Threads and Processes

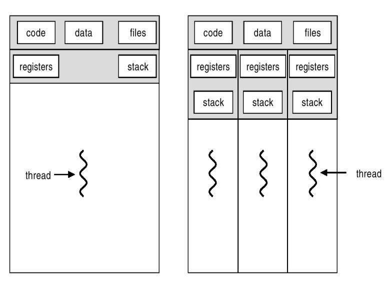
Chapter 4, Livro do William Stallings

Sistemas de Operação, 2004-2005

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 2000, Solaris, Linux, Mach, and OS/2 support multiple threads.

Single-thread vs Multi-thread



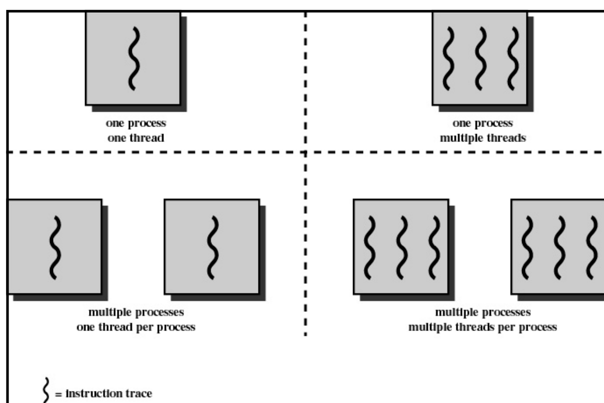


Figure 4.1 Threads and Processes [ANDE97]

Process

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

Thread

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

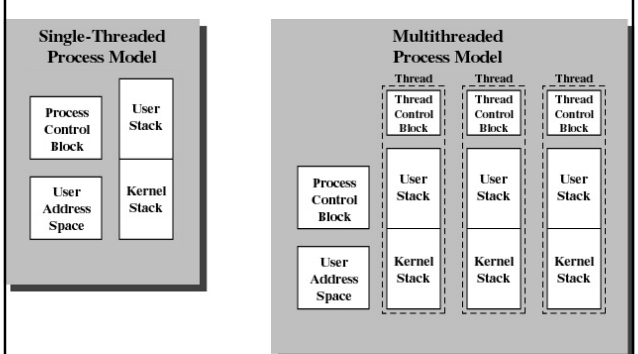


Figure 4.2 Single Threaded and Multithreaded Process Models

Benefits of Threads

- Takes less time to create a new thread than a process.
- Less time to terminate a thread than a process.
- 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.

Solaris Threads

- Unlike processes, threads run within the same address space and share their process' data.
- In such environments, the thread creation and destruction takes place considerably faster compared to a full-blown process' creation or destruction.
- Under Solaris, for example, launching a new thread is about 70 times faster than launching a new process.

copy-on-write fork

- Linux supports copy-on-write fork.
- It leaves the mapped memory shared between a parent process and its child as long as the child doesn't alter the shared addressable region.
- Only when the child writes to the shared address space does the kernel allocate new storage.
- Hence, launching a new process in Linux involves significantly lower overhead compared to Solaris and other OSs.

Uses of Threads

- Foreground to background work.
- Asynchronous processing.
- Speed execution.
- Responsiveness
- Resource Sharing
- Utilization of Multi-processor Architectures

Threads

- 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.

Threading Issues

- Semantics of fork() and exec() system calls.
- Thread cancellation.
- Signal handling
- Thread pools
- Thread specific data

Thread States

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

Remote Procedure Call Using Threads

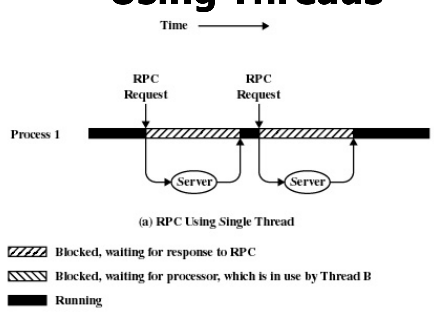


Figure 4.3 Remote Procedure Call (RPC) Using Threads

Remote Procedure Call Using Threads

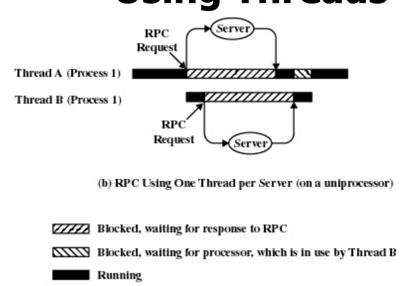


Figure 4.3 Remote Procedure Call (RPC) Using Threads

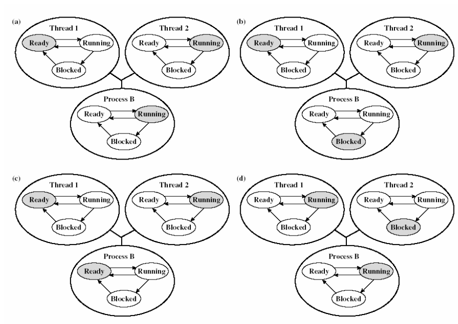
Synchronizing Threads

- Threads share the same address space and resources.
- Therefore, it is the responsibility of the programmer to assure the correctness in the concurrent access to data and resources

User-Level Threads (ULT)

- All thread management is done by the application.
- The kernel is not aware of the existence of threads.
- A context switch between two threads of the same process essentially jumps from one code location to another, plus setting a few CPU registers.

User-level Threads and Processes



Kernel-Level Threads (KLT)

- W2K, Linux(*), and OS/2 are examples of this approach.
- Kernel maintains context information for the process and the threads.
- Scheduling is done on a thread basis.

(*) LinuxTreads: provides **kernel-level** threads

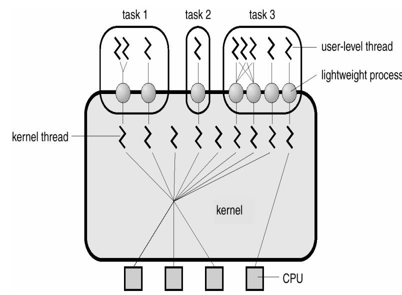
There are other libraries with **user-level** threads

(http://www.ibiblio.org/pub/Linux/docs/faqs/Threads-FAQ/html/ThreadLibs.html)

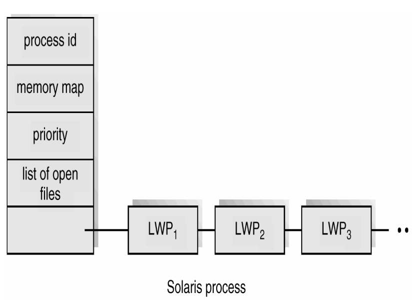
Combined Approaches

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

Solaris 2 Threads



Solaris Process



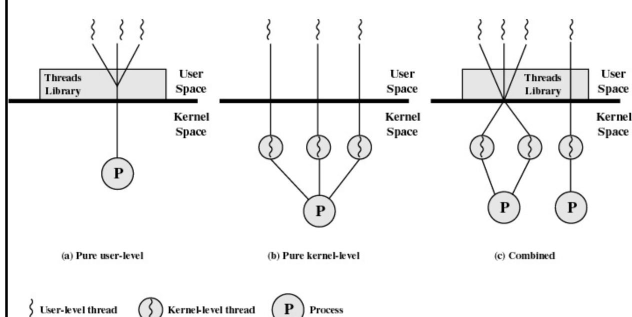


Figure 4.6 User-Level and Kernel-Level Threads

Advantages of ULT

- Thread switching does not require kernel mode privileges.
- Scheduling can be application specific without disturbing the underlying OS scheduler.
- ULTs can run on any operating system (the threads library is a set of application-level utilities shared by all applications).

Disadvantages of ULT

- Many system calls are blocking. When a ULT executes a blocking system call all the threads in that process will be blocked.
- Multithreaded applications that make use of ULT cannot take advantage of multiprocessing.

Advantages of KLT

- The two previous problems (blocking and support for multiprocessing) are solved with KLT.
- Kernel routines themselves can also be multithreaded.

Disadvantages of KLT

- The transfer of control between two threads of the same process require a mode switch to the kernel.

Thread Operation Latencies

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

VAX machine running Unix (values in μs) proc call=7 μs; kernel trap=17 μs

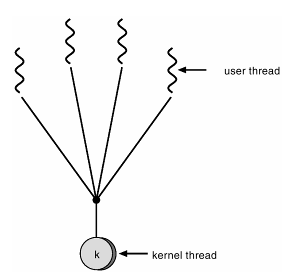
Combined Approach

- Multiple threads within the same application can run in parallel on multiple processors.
- A blocking system call does not block the entire process.
- The approach combines the advantages of ULT and KLT.

Multithreading Models

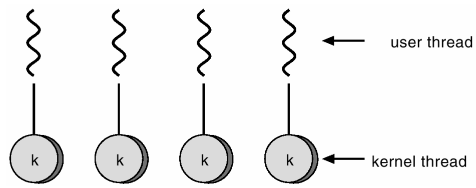
- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One Model



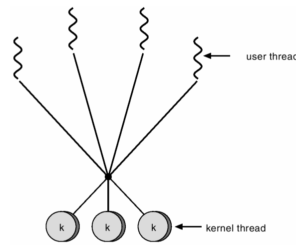
- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.

One-to-one Model



- Each user-level thread maps to kernel thread.
- Examples
  - Windows 95/98/NT/2000
  - OS/2

Many-to-Many Model



- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- Solaris 2

PThreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems.

Linux Threads

- Linux refers to them as *tasks* rather than *threads*.
- Thread creation is done through clone() system call.
- Clone() allows a child task to share the address space of the parent task (process)

Linux Process

- State
- Scheduling information
- Identifiers
- Interprocess communication
- Links
- Times and timers
- File system
- Virtual memory
- Processor-specific context

Linux States of a Process

- Running
- Interruptable
- Uninterruptable
- Stopped
- Zombie

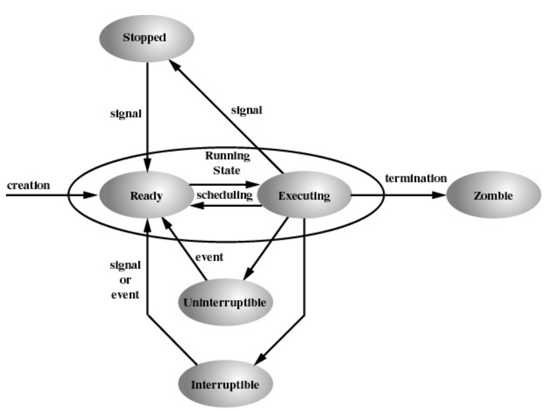
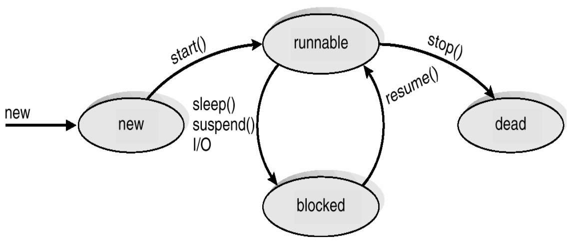


Figure 4.18 Linux Process/Thread Model

Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java threads are managed by the JVM.
- One process: multiple threads.

Java Threads States



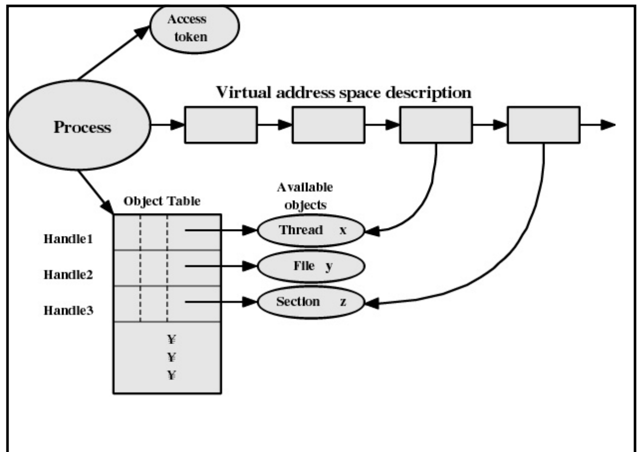
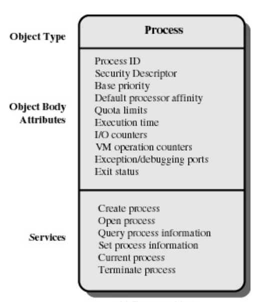


Figure 4.12 Windows 2000 Process and Its Resources

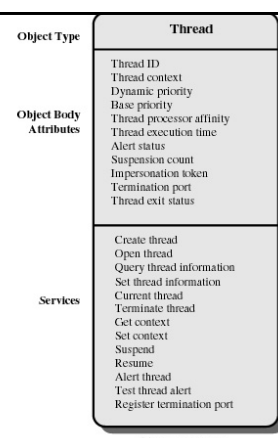
Windows 2000 Process Object



(a) Process object

Windows 2000 Thread Object

- Implements the one-to-one mapping.
- Each thread contains
  - a thread id
  - register set
  - separate user and kernel stacks
  - private data storage area



(b) Thread object

Windows 2000 Thread States

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

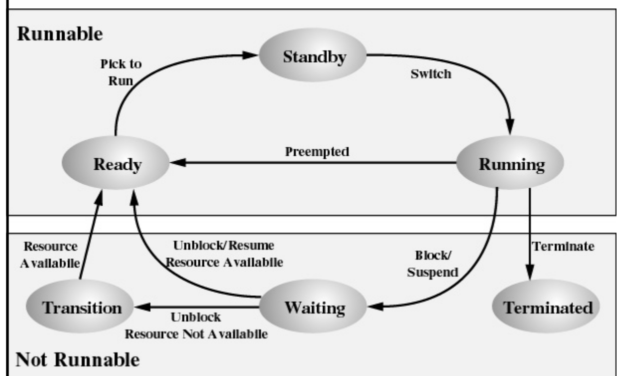


Figure 4.14 Windows 2000 Thread States

Solaris

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

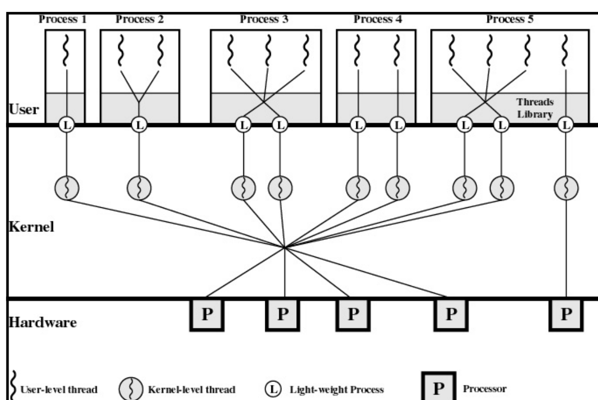


Figure 4.15 Solaris Multithreaded Architecture Example

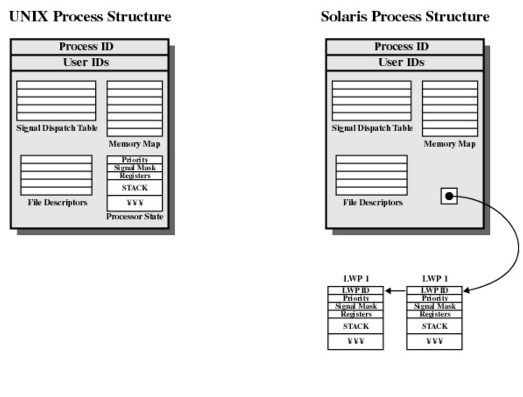


Figure 4.16 Process Structure in Traditional UNIX and Solaris [LEWI96]

Solaris Thread Execution

- Synchronization
- Suspension
- Preemption
- Yielding

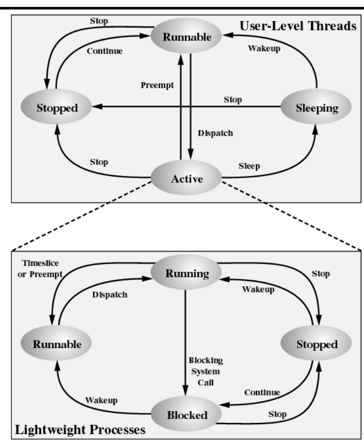


Figure 4.17 Solaris User-Level Thread and LWP States

Microkernels

- Small operating system core
- Contains only essential 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

Benefits of a Microkernel Organization

- Uniform interface on request made by a process
  - 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

Benefits of Microkernel Organization

- Distributed system support
  - Message are sent without knowing what the target machine is
- 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
- Inter-process communication
- I/O and interrupt management