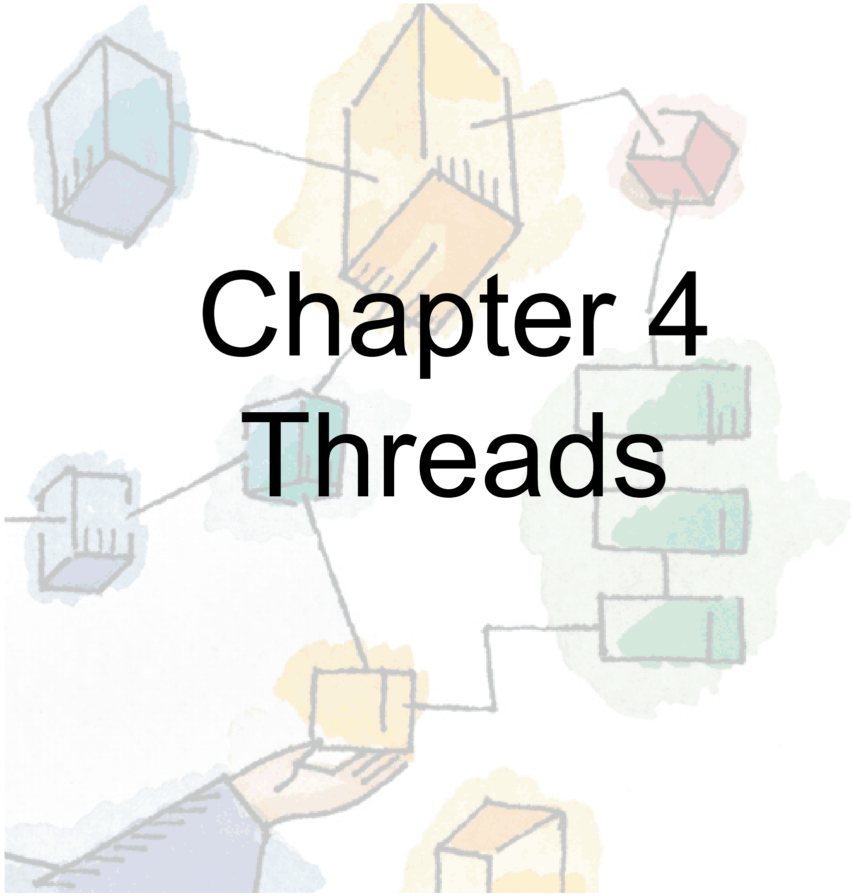
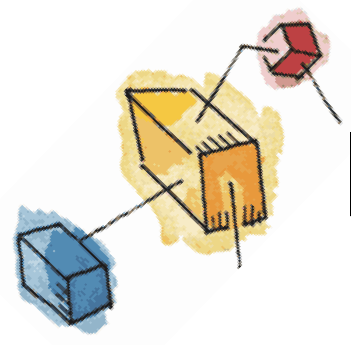
Operating Systems:
Internals and Design Principles
William Stallings

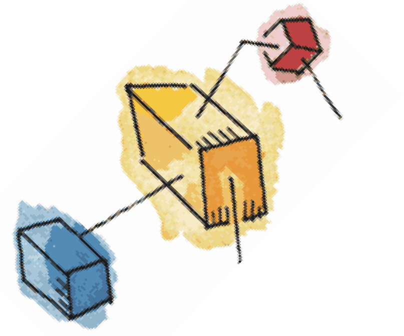


Chapter 4 Threads



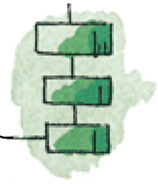
Processes and Threads

- Processes have two characteristics:
  - **Resource ownership** - process includes a virtual address space to hold the process image
    - the OS performs a protection function to prevent unwanted interference between processes with respect to resources
  - **Scheduling/execution** - follows an execution path that may be interleaved with other processes
    - a process has an execution state (Running, Ready, etc.) and a dispatching priority and is scheduled and dispatched by the OS
- These two characteristics are treated independently by modern operating systems:
  - 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***



Process

- The unit of resource allocation and a unit of protection
- A process is associated with
  - A virtual address space which holds the process image
  - Protected access to
    - Processors,
    - Other processes,
    - Files,
    - I/O resources



Multiple Threads in Process

- Each thread has
  - Access to the memory and resources of its process (all threads of a process share this)

  - An execution state (running, ready, etc.)
  - Saved thread context when not running
  - An execution stack
  - Some per-thread static storage for local variables

Threads vs. processes

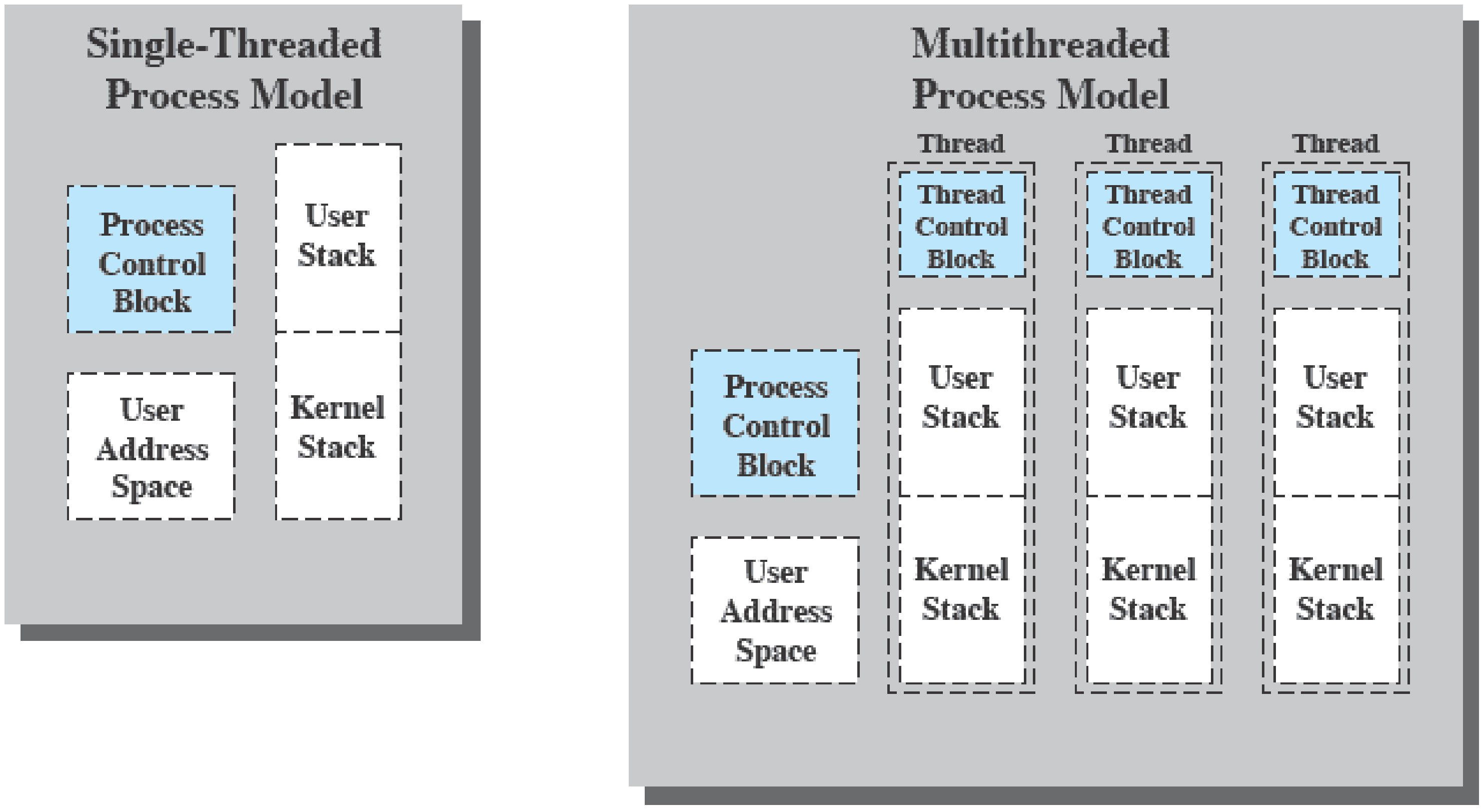
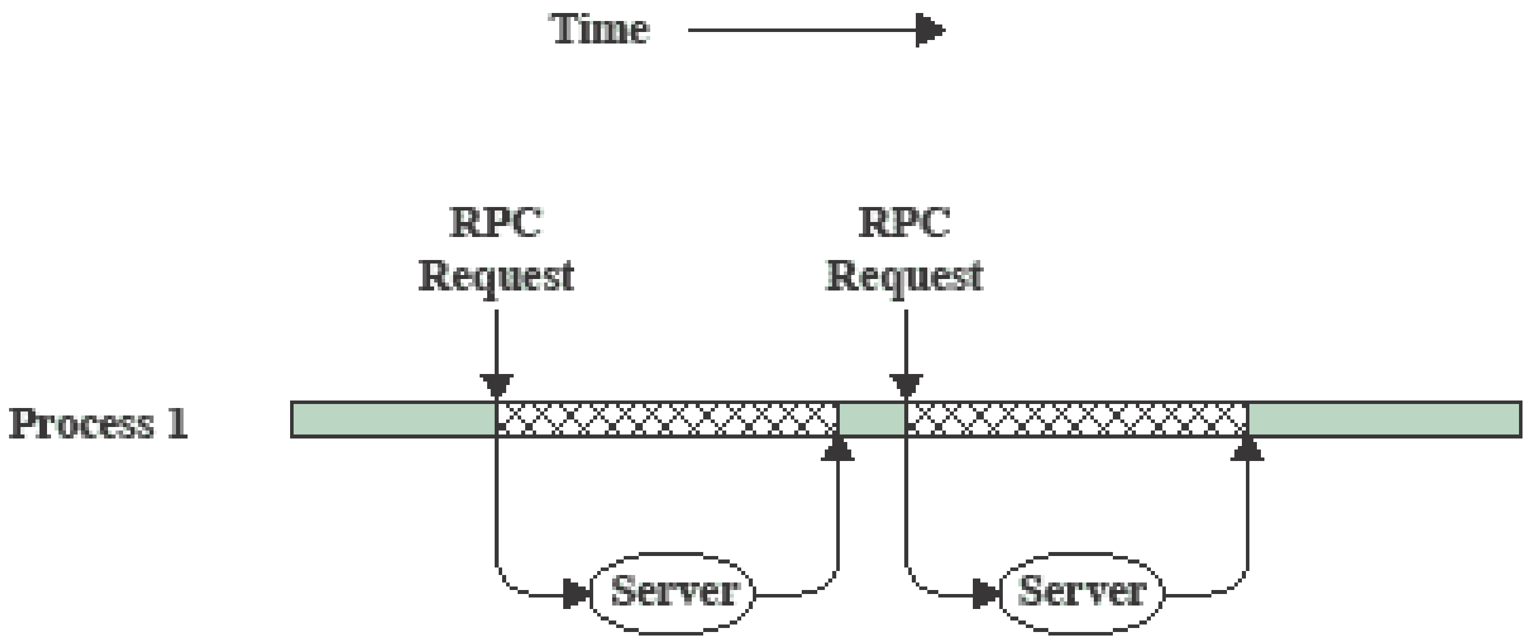


Figure 4.2 Single Threaded and Multithreaded Process Models

Example: Remote Procedure Call

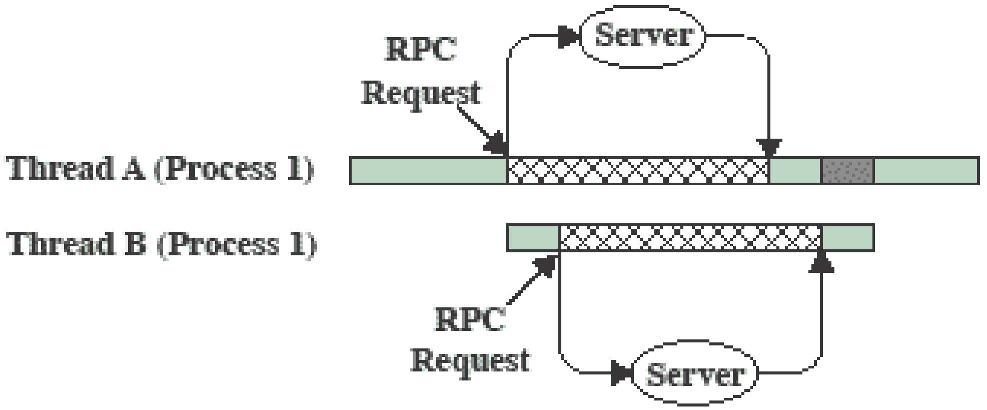
- Consider:
  - A program that performs two remote procedure calls (RPCs)
  - to two different hosts
  - to obtain a combined result.

RPC Using Single Thread



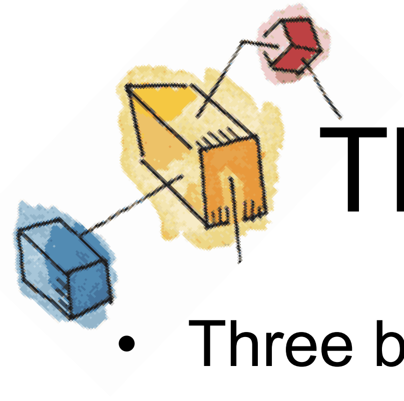
(a) RPC Using Single Thread

RPC Using One Thread per Server



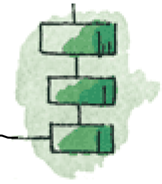
(b) RPC Using One Thread per Server (on a uniprocessor)

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



Thread Execution States

- Three basic states: Running, Ready, and Blocked
- Operations associated with a change in thread state
  - Spawn (another thread)
    - Allocate register context and stacks
  - Block
    - moved to an event queue waiting for the event
    - Issue: Will blocking a thread block other, or *all,* threads within the same process?
  - Unblock
    - moved to the Ready queue for execution
  - Finish (thread)
    - De-allocate register context and stacks



Thread Synchronization

- It is necessary to synchronize the activities of the various threads so that they do not interfere with each other
  - all threads of a process share the same address space and other resources
  - any alteration of a resource by one thread affects the other threads in the same process
- In general, the techniques used for thread synchronization are the same as those for process synchronization

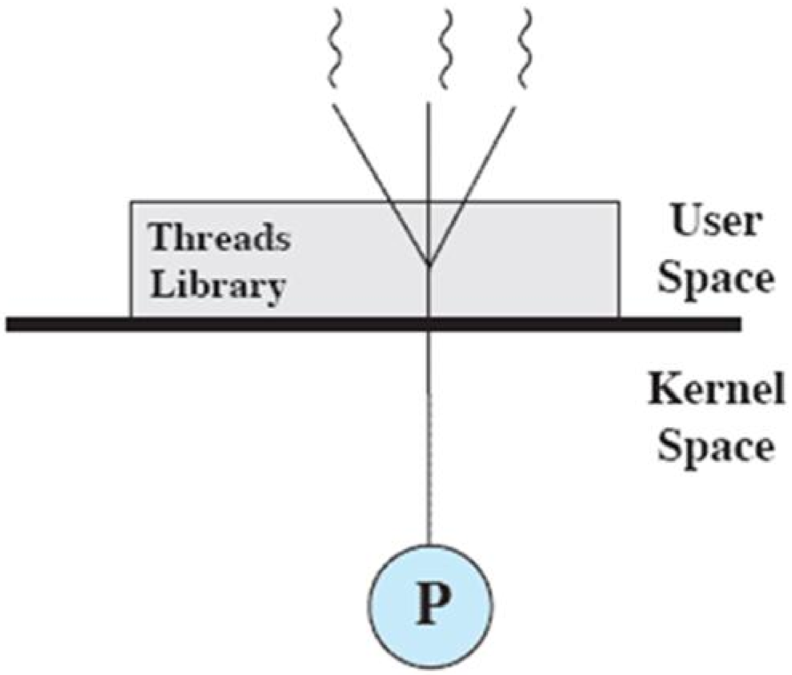
Types of Threads

- 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
  - Multithreading is managed by a runtime threads library
- The kernel is not aware of the existence of threads
  - Only schedule the process as a unit and assigns a single execution state to that process

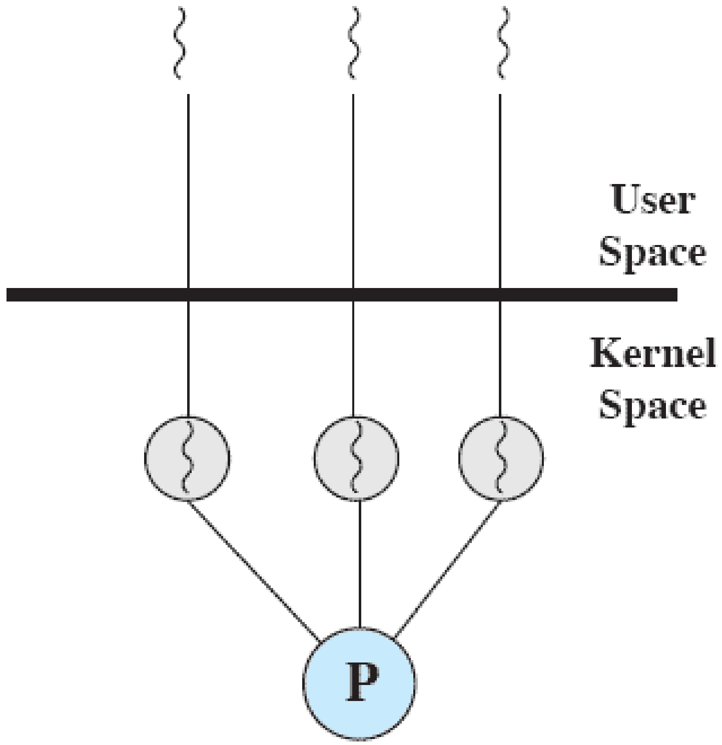


(a) Pure user-level

Disadvantages of ULT

- 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 all of the threads within the process are blocked
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing

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

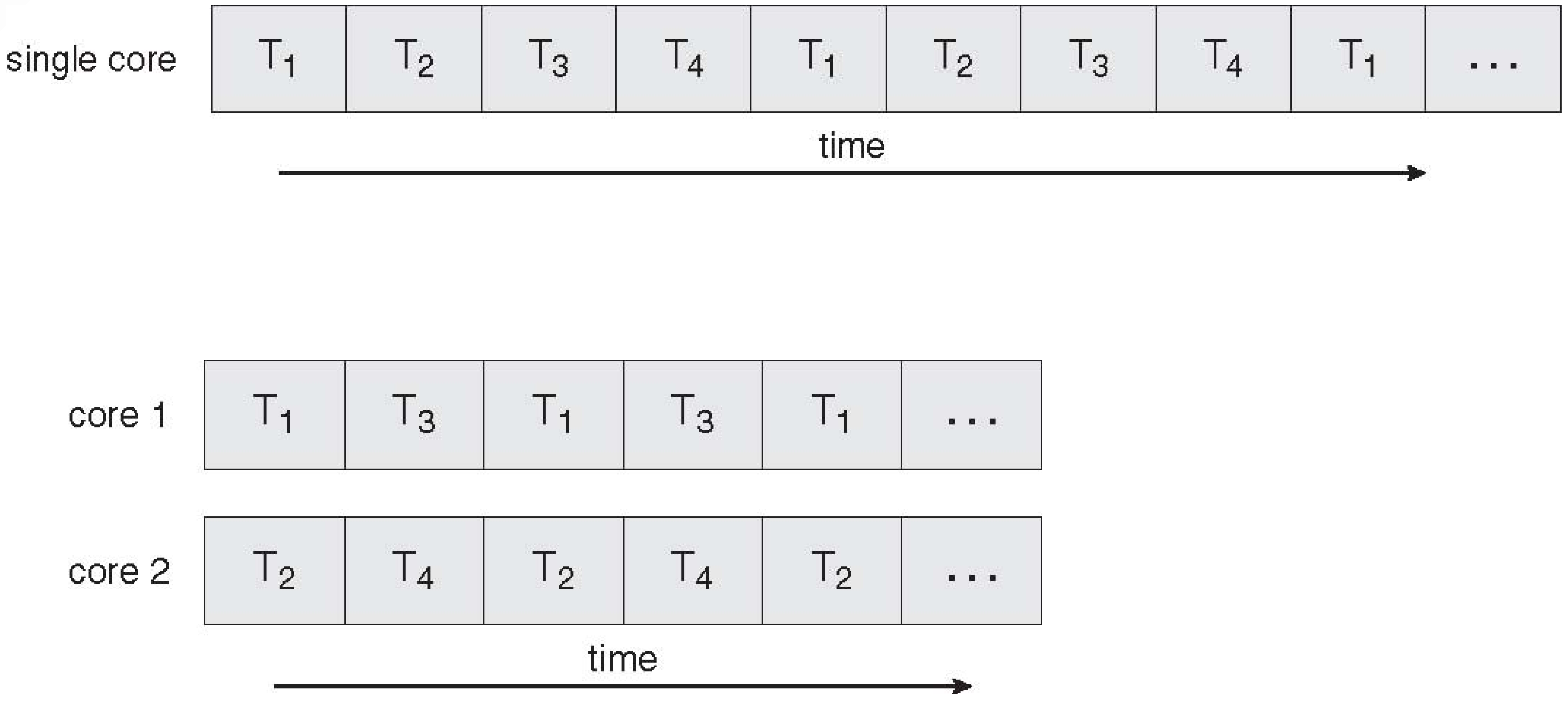
Kernel-Level Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures
- However, several actions will affect all of the threads in a process and the OS must manage at the process level
  - Suspending a process involves suspending all threads of the process
  - Termination of a process, terminates all threads within the process

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.

Multicore & Multithreading


single core
T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4 T_1 . . .
time
core 1
T_1 T_3 T_1 T_3 T_1 . . .
core 2
T_2 T_4 T_2 T_4 T_2 . . .
time

Multicore & Multithreading

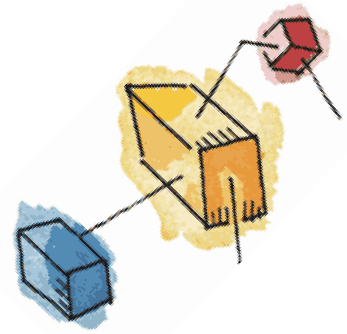
- Achieves concurrency without the overhead of using multiple processes
  - Threads within the same process can exchange information through their common address space and have access to the shared resources of the process

Multicore & Multithreading

- Multicore systems putting pressure on programmers, challenges include
  - Dividing activities
  - Balance
  - Data dependency
  - Testing and debugging

Linux Threads

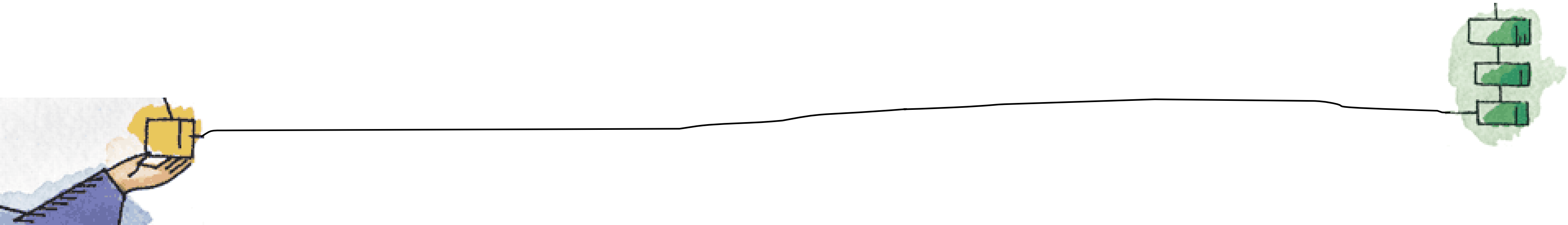
- Linux uses the same internal representation for processes and threads; a thread is simply a new process (or task) that happens to share the same address space as its parent
- A distinction is only made when a new thread is created by the `clone` system call
  - `fork` creates a new process with its own entirely new process context
  - `clone` creates a new process with its own identity, but that is allowed to share the data structures of its parent

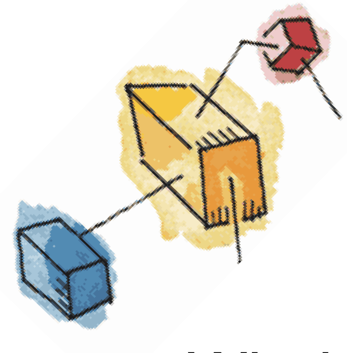


Linux Threads

- Using `clone` gives an application fine-grained control over exactly what is shared between two threads

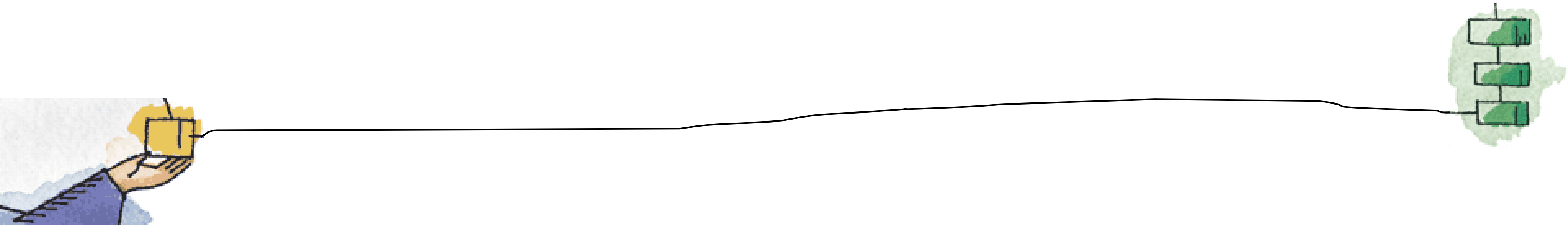
flag	meaning
`CLONE_FS`	File-system information is shared.
`CLONE_VM`	The same memory space is shared.
`CLONE_SIGHAND`	Signal handlers are shared.
`CLONE_FILES`	The set of open files is shared.





Windows Threads

- Windows makes use of two types of process-related objects:
- Processes
  – an entity corresponding to a user job or application that owns resources
- Threads
  – a dispatchable unit of work that executes sequentially and is interruptible



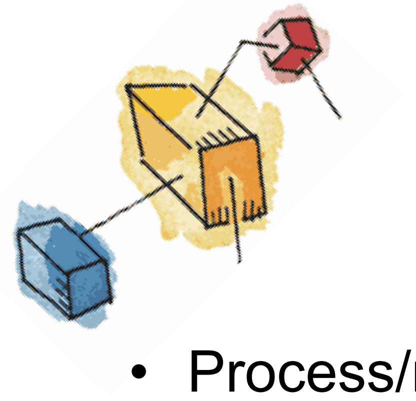
Process & Thread Objects

Object Type	Process
Object Body Attributes	Process ID Security Descriptor Base priority Default processor affinity Quota limits Execution time I/O counters VM operation counters Exception/debugging ports Exit status
Services	Create process Open process Query process information Set process information Current process Terminate process

(a) Process object

Object Type	Thread
Object Body Attributes	Thread ID Thread context Dynamic priority Base priority Thread processor affinity Thread execution time Alert status Suspension count Impersonation token Termination port Thread exit status
Services	Create thread Open thread Query thread information Set thread information Current thread Terminate thread Get context Set context Suspend Resume Alert thread Test thread alert Register termination port

(b) Thread object



Summary

- Process/related to resource ownership
- Thread/related to program execution
- User-level threads
  - created and managed by a threads library that runs in the user space of a process
  - a mode switch is not required to switch from one thread to another
  - only a single user-level thread within a process can execute at a time
  - if one thread blocks, the entire process is blocked
- Kernel-level threads
  - threads within a process that are maintained by the kernel
  - a mode switch is required to switch from one thread to another
  - multiple threads within the same process can execute in parallel on a multiprocessor
  - blocking of a thread does not block the entire process