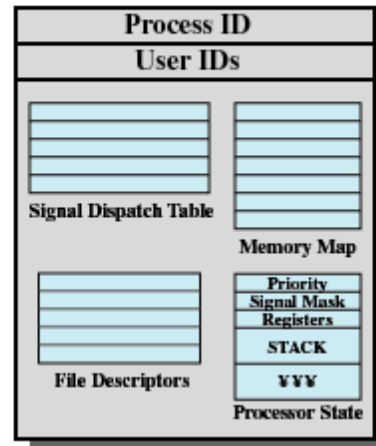
- Process includes the user's address space, stack, and process control block
- User-level threads
- Lightweight processes (LWP)
- Kernel threads



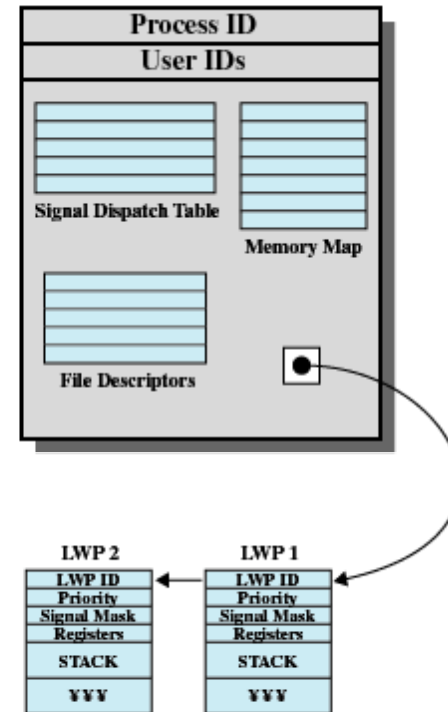


**Solaris Multithreaded Architecture Example**

## UNIX Process Structure



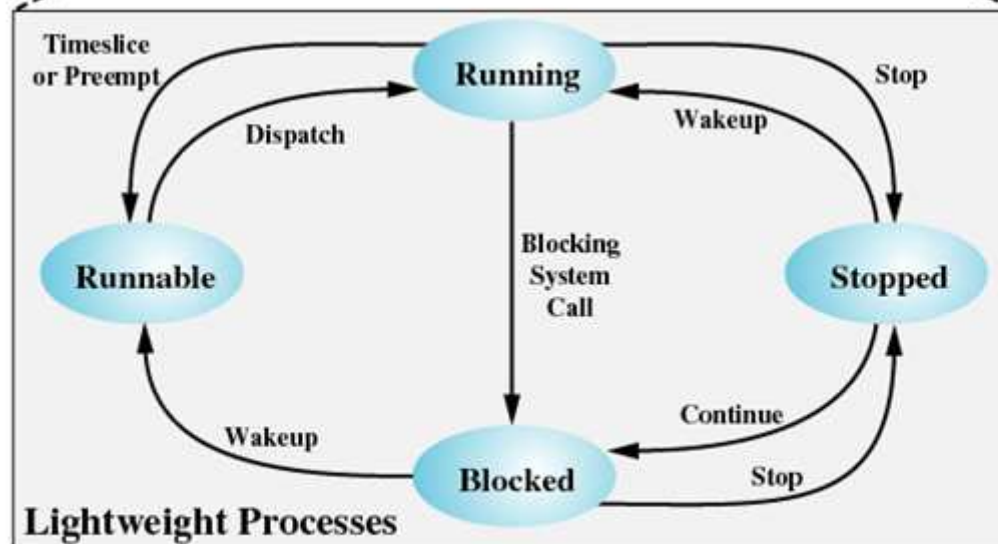
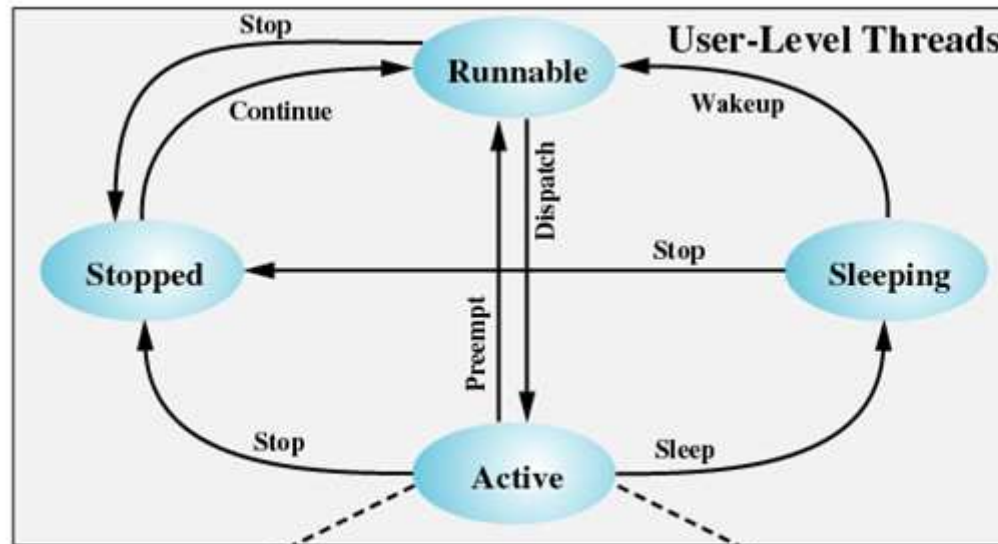
## Solaris Process Structure



## Process Structure in Traditional UNIX and Solaris

# 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

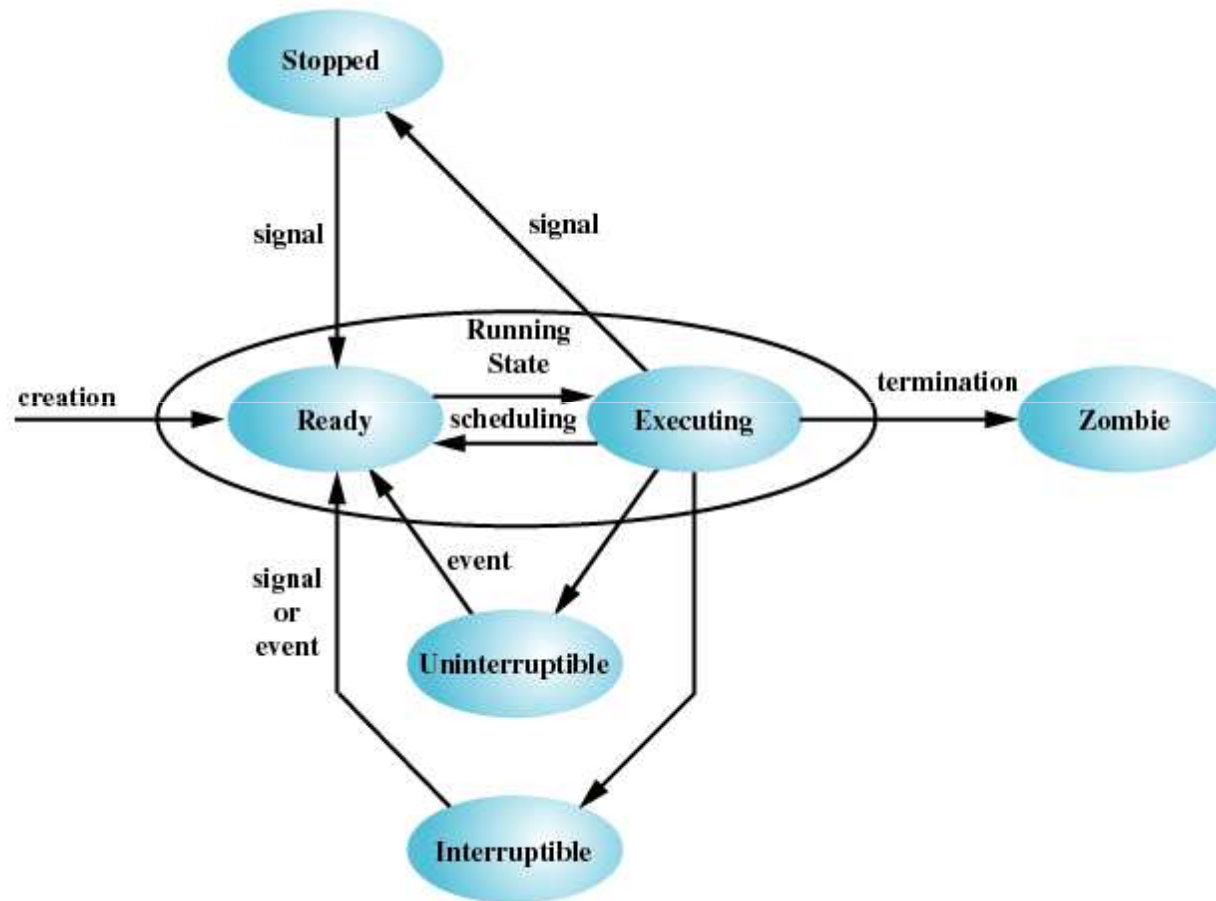


Solaris User-Level Thread and LWP States

# 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



Linux Process/Thread Model

# Linux Kernel Level Threads

Flag	Description
CLONE_VM	Share data and stack
CLONE_FS	Share file system info
CLONE_FILES	Share open files
CLONE_SIGHAND	Share signals
CLONE_PID	Share PID parent

```
int clone(int (*fn)(), void **stack, int flags, int argc,... /* args */);
```

fn	The thread routine
stack	The thread's stack
flags	Flags as defined above
argc	number of remaining parameters
/* args */	the parameters expected by the thread.

Thanks