

*Operating Systems:
Internals and Design Principles, 8/E*
William Stallings



Chapter 4 Threads

Patricia Roy
Manatee Community
College, Venice, FL
©2015, Prentice Hall

Outline

- 4.1 Processes and Threads
- 4.2 Types of Threads
- 4.3 Multicore and Multithreading
- 4.4 Linux Process and Thread Management



Processes and Threads

- **Resource ownership** - process includes a virtual address space to hold the process image (**collection of program, data, stack, etc**) to prevent interference between processes
- **Scheduling/execution** - follows an execution path (**trace**) that may be interleaved with other processes
- These two characteristics are treated independently by the operating system

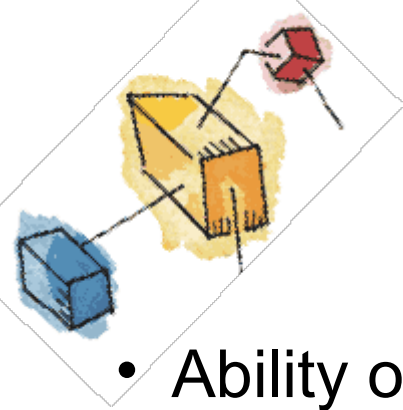




Processes and Threads

- 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

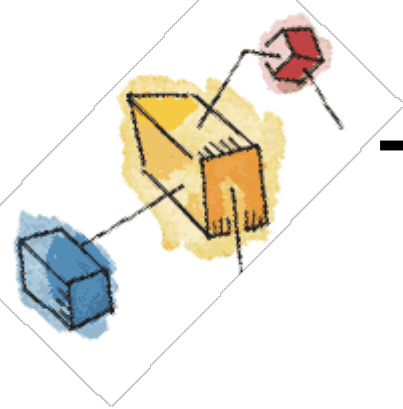




Multithreading

- Ability of OS to supports multiple, concurrent paths of execution within a single process
- MS-DOS supports a single user process and thread
- Traditional UNIX supports multiple user processes but only supports one thread per process
- Modern UNIX supports multiple processes with multiple threads
- Java run-time environment is a single process with multiple threads





Threads and Processes

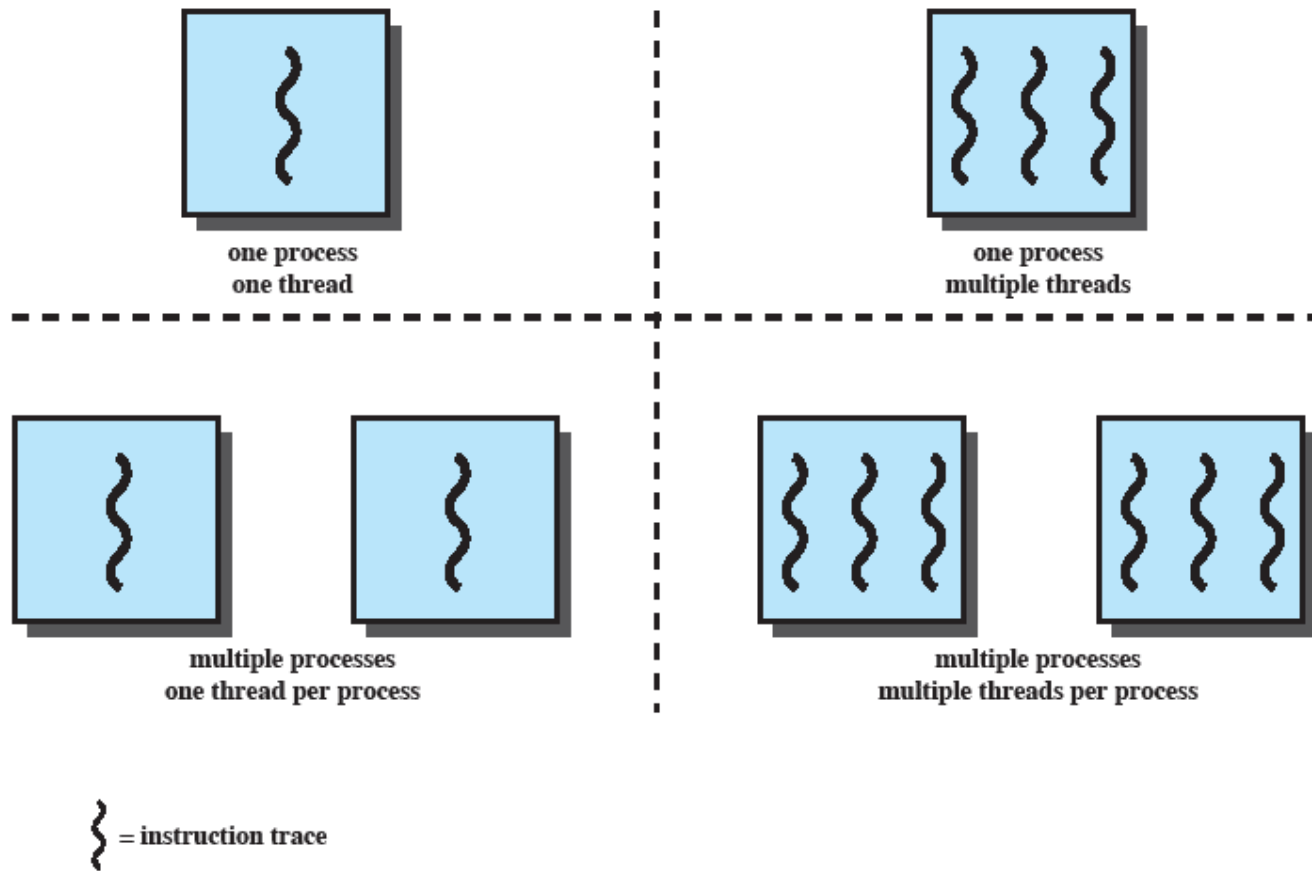
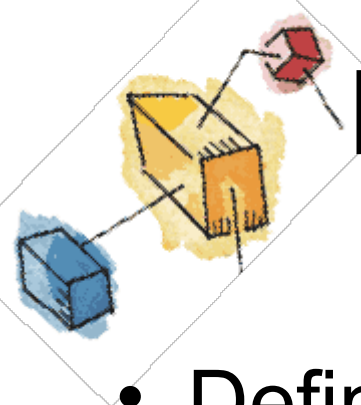


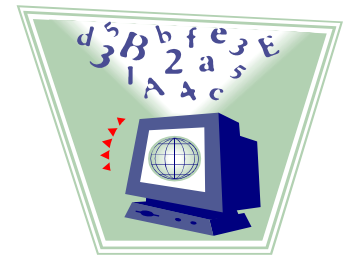
Figure 4.1 Threads and Processes [ANDE97]





Processes in multithreaded environment

- Defined as a **unit of resource allocation and unit of protection**
- A virtual address space which holds the process image
- Protected access to
 - processors,
 - other processes,
 - files, and I/O resources





One or More Threads in a Process

- There could be one / more threads in a process
- Each with the following characteristics:
 - 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



Threads

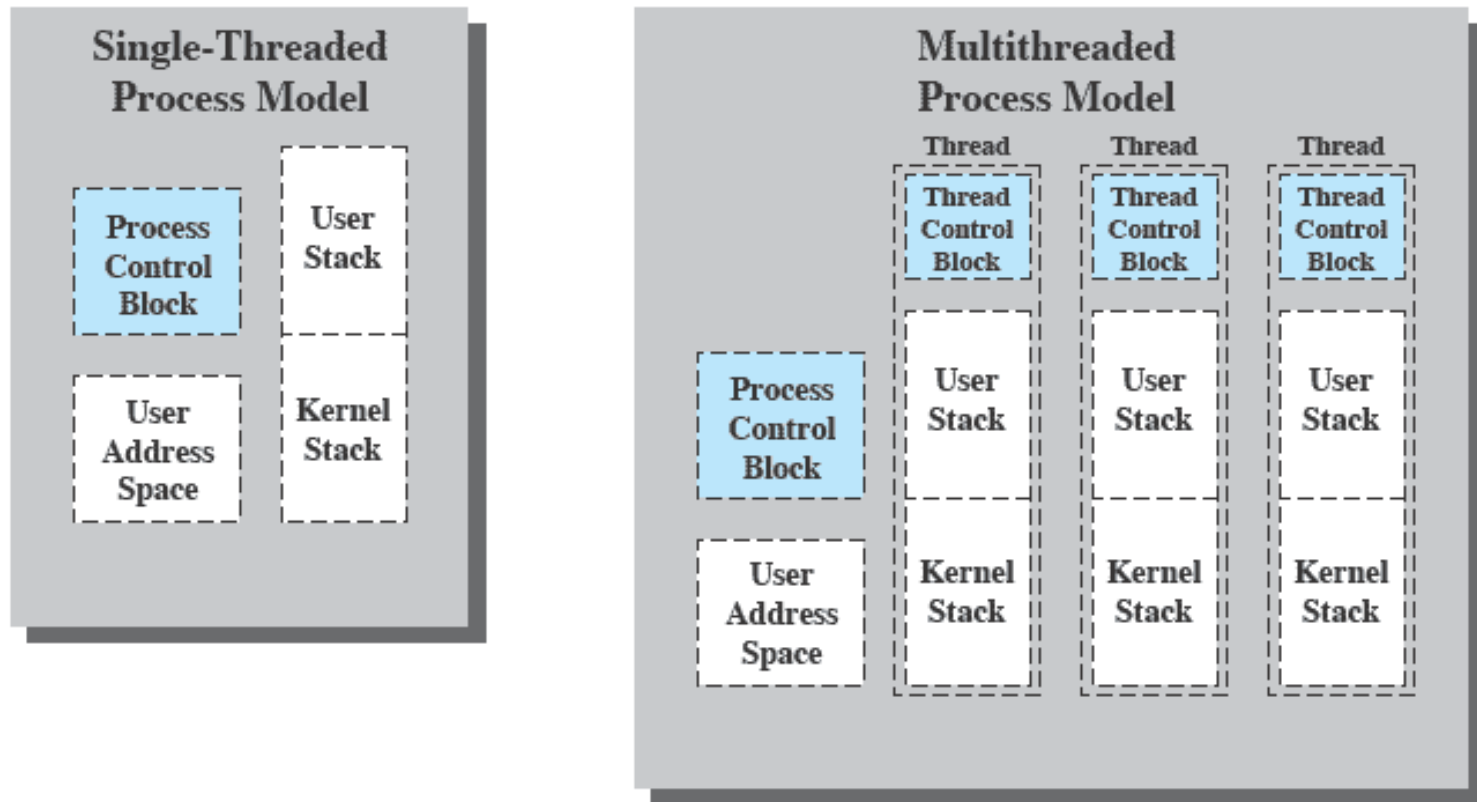
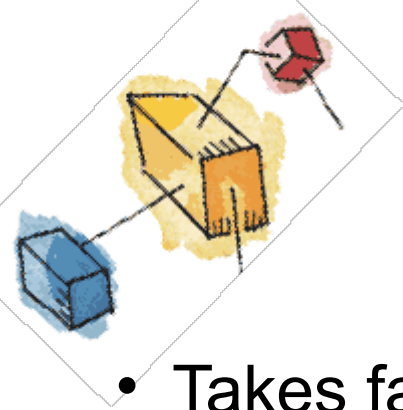


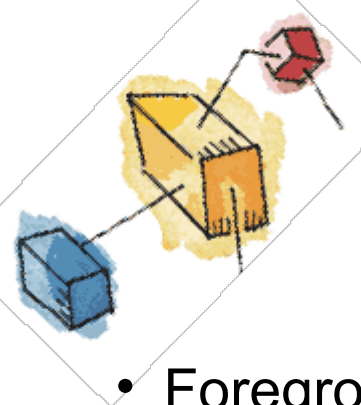
Figure 4.2 Single Threaded and Multithreaded Process Models



Benefits of Threads

- Takes far **less time to create** a new thread than creating a brand-new process (**10 x faster**)
- **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**, contradicts with communication between processes





Uses of Threads in a Single-User Multiprocessing System

- Foreground and background work
 - E.g. Spreadsheet program:
 - Thread1: display menu and read user input
 - Thread2: execute commands and updates the spreadsheet
- Asynchronous processing
- Speed of execution
 - Multiple threads from a process execute simultaneously
 - Eventhough one thread is blocked, another thread is available for execution
- Modular program structure
 - Separate into subsections
 - Useful for the program that involves variety of activities with a lot of sources and destinations

**** Scheduling and dispatching are done on a thread basis!**

– State-level information



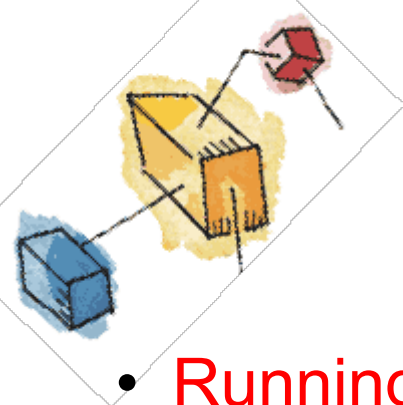


Actions That Affect All Threads in a Process

- **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

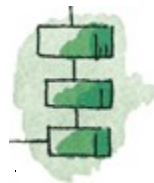
****Require management in the process level!**





Thread Functionality: (Thread States)

- Running, Ready and Blocked
- NO Suspend state → process level
- States associated with a change in thread state
 - Spawn
 - When a new process is spawned, a thread for that process is also spawned
 - A thread within a process may spawn another thread
 - Block – waiting for an event
 - Unblock – when an event for the blocked thread occurs
 - Finish
 - When complete, deallocate register context and stacks

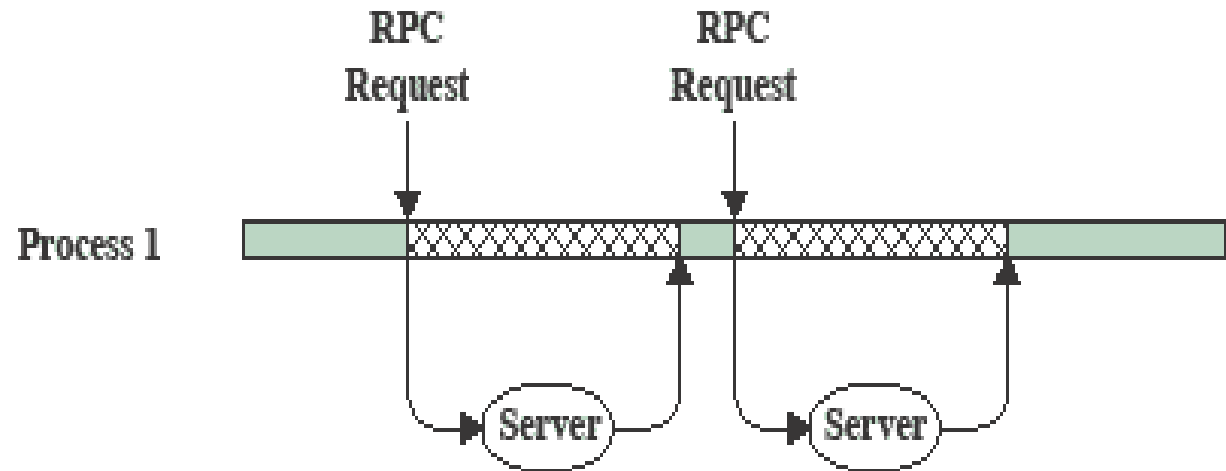




Remote Procedure Call (RPC) Using Single Thread

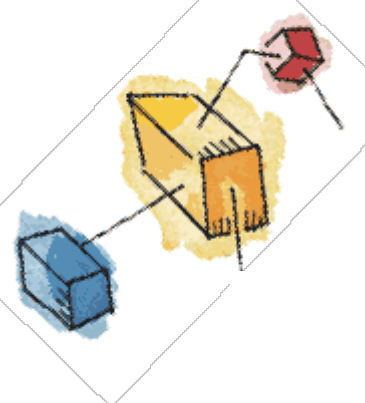
- Results are obtained in sequence
- Program has to wait in turn

Time →

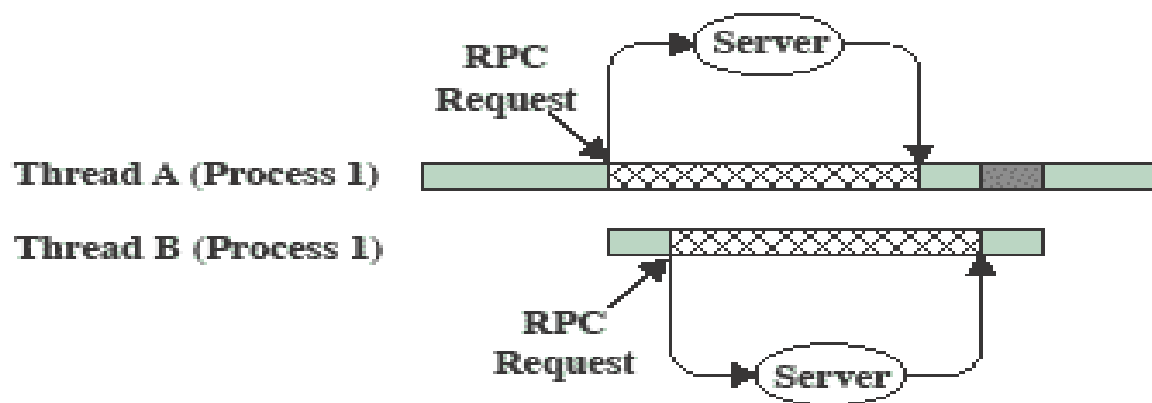


(a) RPC Using Single Thread








RPC Using One Thread per Server



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

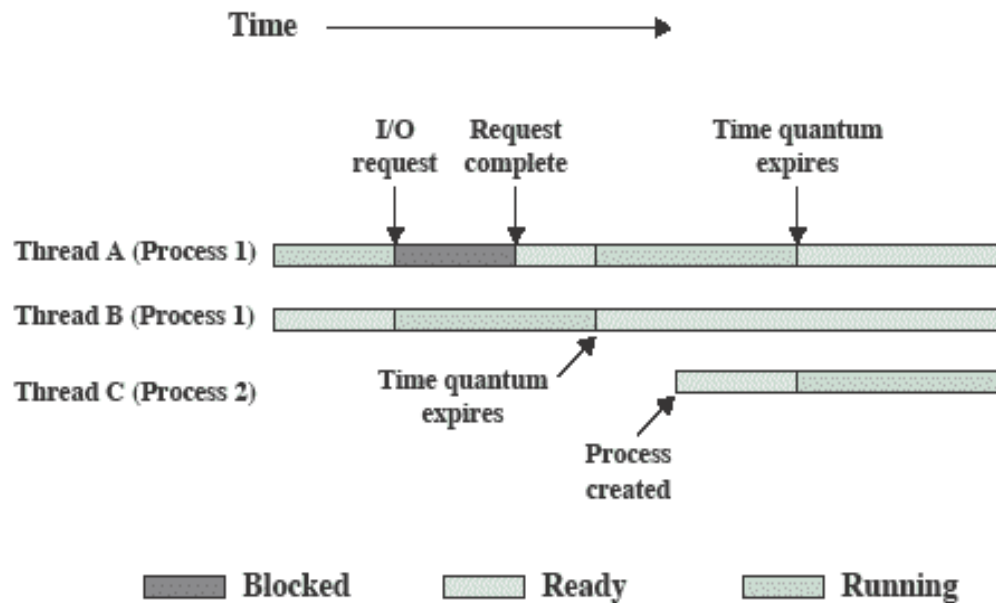
-  Blocked, waiting for response to RPC
-  Blocked, waiting for processor, which is in use by Thread B
-  Running

Rewriting the program to use separate thread will substantially speedup the process





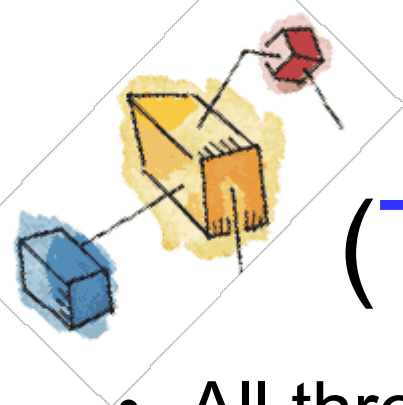
Multithreading



- On a uniprocessor,
 - Multiple threads are interleaved within multiple processes
- Execution passes from one thread to another
 - Blocked
 - Timely exhausted

Figure 4.4 Multithreading Example on a Uniprocessor





Thread Functionality: (Thread Synchronization)

- All threads of a process share the same address space & resources
- Any alteration of resources by one thread will affect other's environment in the same process
- Thus → necessary to synchronize the activities of various threads so they will not interfere and corrupt the data structure
- E.g: If 2 threads are trying to add an element to a linked list at the same time, one element may be lost or end up malfunction.

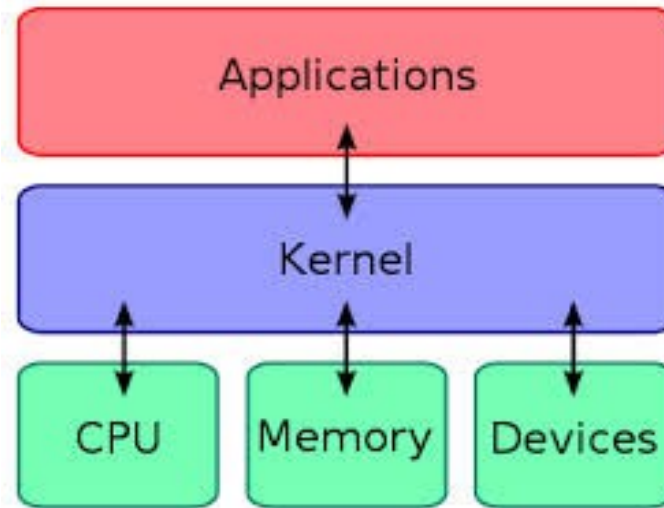


Types of threads

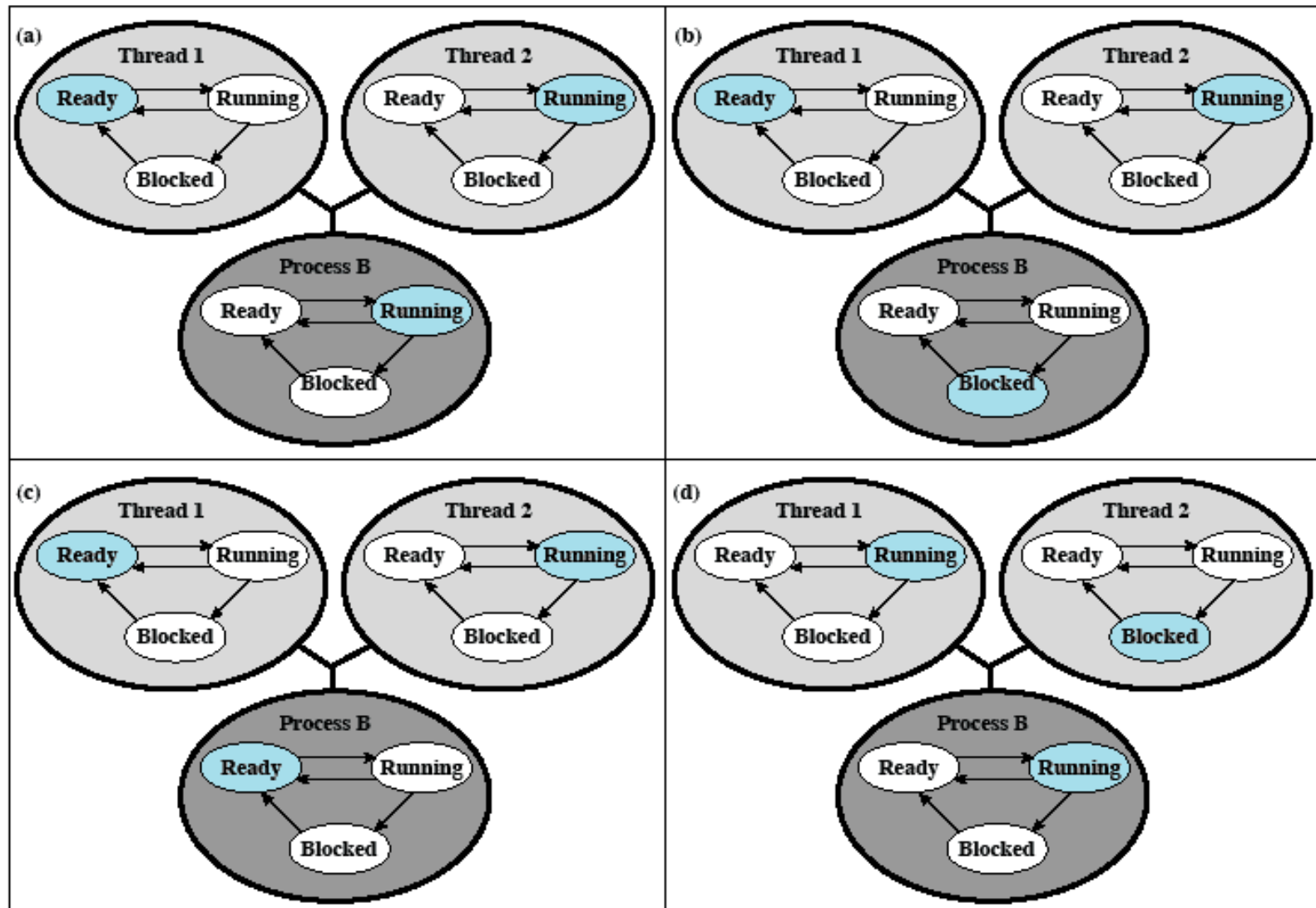
- User-Level



- Kernel-Level



Thread Scheduling Vs. Process Scheduling



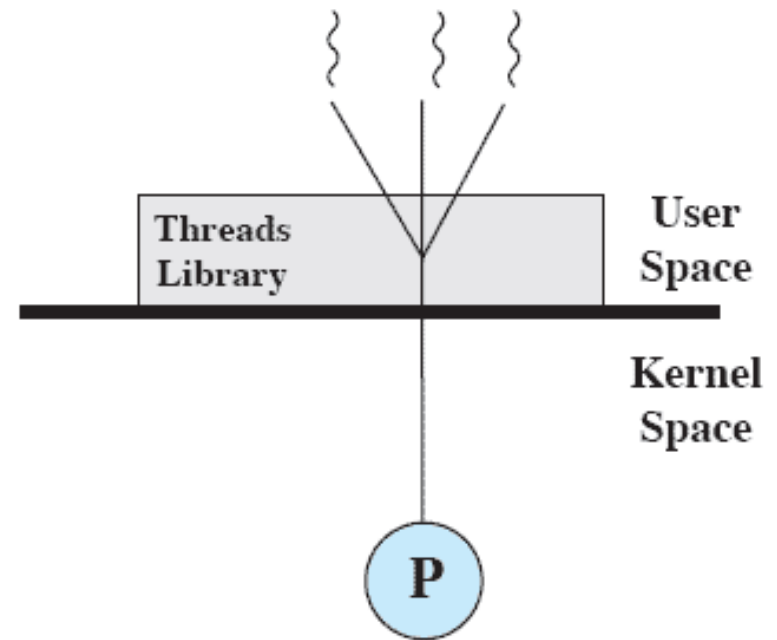
Colored state
is current state

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



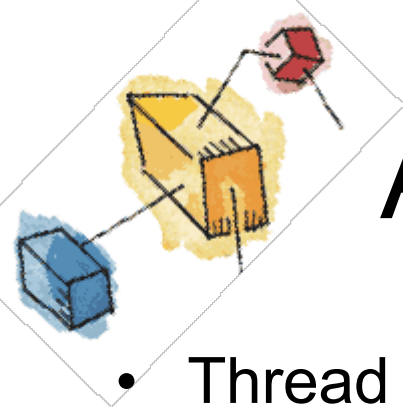
User-Level Threads (ULT)

- All thread management is done by the application
- The kernel is not aware of the existence of threads
- Can be programmed to be multithreaded using **thread library**:
 - Contains code for creating and destroying threads, message and data passing between threads, scheduling, execution, saving and storing thread context (PC, user register, stack



(a) Pure user-level

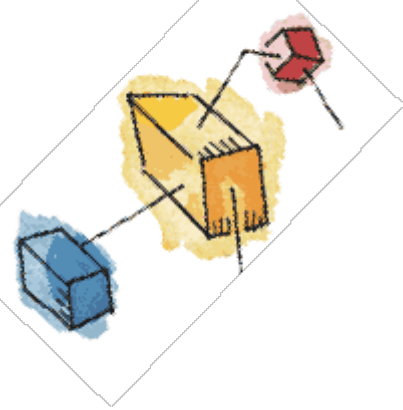




Advantages of using ULTs

- Thread switching does not require kernel mode privileges
 - All thread mgt data structure are within the user address space of a single process
 - Thus, no need to switch to Kernel mode
- Scheduling can be application specific ([Round-Robin or priority-based](#))
- ULTs can run on any OS – no changes on underlying kernel





Disadvantages

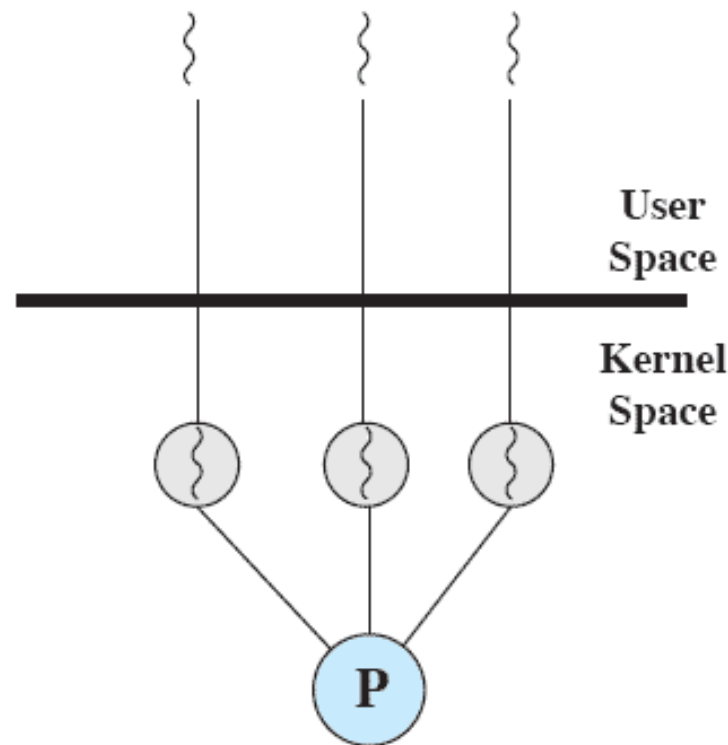
- If many system calls are blocking → all of the threads within a process will be blocked
- Multithreaded applications cannot take advantage of multiprocessing
 - A kernel assigns only one process to one processor at a time
 - Thus, only a single thread within one process can execute at a time
- **Solution:**
 - Multiple processes rather than multiple threads, but this will eliminate the advantages of the threads
 - Jacketing → convert a blocking system call into a non-blocking system call
 - Jacketing code will check if the requested I/O device is busy. If it is, the thread enters the Blocked state, and the control is passed to another state.



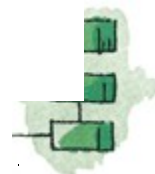


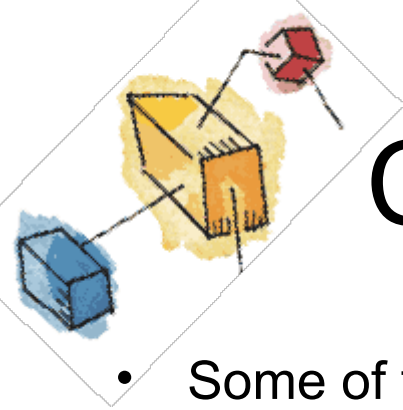
Kernel-Level Threads (KLT)

- Windows is an example of this approach
- Kernel maintains context information for the process and the threads
- Scheduling is done on a thread basis
- **Overcome ULT drawbacks:**
 - Simultaneously schedule multiple threads on different processors
 - When one thread in a process is blocked, the kernel can schedule another thread of the same process
- **Disadvantage:**
 - Transfer of control from one thread to another requires a mode switch (switching from user mode to kernel mode)



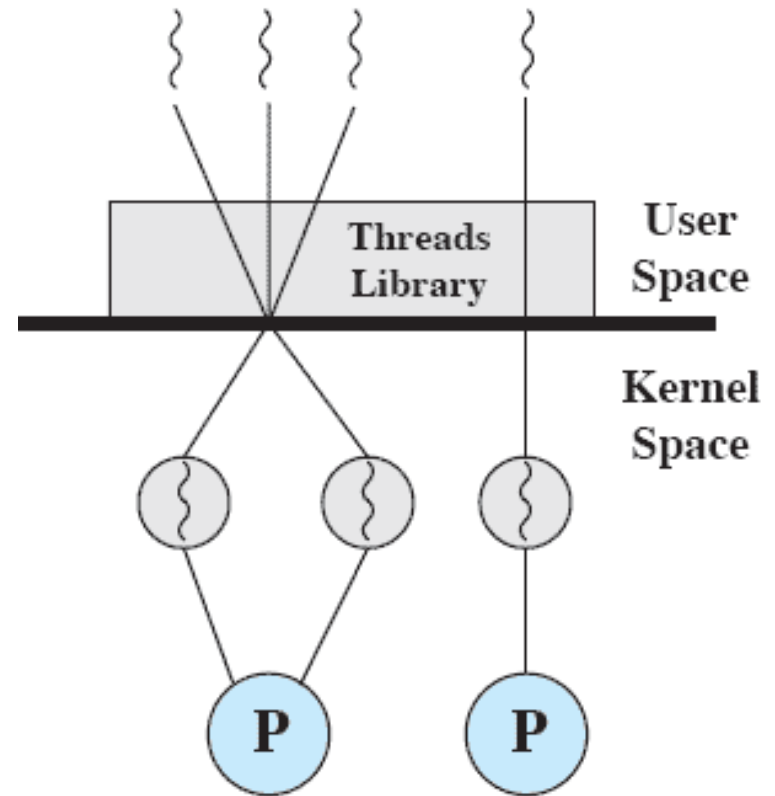
(b) Pure kernel-level





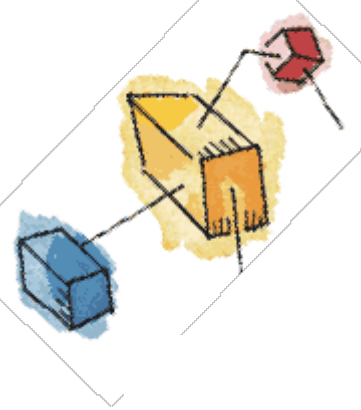
Combined Approaches

- Some of the OS combine ULT and KLT:
 - Example is Solaris
- Thread creation done in the user space
- Also the bulk of scheduling and synchronization of threads within application
- Multiple threads within same applications can run in parallel on multiple processors
- Blocking system call may not block the entire process



(c) Combined



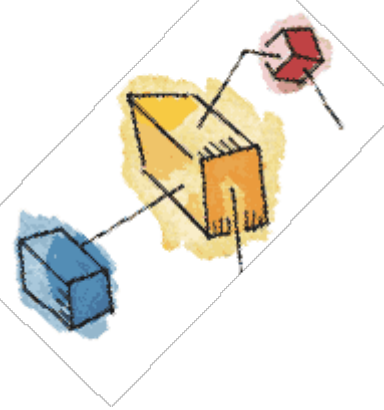


Relationship Between Thread and Processes

Table 4.2 Relationship Between Threads and Processes

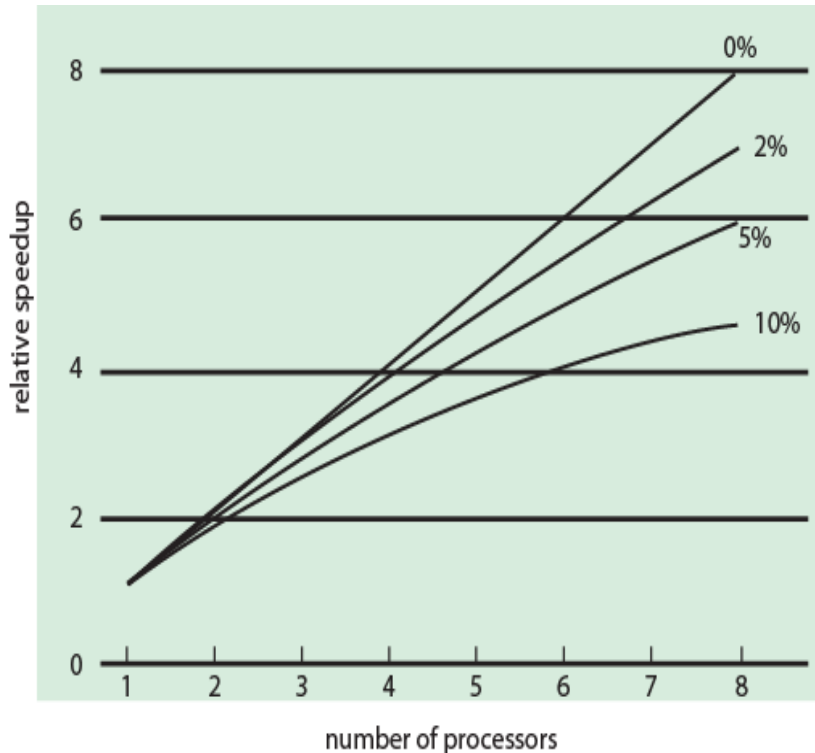
Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	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



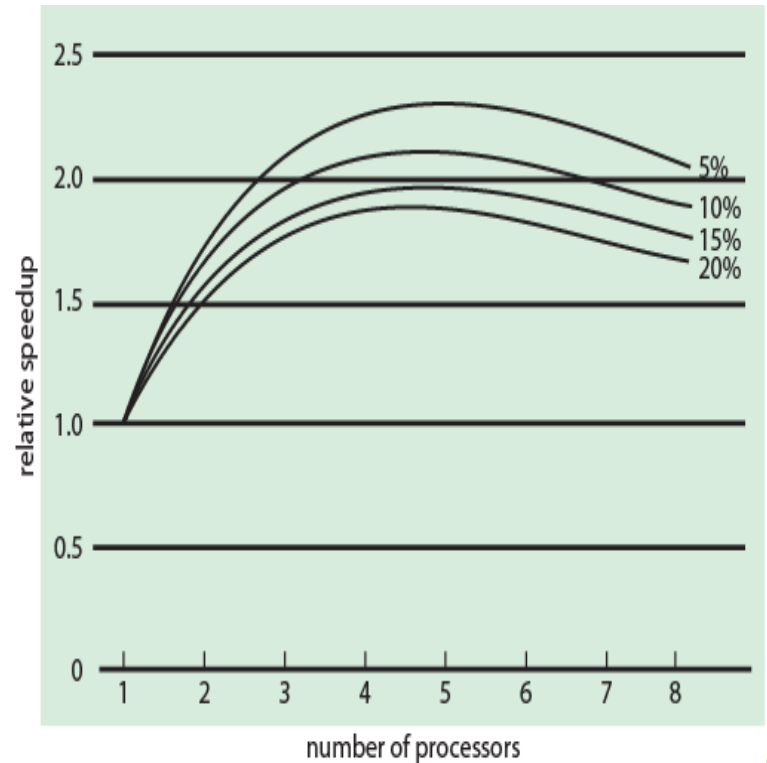


Multicore & Multithreading

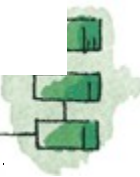
Performance effect on MultiCore

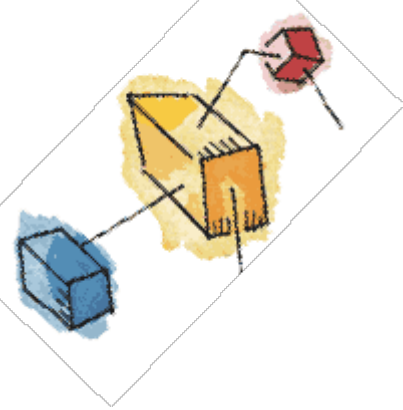


(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

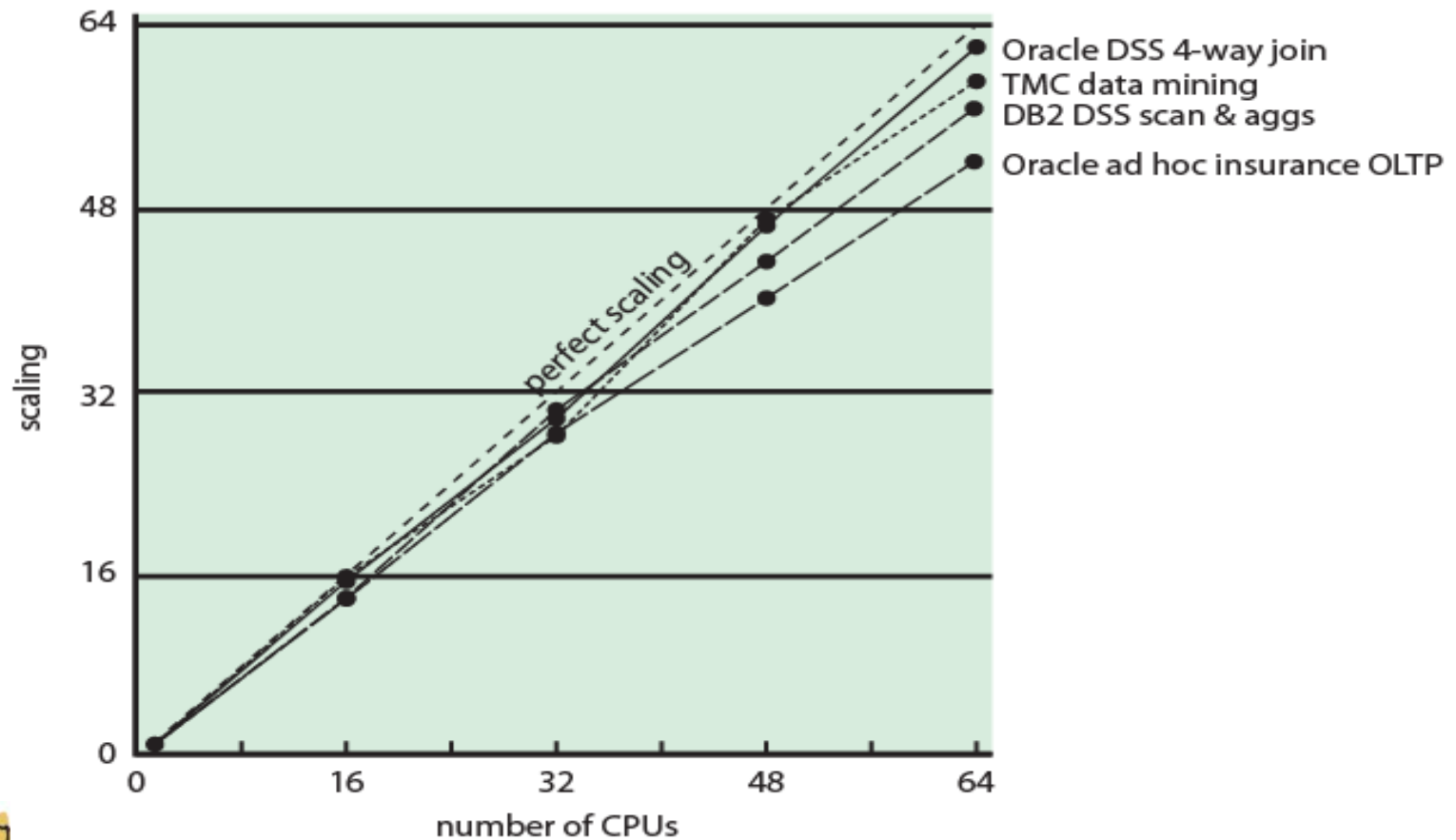


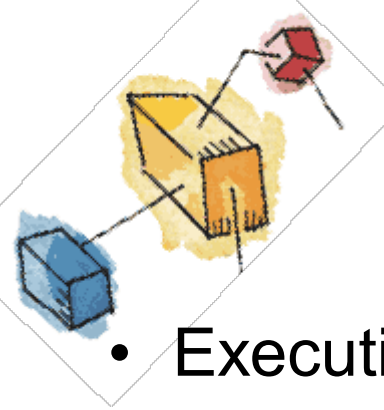
(b) Speedup with overheads





Database workload on Multi Processor Hardware





Linux Process & Thread Management (Linux Tasks)

- Execution State (**Ready, executing, suspend, stopped**)
- Scheduling information (**normal or real time; priority**)
- Identifiers
- Interprocess communication
- Links (**parents, siblings, children**)
- Times and timers (**process creation, processor time, interval for system call**)
- File system (**pointers to any files opened**)
- Address space (**Virtual address**)
- Processor-specific context (**register and stack information**)



Linux Process/Thread Model

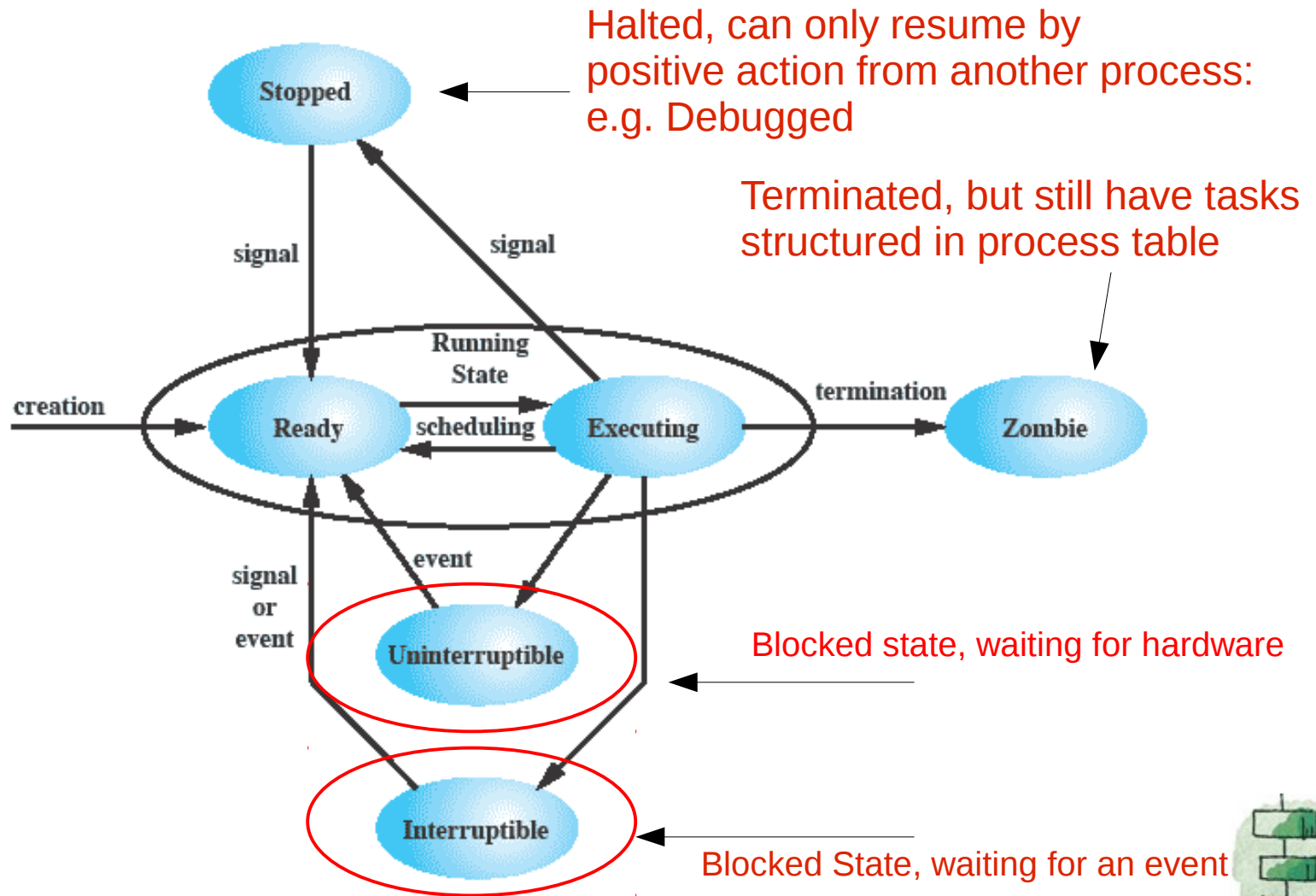
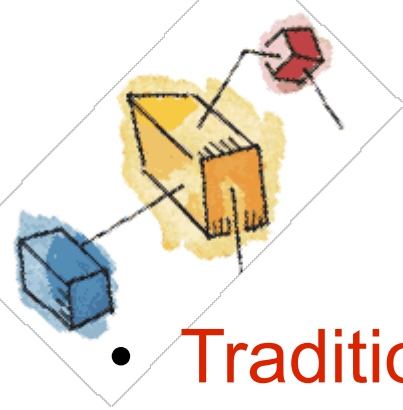


Figure 4.18 Linux Process/Thread Model

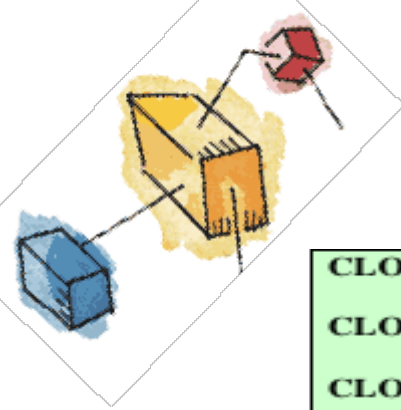


Linux Threads

- **Traditional**: Support single thread of execution per process. All threads are mapping into a single kernel-level process
- **Modern**: support multiple kernel thread. Does not recognize between thread and process
- A process is created by **copying** the attributes of the current process.
- A new process can be **cloned** so that it shares resources such as files, and virtual memory.

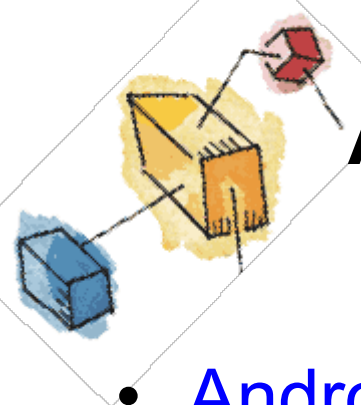


Linux Clone () Flags



CLONE_CLEARID	Clear the task ID.
CLONE_DETACHED	The parent does not want a SIGCHLD signal sent on exit.
CLONE_FILES	Shares the table that identifies the open files.
CLONE_FS	Shares the table that identifies the root directory and the current working directory, as well as the value of the bit mask used to mask the initial file permissions of a new file.
CLONE_IDLETASK	Set PID to zero, which refers to an idle task. The idle task is employed when all available tasks are blocked waiting for resources.
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Caller and new task share the same parent process.
CLONE_PTRACE	If the parent process is being traced, the child process will also be traced.
CLONE_SETTID	Write the TID back to user space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Shares the table that identifies the signal handlers.
CLONE_SYSVSEM	Shares System V SEM_UNDO semantics.
CLONE_THREAD	Inserts this process into the same thread group of the parent. If this flag is true, it implicitly enforces CLONE_PARENT.
CLONE_VFORK	If set, the parent does not get scheduled for execution until the child invokes the <i>execve()</i> system call.
CLONE_VM	Shares the address space (memory descriptor and all page tables).





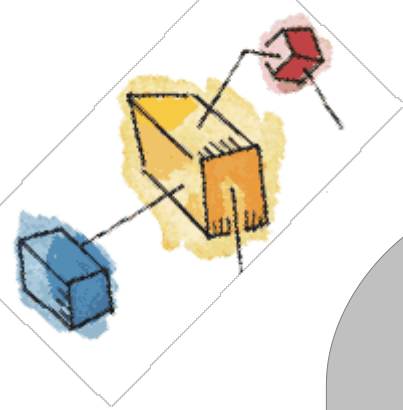
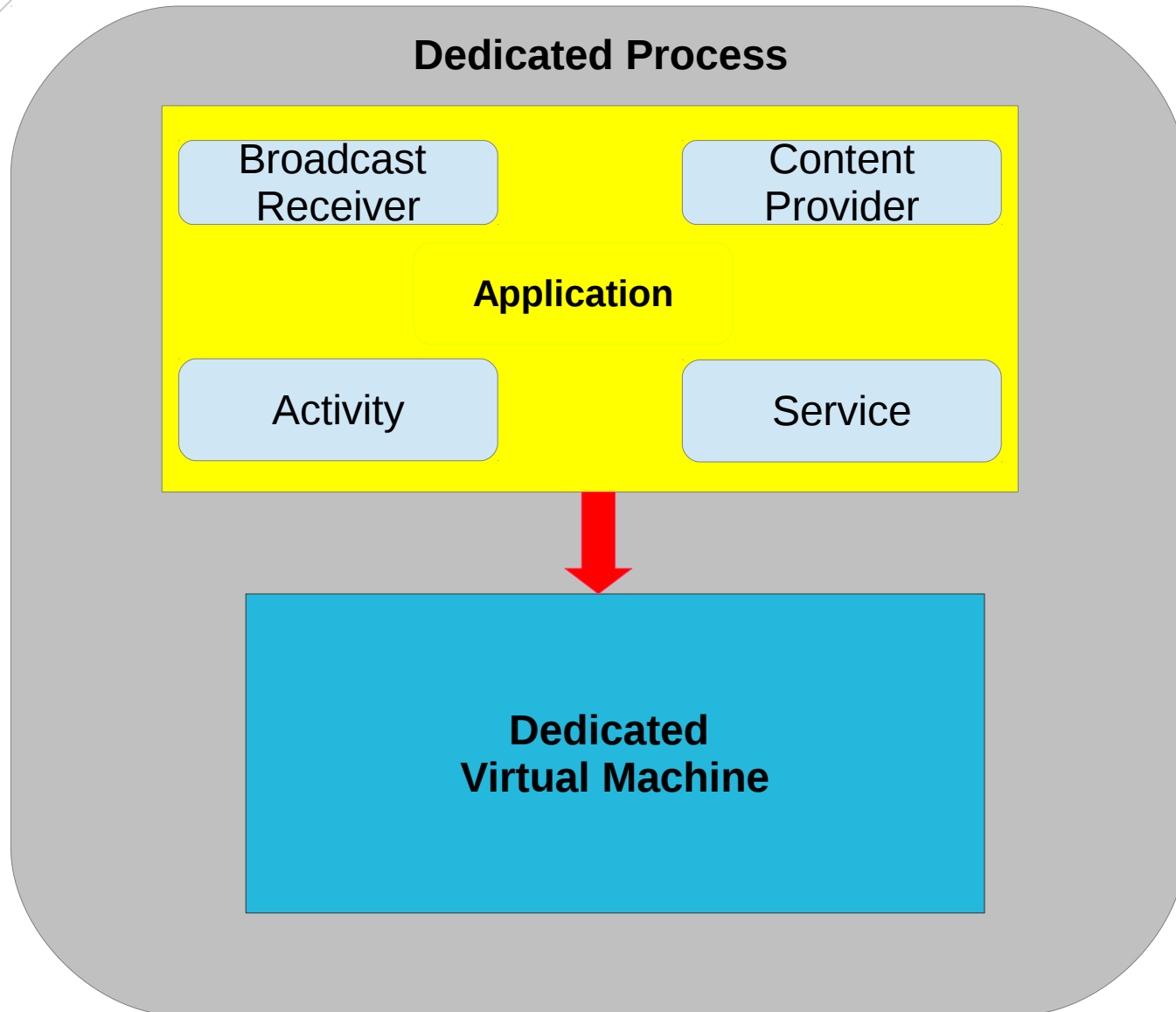
Android Process and Thread Management

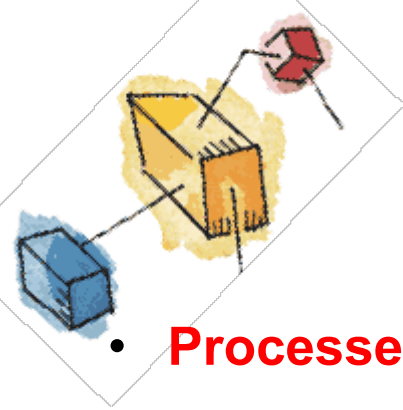
- **Android Applications:**

- **Activities**: corresponds to a single screen visible as a user interface. E.g: **email application** might have one activity that shows a list of new emails, compose, read, etc...
- **Services**: Used to perform background operation that takes considerable amount of time to finish. E.g: **create thread or process to play music** at background.
- **Content provider**: acts as an interface to application data that can be used by an app. E.g **NotePad** uses content provided to save notes.
- **Broadcast Provided**: responds to a system-wide broadcast announcements. E.g: **Let other apps know that some data has been downloaded to the device.**



Android Application





Android:

Process and Threads

- **Processes are killed beginning with the lowest precedence:**
 - **Foreground Process:** A process that is required for what the user is currently doing.
 - **Visible process:** A process that hosts a component that is not in the foreground, but still visible to the user
 - **Service Process:** A process running a service that does not fall into either of the higher categories. E.g: playing music in the background while downloading data from network
 - **Background process:** A process hosting an activity in the stopped state
 - **Empty process:** A process that doesn't hold any active application components. Is keeping alive for caching purposes, to improve startup time the next time a component needs to run in it.



Summary

•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

- Process/related to resource ownership

- Thread/related to program execution

