Operating Systems

Instructor: Dr. Shachi Sharma

(Semester: Winter 2019)

Threads and Threads Management

Slides Courtesy: William Stallings

Processes and Threads

Traditional processes have two characteristics:

Resource Ownership

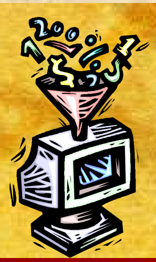
Process includes a virtual address space to hold the process image

- the OS provides protection 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
- Traditional processes are *sequential*; i.e. only *one* execution path



Processes and Threads

- ***Multithreading -*** The ability of an OS to support multiple, concurrent paths of execution within a single process
- The unit of resource ownership is referred to as a ***process*** or ***task***
- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***

Single Threaded Approaches

- A single execution path per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- MS-DOS, some versions of UNIX supported only this type of process.

Multithreaded Approaches

- The right half of Figure 4.1 depicts multithreaded approaches
- A Java run-time environment is a system of *one* process with multiple threads; Windows, some UNIXes, support *multiple* multithreaded processes.

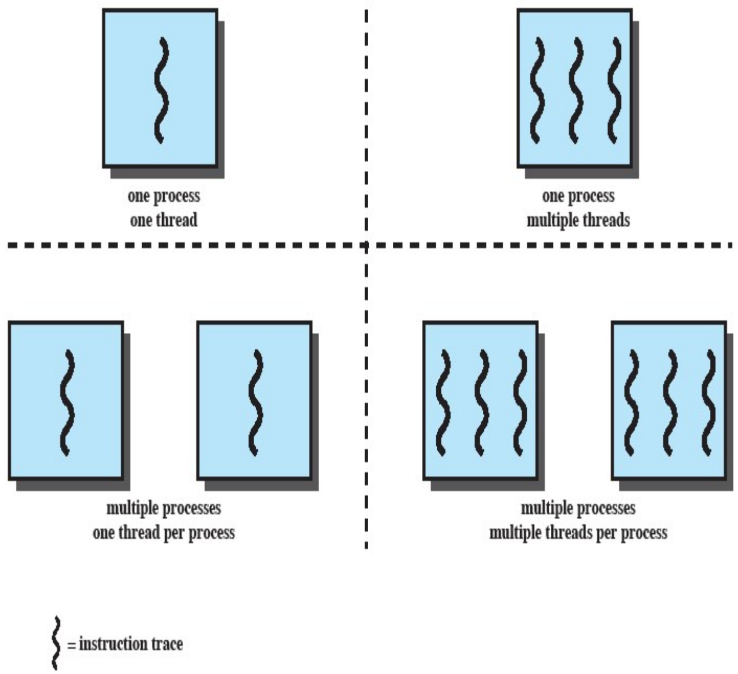
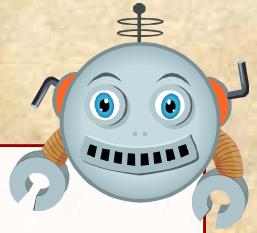


Figure 4.1 Threads and Processes [ANDE97]

Processes

- In a multithreaded environment the process is the unit that owns resources and the unit of protection.
  - i.e., the OS provides protection at the process level
- Processes have
  - A virtual address space that holds the process image
  - Protected access to
    - processors
    - other processes
    - files
    - I/O resources

One or More Threads in a Process



Each thread has:

- an execution state (Running, Ready, etc.)
- saved thread context when not running (TCB)
- an execution stack
- some per-thread static storage for local variables
- access to the shared memory and resources of its process (all threads of a process share this)

Threads vs. Processes

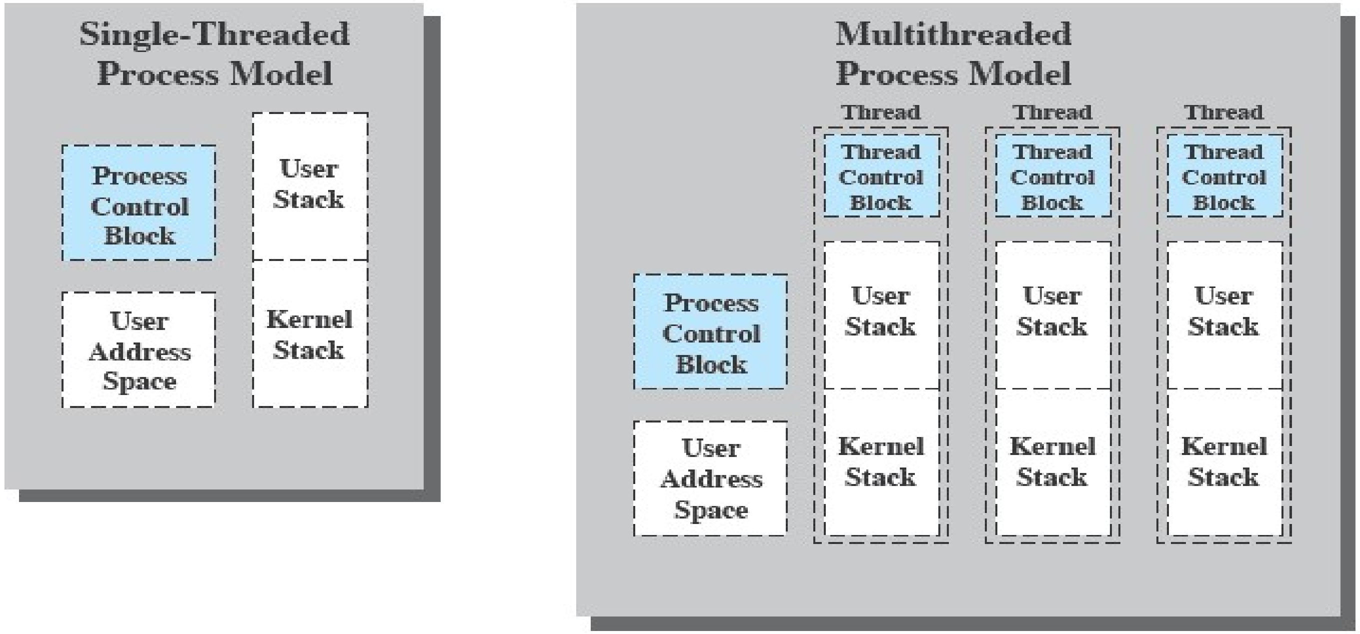


Figure 4.2 Single Threaded and Multithreaded Process Models

Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs

Thread Use in a Single-User System

- Foreground and background work
- Asynchronous processing
- Speed of execution
- Modular program structure

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

- suspending a process involves suspending all threads of the process
- termination of a process terminates all threads within the process

Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn (create)
- Block
- Unblock
- Finish

Thread Execution

- A key issue with threads is whether or not they can be scheduled independently of the process to which they belong.
- Or, is it possible to block one thread in a process without blocking the entire process?
    - If not, then much of the flexibility of threads is lost.

Multithreading on a Uniprocessor

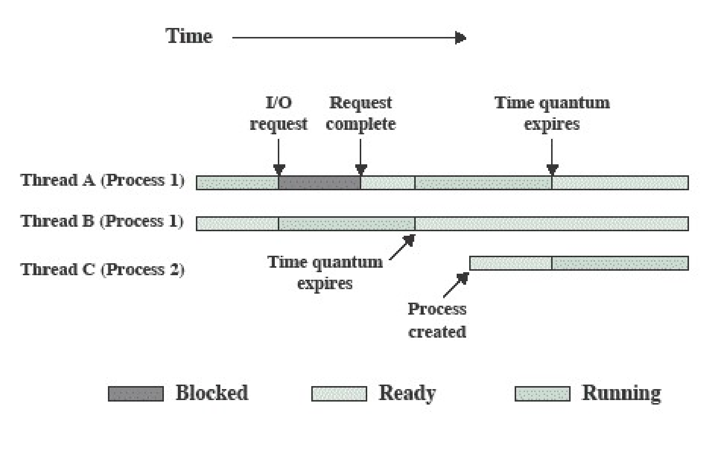
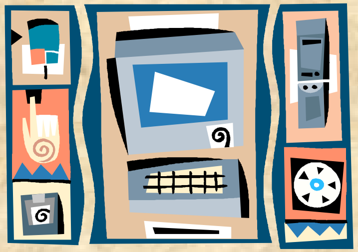
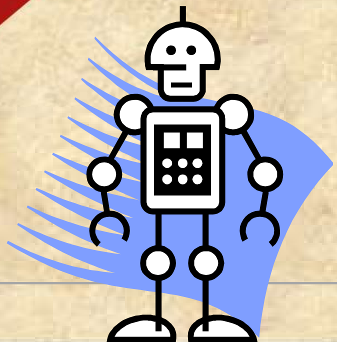


Figure 4.4 Multithreading Example on a Uniprocessor



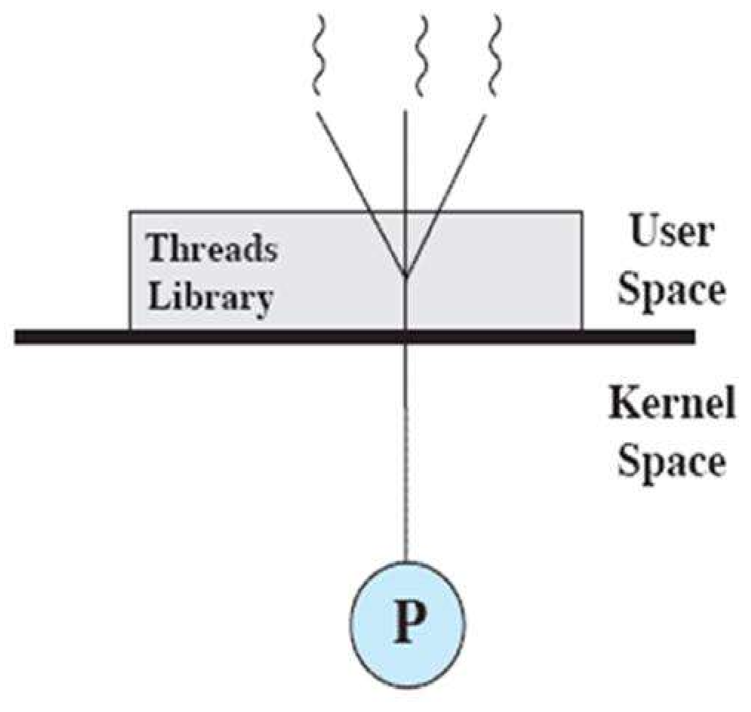
Types of Threads

NOTE: we are talking about threads for *user* processes. Both ULT & KLT execute in user mode. An OS may also have threads but that is not what we are discussing here.



User-Level Threads (ULTs)

- Thread management is done by the application
- The kernel is not aware of the existence of threads
- Not the kind we've discussed so far.



(a) Pure user-level

Relationships Between ULT States and Process States

Possible transitions from 4.6a:

4.6a→4.6b
4.6a→4.6c
4.6a→4.6d

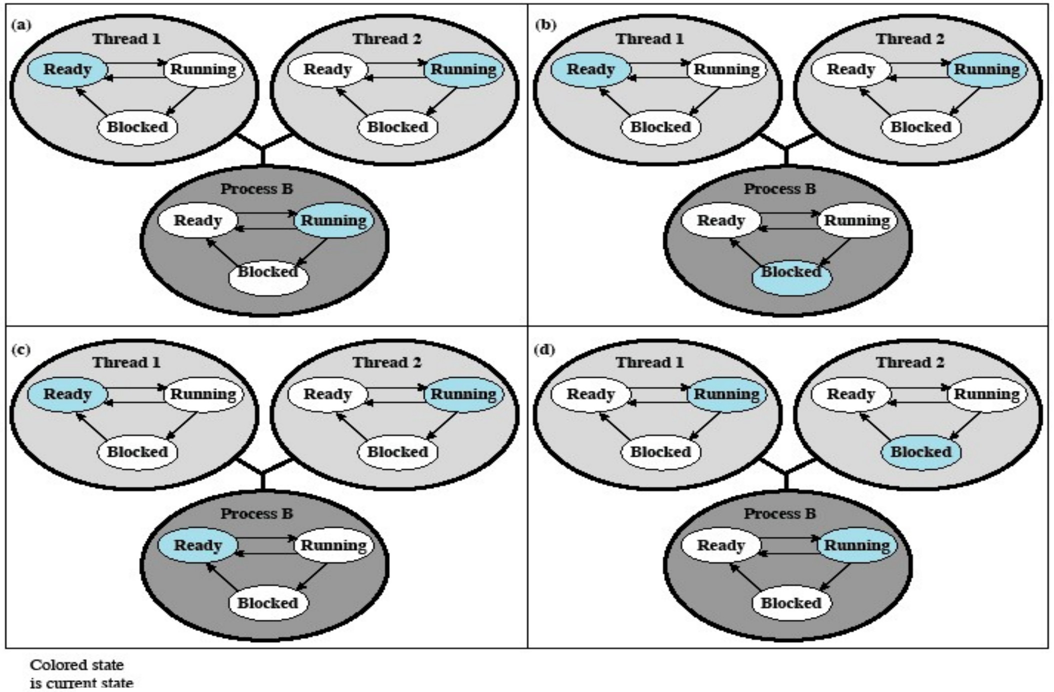
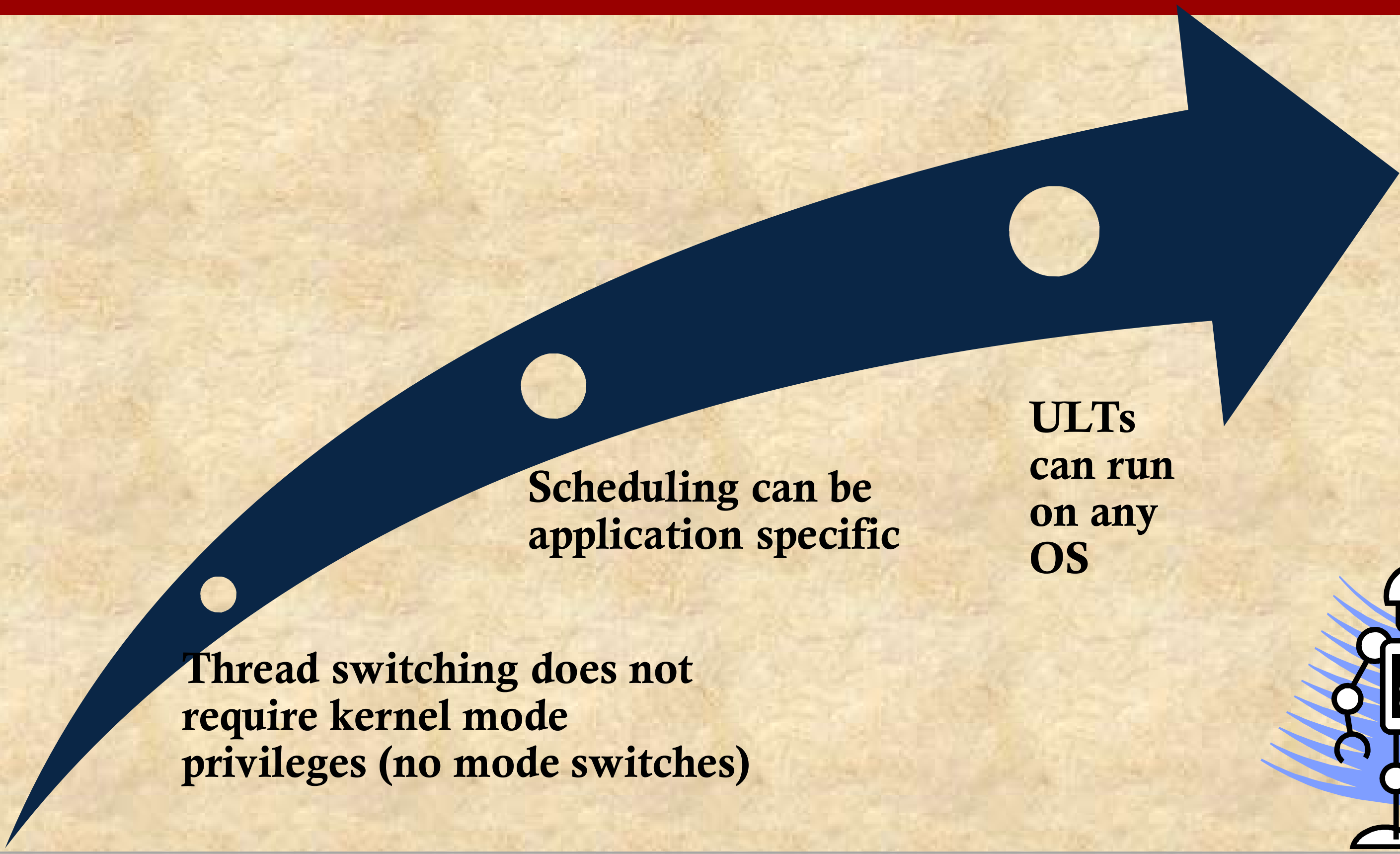
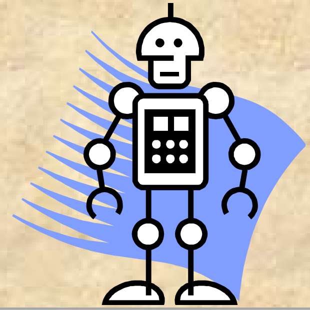


Figure 4.6 Examples of the Relationships between User-Level Thread States and Process States

Advantages of ULTs





Disadvantages of ULTs

- 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

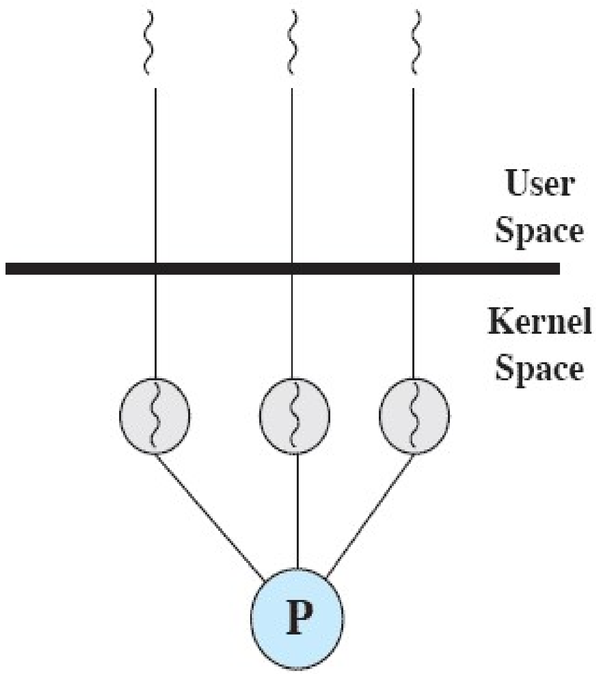
Overcoming ULT Disadvantages

Jacketing

- converts a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads

Kernel-Level Threads (KLTs)

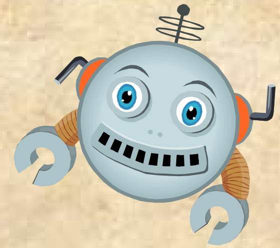


(b) Pure kernel-level

- Thread management is done by the kernel (could call them K*M*T)
  - no thread management is done by the application
  - Windows is an example of this approach

Advantages of KLTs

- 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

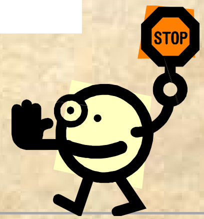


Disadvantage of KLTs

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

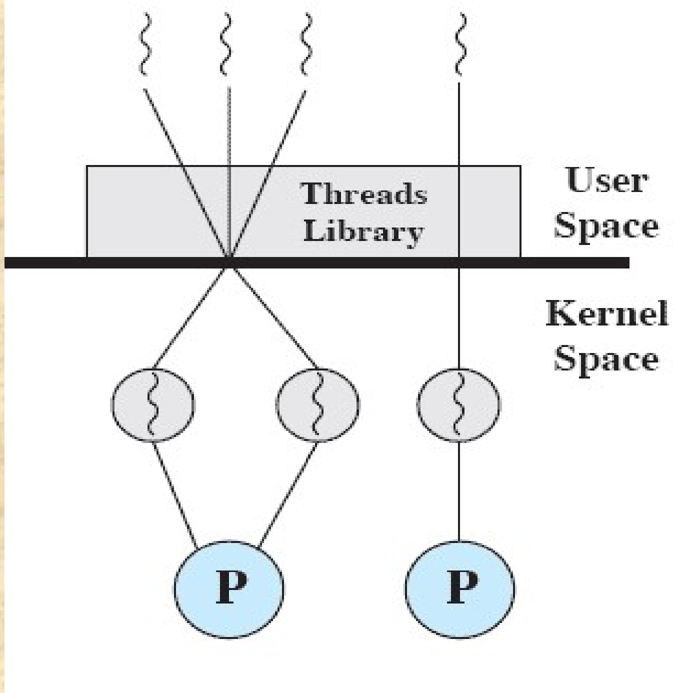
Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1 Thread and Process Operation Latencies (µs)



Combined Approaches

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- Solaris is an example



(c) Combined

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

Table 4.2 Relationship between Threads and Processes

Multiple Cores & Multithreading

- Multithreading and multicore chips have the potential to improve performance of applications that have large amounts of parallelism
- Gaming, simulations, etc. are examples
- Performance doesn't necessarily scale linearly with the number of cores …

Amdahl's Law

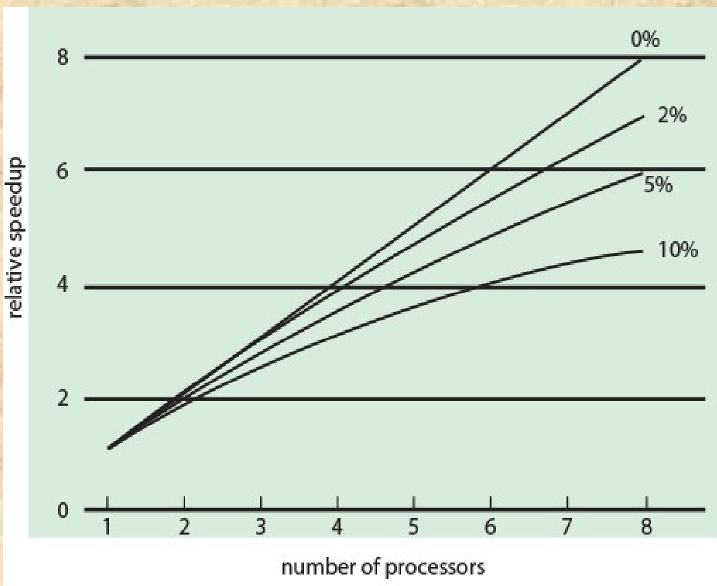
- Speedup depends on the amount of code that must be executed sequentially
- Formula:

$$\text{Speedup} = \frac{\text{time to run on single processor}}{\text{time to execute on N || processors}}$$

$$= \frac{1}{(1 - f) + f\,/\,N}$$

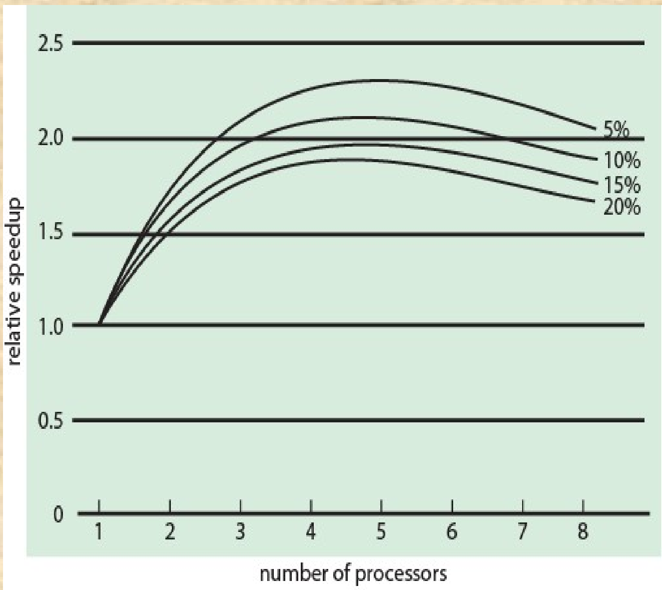
(where f is the amount of parallelizable code)

Performance Effect of Multiple Cores



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

Figure 4.7 (a)



(b) Speedup with overheads

Figure 4.7 (b)

Database Workloads on Multiple-Processor Hardware

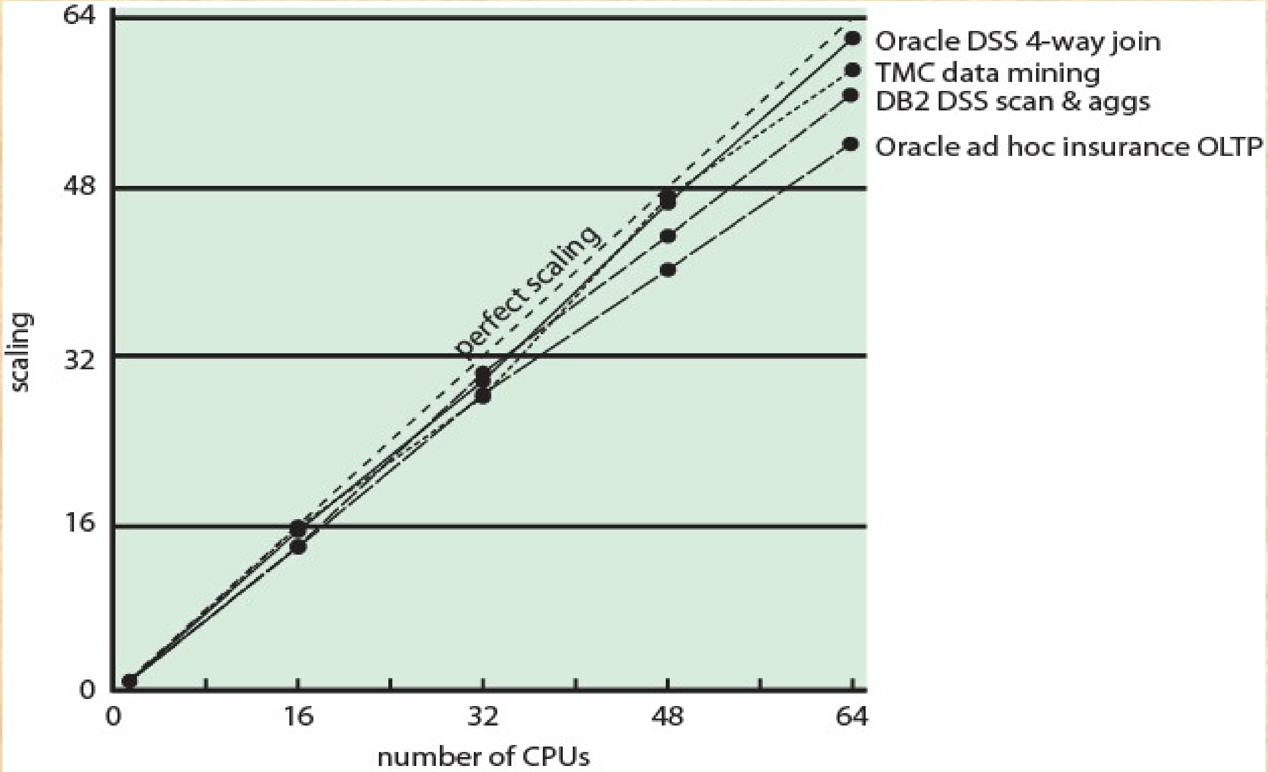


Figure 4.8 Scaling of Database Workloads on Multiple Processor Hardware

Applications That Benefit [MCDO06]

- Multithreaded native applications
  - characterized by having a small number of highly threaded processes
- Multiprocess applications
  - characterized by the presence of many single-threaded processes
- Java applications
- Multi-instance applications
  - multiple instances of the application in parallel

Windows Processes

Processes and services provided by the Windows Kernel are relatively simple and general purpose

- implemented as objects
- created as new process or a copy of an existing
- an executable process may contain one or more threads
- both processes and thread objects have built-in synchronization capabilities

Relationship Between Process and Resource

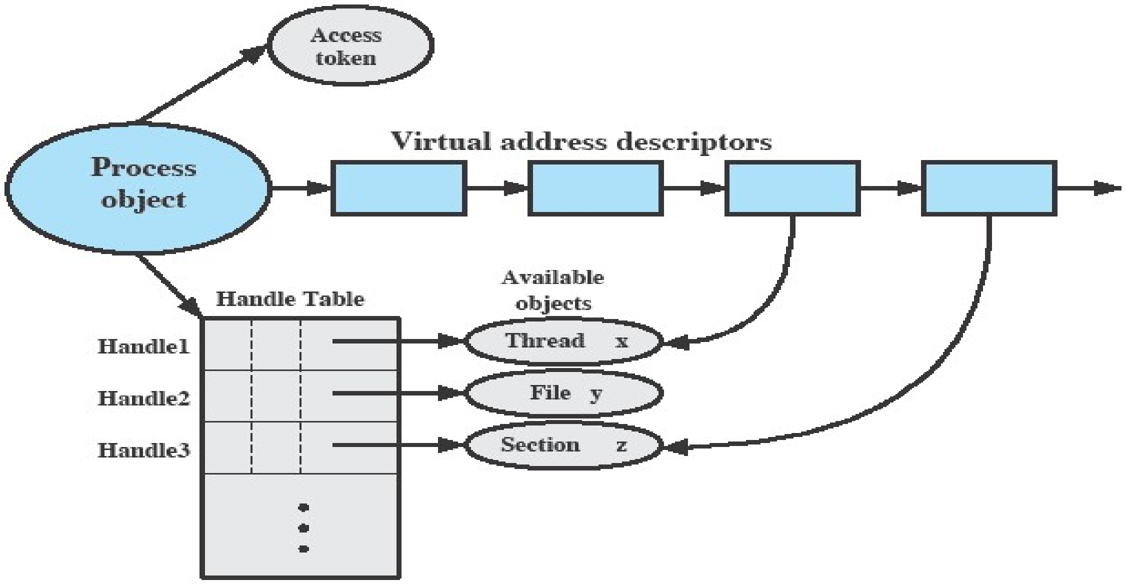
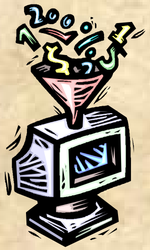


Figure 4.10 A Windows Process and Its Resources
Figure 4.12 A Windows Process and Its Resources

Process and Thread Objects

Windows makes use of two types of process-related objects:

Processes	Threads
• an entity corresponding to a user job or application that owns resources	• a dispatchable unit of work that executes sequentially and is interruptible



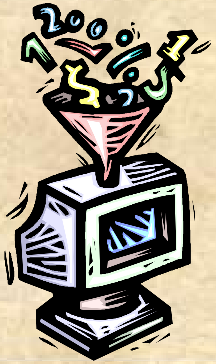
Windows Process and Thread Objects

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

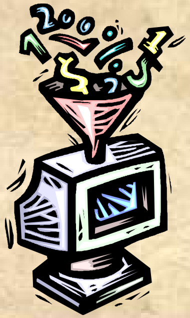


Windows Process Object Attributes

Process ID	A unique value that identifies the process to the operating system.
Security descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that the process's threads have performed.
Exception/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally, these are connected to environment subsystem and debugger processes, respectively.
Exit status	The reason for a process's termination.

Table 4.3 Windows Process Object Attributes

Windows Thread Object Attributes

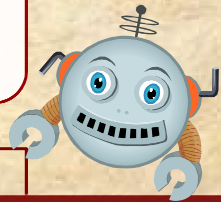


Thread ID	A unique value that identifies a thread when it calls a server.
Thread context	The set of register values and other volatile data that defines the execution state of a thread.
Dynamic priority	The thread's execution priority at any given moment.
Base priority	The lower limit of the thread's dynamic priority.
Thread processor affinity	The set of processors on which the thread can run, which is a subset or all of the processor affinity of the thread's process.
Thread execution time	The cumulative amount of time a thread has executed in user mode and in kernel mode.
Alert status	A flag that indicates whether a waiting thread may execute an asynchronous procedure call.
Suspension count	The number of times the thread's execution has been suspended without being resumed.
Impersonation token	A temporary access token allowing a thread to perform operations on behalf of another process (used by subsystems).
Termination port	An interprocess communication channel to which the process manager sends a message when the thread terminates (used by subsystems).
Thread exit status	The reason for a thread's termination.

Table 4.4 Windows Thread Object Attributes

Multithreaded Process

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
- Threads in different processes can exchange information through shared memory that has been set up between the two processes

Thread States

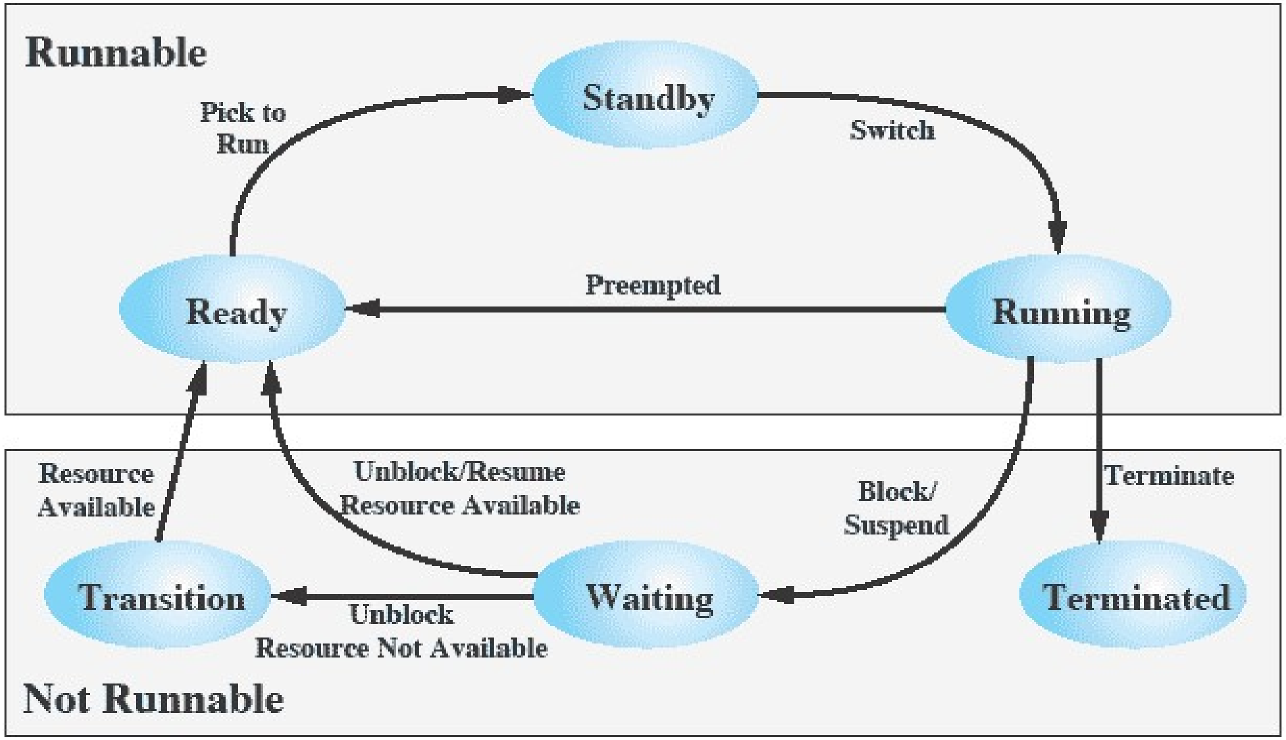


Figure 4.12 Windows Thread States

Symmetric Multiprocessing Support (SMP)

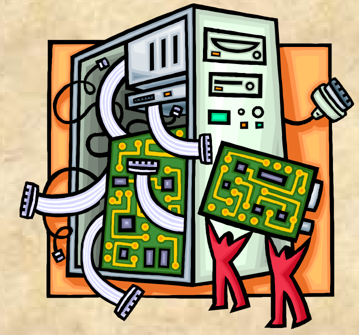
Threads of any process can run on any processor

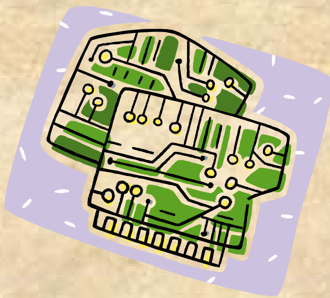
Soft Affinity

- the dispatcher tries to assign a ready thread to the same processor it last ran on
- helps reuse data still in that processor's memory caches from the previous execution of the thread

Hard Affinity

- an application restricts thread execution to certain processors



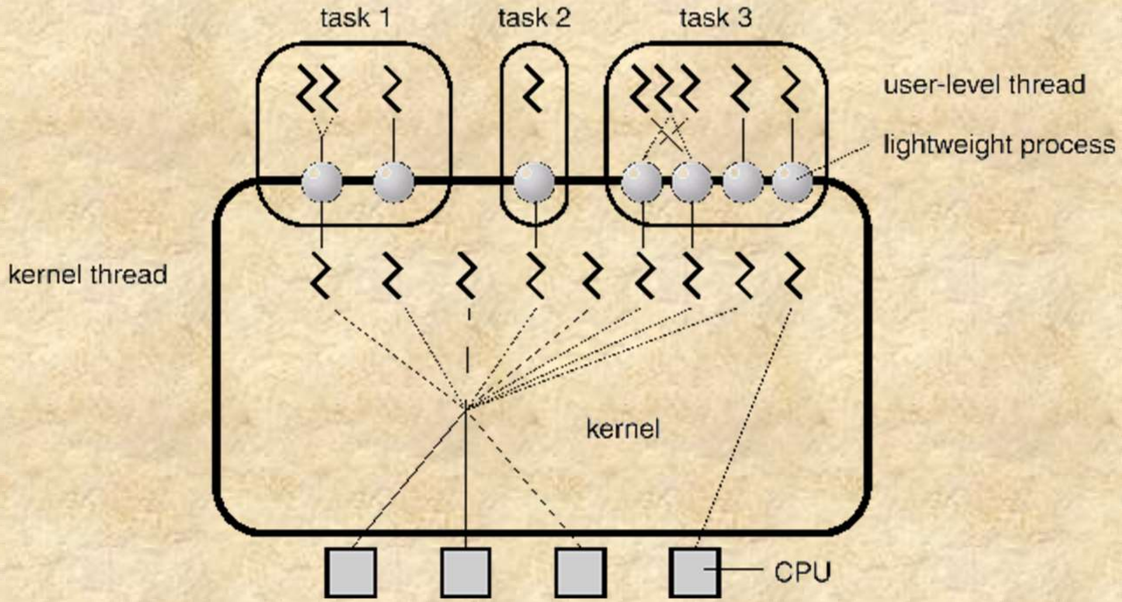


Solaris Process

- makes use of four thread-related concepts:

Process	• includes the user's address space, stack, and process control block
User-level Threads	• a user-created unit of execution within a process
Lightweight Processes (LWP)	• a mapping between ULTs and kernel threads
Kernel Threads	• fundamental entities that can be scheduled and dispatched to run on one of the system processors

Solaris ULTs, LWPs and KLTs



A Lightweight Process (LWP) Data Structure Includes:

- An LWP identifier
- The priority of this LWP
- A signal mask
- Saved values of user-level registers
- The kernel stack for this LWP
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

Solaris Thread States

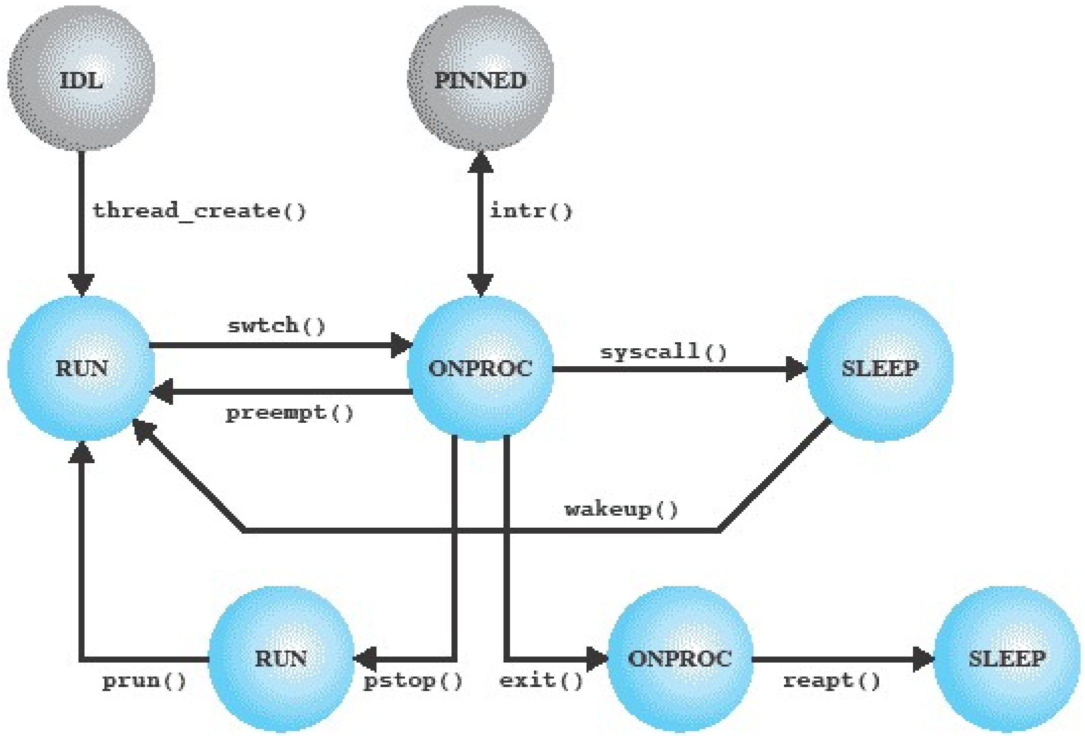


Figure 4.15 Solaris Thread States

Interrupts as Threads

- Most operating systems contain two fundamental forms of concurrent activity:

Processes (threads)

- cooperate with each other and manage the use of shared data structures by primitives that enforce mutual exclusion and synchronize their execution

Interrupts

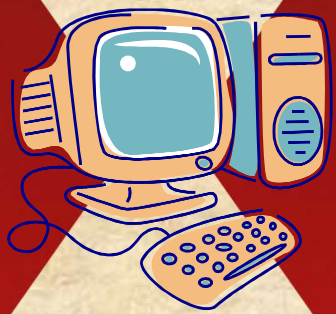
- synchronized by preventing their handling for a period of time

Solaris Solution

- Solaris employs a set of kernel threads to handle interrupts
  - an interrupt thread has its own identifier, priority, context, and stack
  - the kernel controls access to data structures and synchronizes among interrupt threads using mutual exclusion primitives
  - interrupt threads are assigned higher priorities than all other types of kernel threads

Linux Tasks

A process, or task, in Linux is represented by a task_struct data structure



This structure contains information in a number of categories

Linux Process/Thread Model

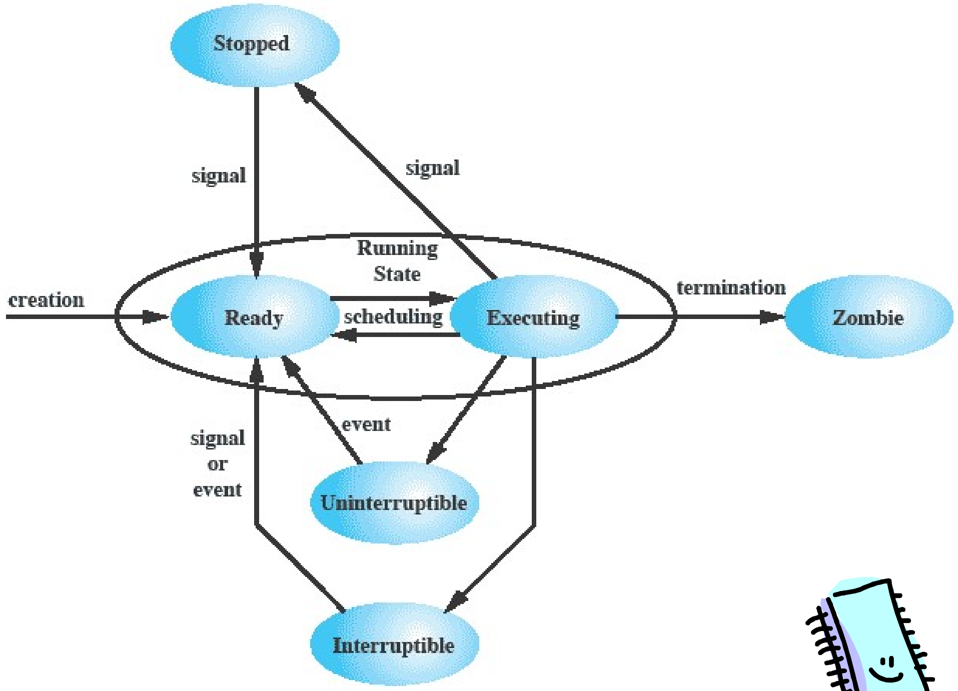


Figure 4.16 Linux Process/Thread Model

Linux Threads

Linux does not recognize a distinction between threads and processes

User-level threads are mapped into kernel-level processes

A new process is created by copying the attributes of the current process

The new process can be *cloned* so that it shares resources

The clone() call creates separate stack spaces for each process

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 *execve()* system call.
CLONE_VM	Shares the address space (memory descriptor and all page tables).