Operating Systems: Internals and Design Principles

Chapter 4 Threads

Ninth Edition

By William Stallings

Process 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 the entity that is scheduled and dispatched by the OS

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*** **-** The ability of an OS to support multiple, concurrent paths of execution within a single process

Single Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- MS-DOS is an example

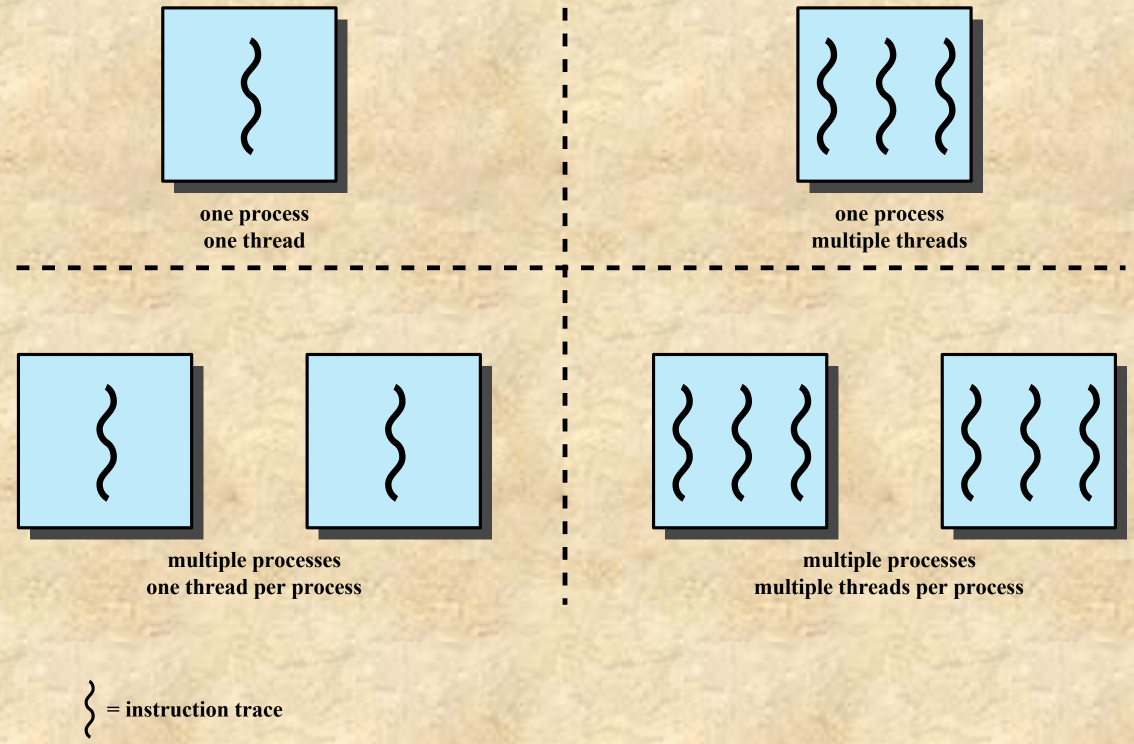


Figure 4.1 Threads and Processes

Multithreaded Approaches

- The right half of Figure 4.1 depicts multithreaded approaches
- A Java run-time environment is an example of a system of one process with multiple threads

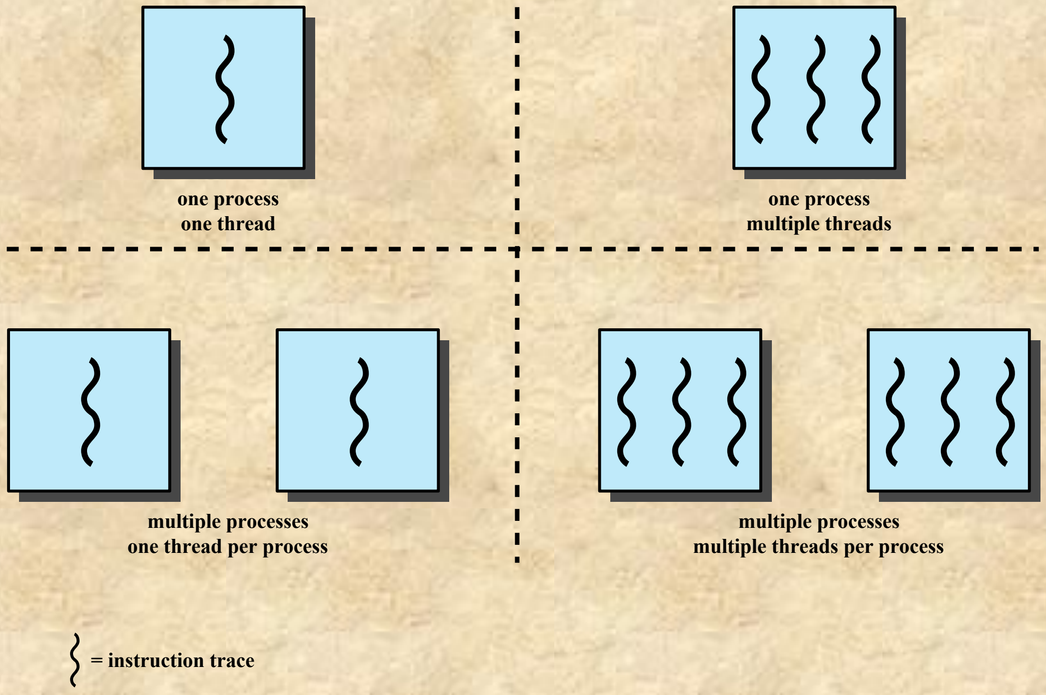


Figure 4.1 Threads and Processes

Process

- Defined in a multithreaded environment as "the unit of resource allocation and a unit of protection"
- Associated with processes:
  - A virtual address space that holds the process image
  - Protected access to:
    - Processors
    - Other processes (for interprocess communication)
    - Files
    - I/O resources (devices and channels)

One or More Threads in a Process

Each thread has:

- An execution state (Running, Ready, etc.)
- A 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 processes, shared with all other threads in that process

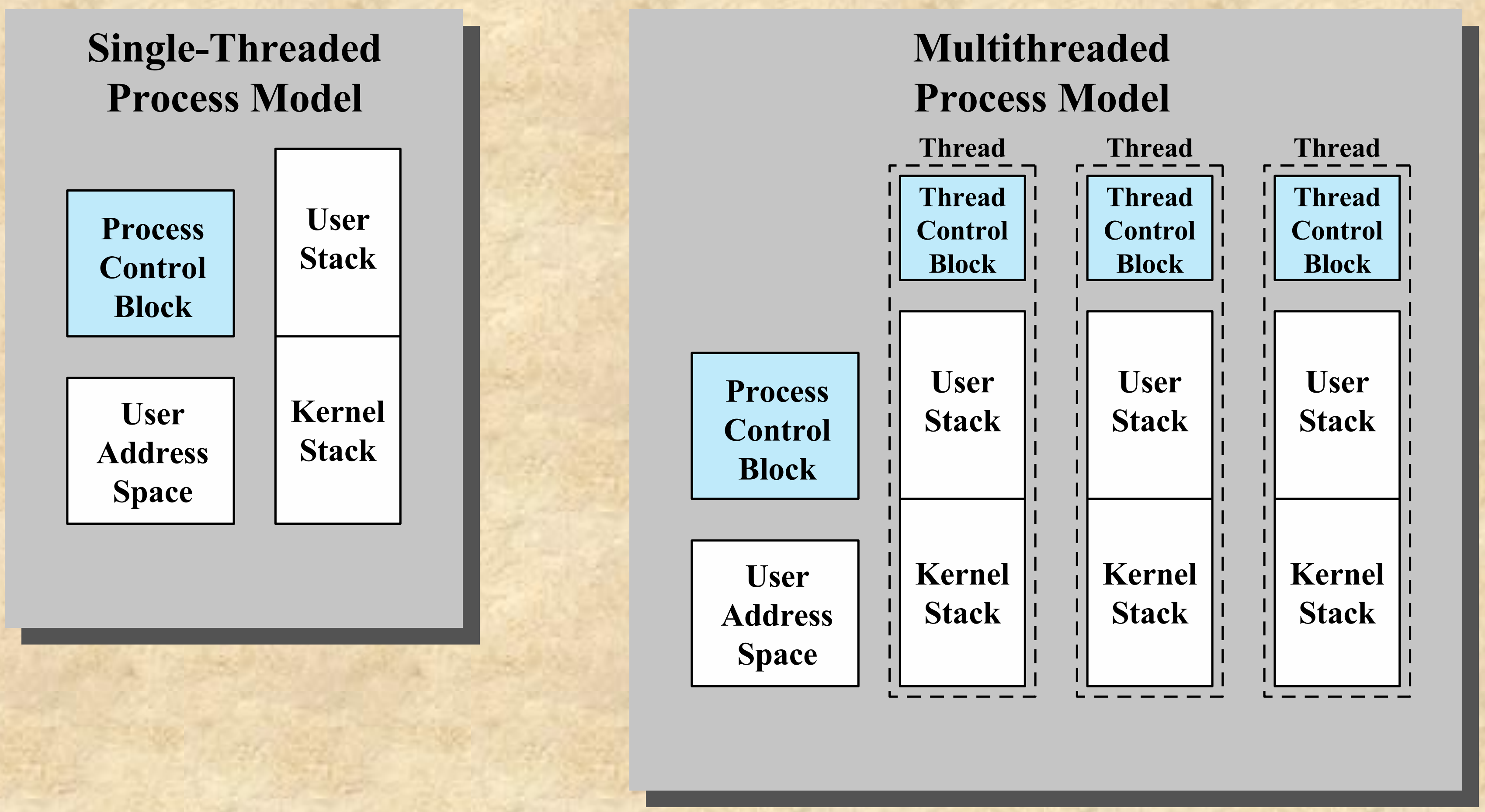
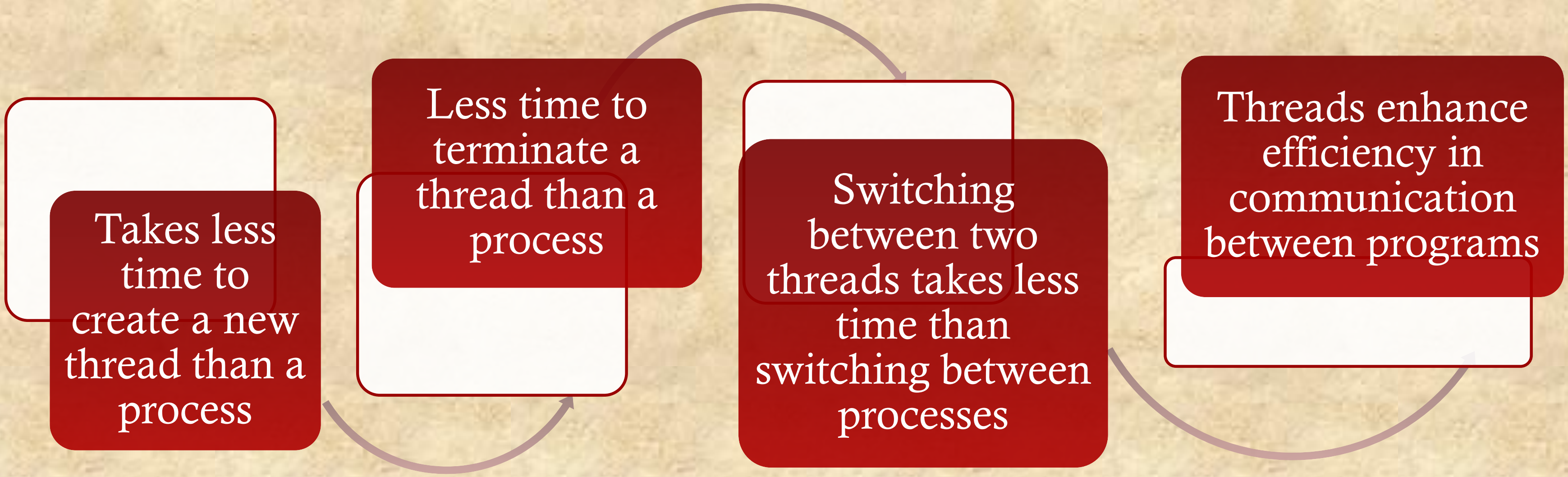


Figure 4.2 Single Threaded and Multithreaded Process Models

Key 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 (one thread could display menus and read user input, while another thread executes user commands)
- Asynchronous processing (e.g. periodic backup)
- Speed of execution (multiple threads from the same process may be able to execute simultaneously)
- Modular program structure (Programs that involve a variety of activities or a variety of sources and destinations of input and output may be easier to design and implement using threads.)

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

- There are, however, several actions that affect all of the threads in a process and that 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

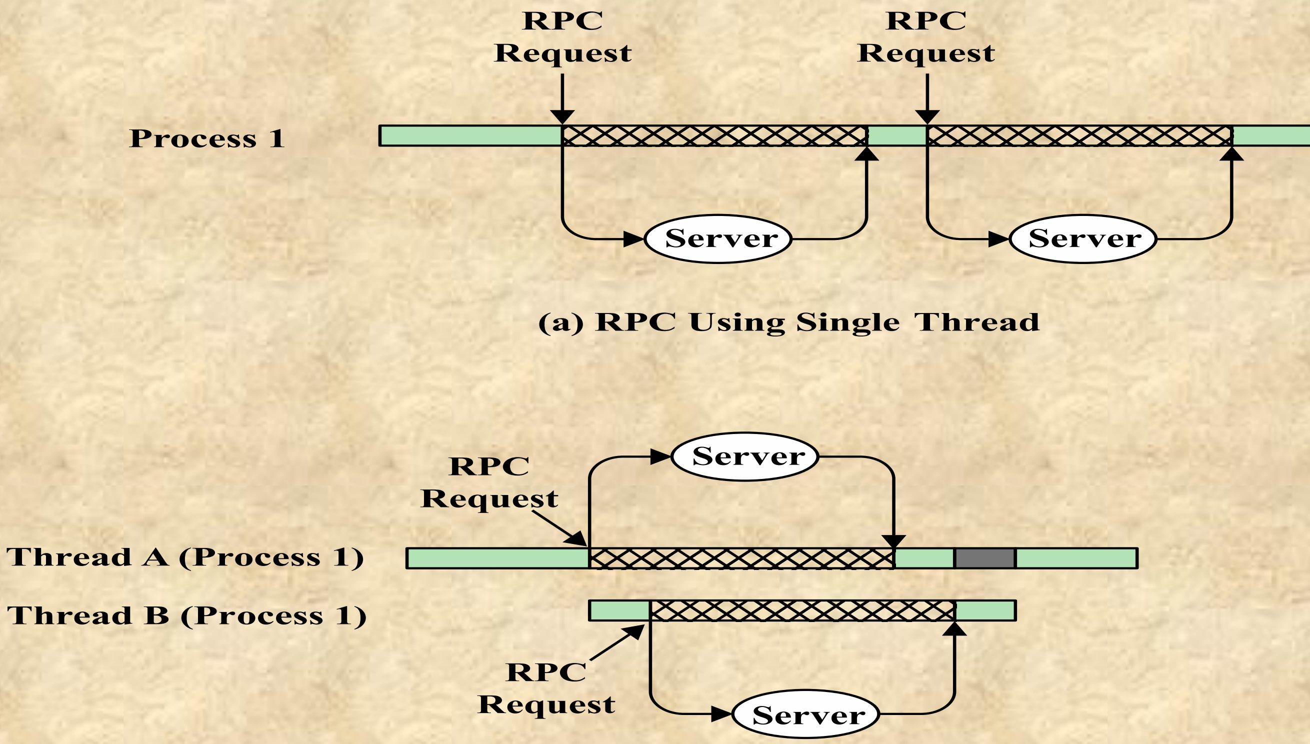
Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish



▨ Blocked, waiting for response to RPC

⬛ Blocked, waiting for processor, which is in use by Thread B

🟩 Running

Figure 4.3 Remote Procedure Call (RPC) Using Threads

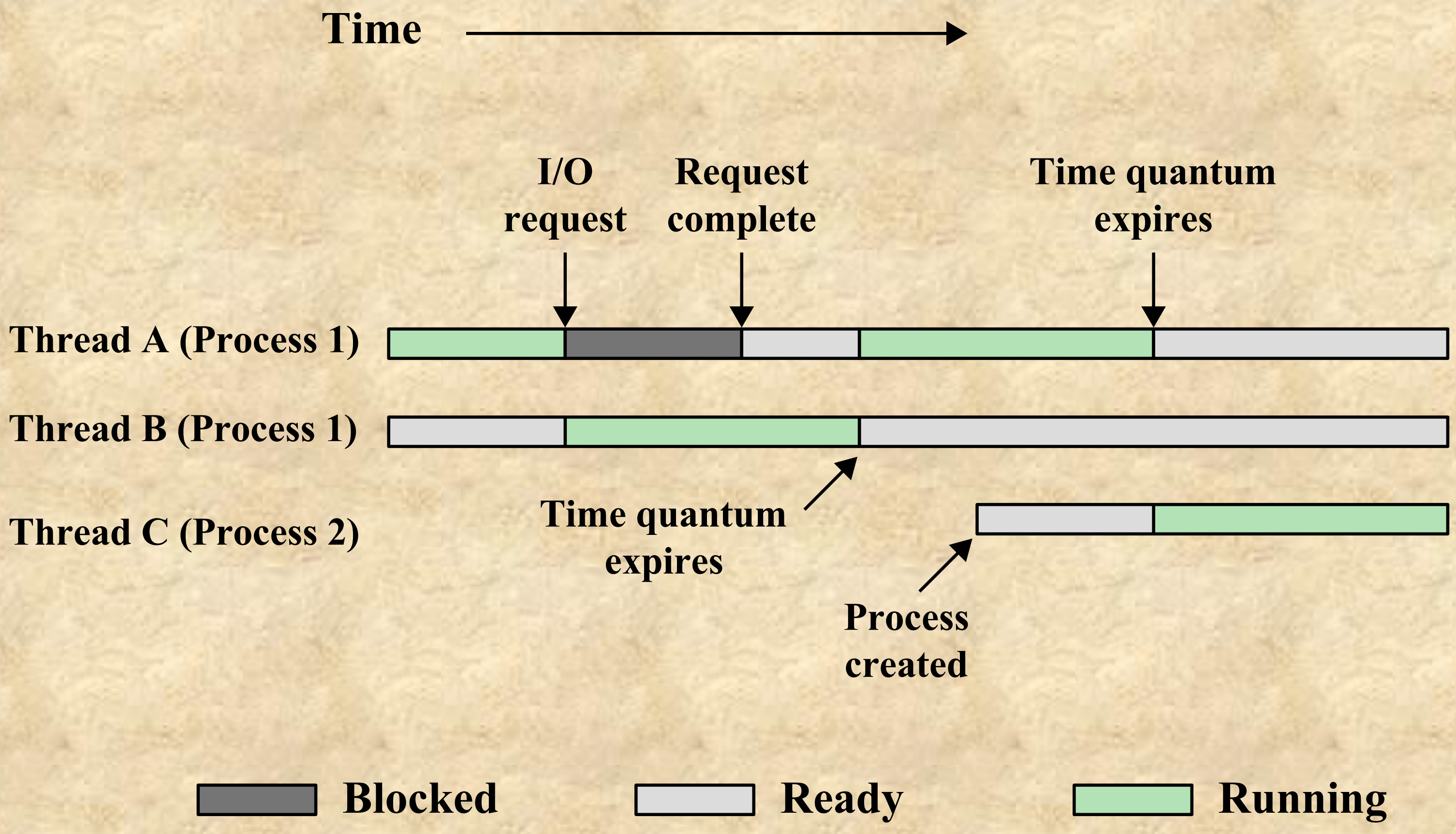


Figure 4.4 Multithreading Example on a Uniprocessor

Thread Synchronization

- It is necessary to synchronize the activities of the various threads
  - 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

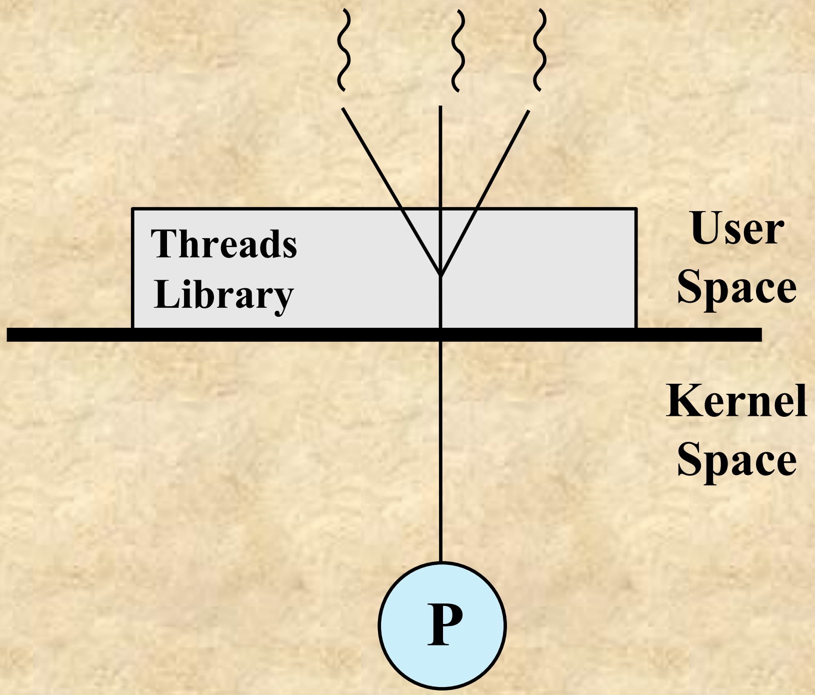
Types of Threads

User Level Thread (ULT)

Kernel level Thread (KLT)

User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level

The kernel is unaware of thread activities and continues to schedule the process as a unit and assigns a single execution state (Ready, Running, Blocked, etc.) to that process.

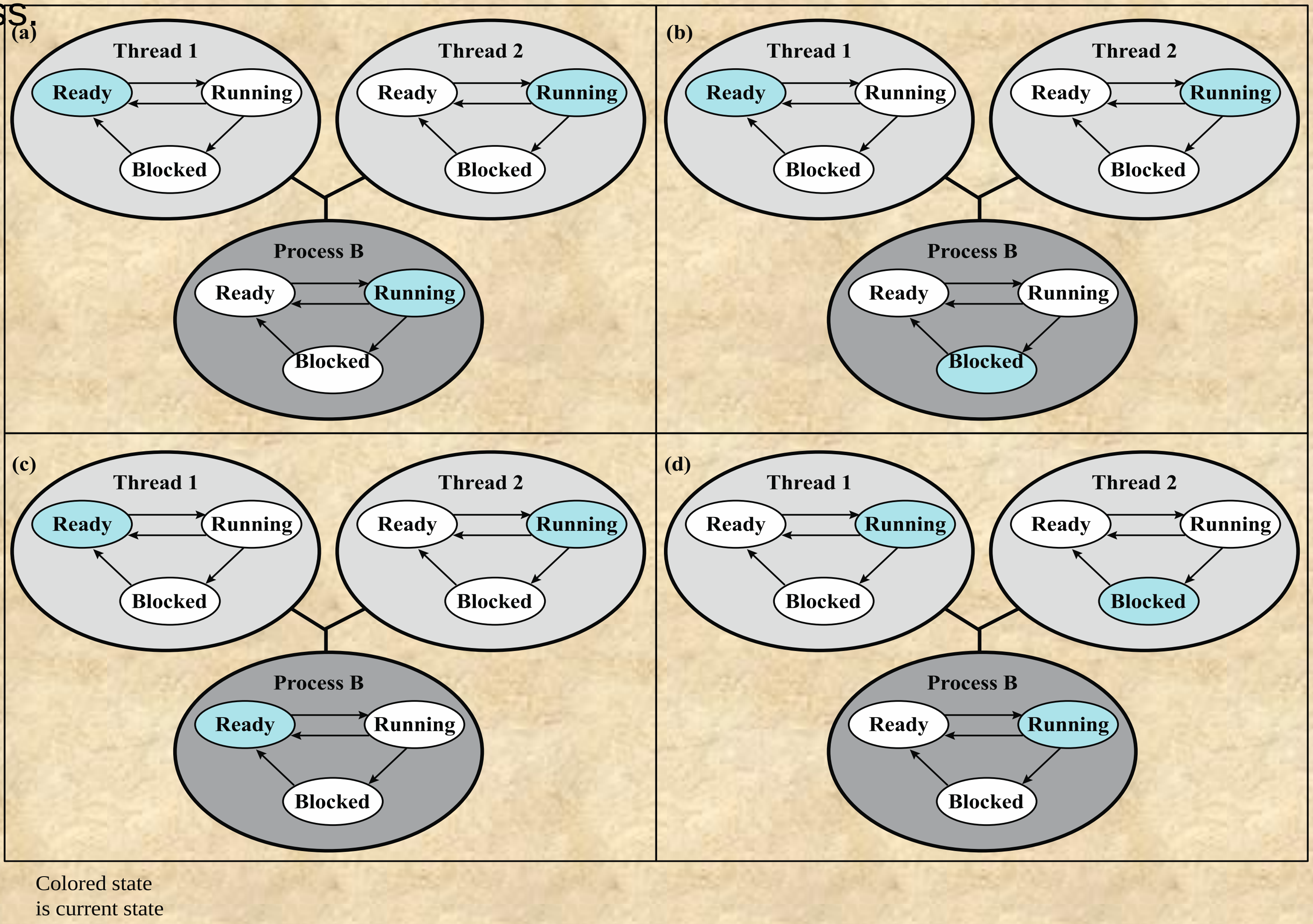
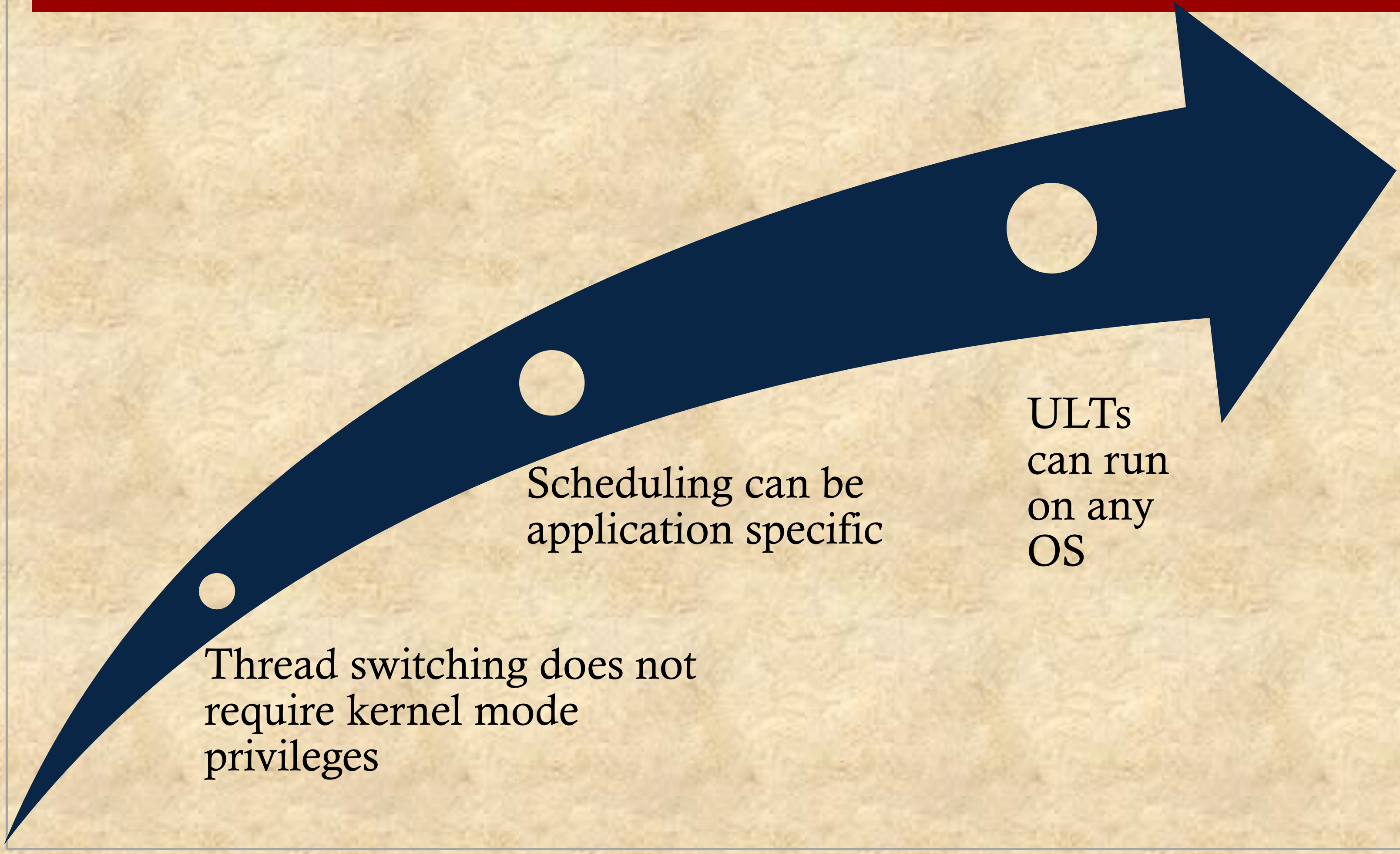


Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

Advantages of ULTs



Thread switching does not require kernel mode privileges

Scheduling can be application specific

ULTs can run on any OS

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 as well
- 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

Overcoming ULT Disadvantages

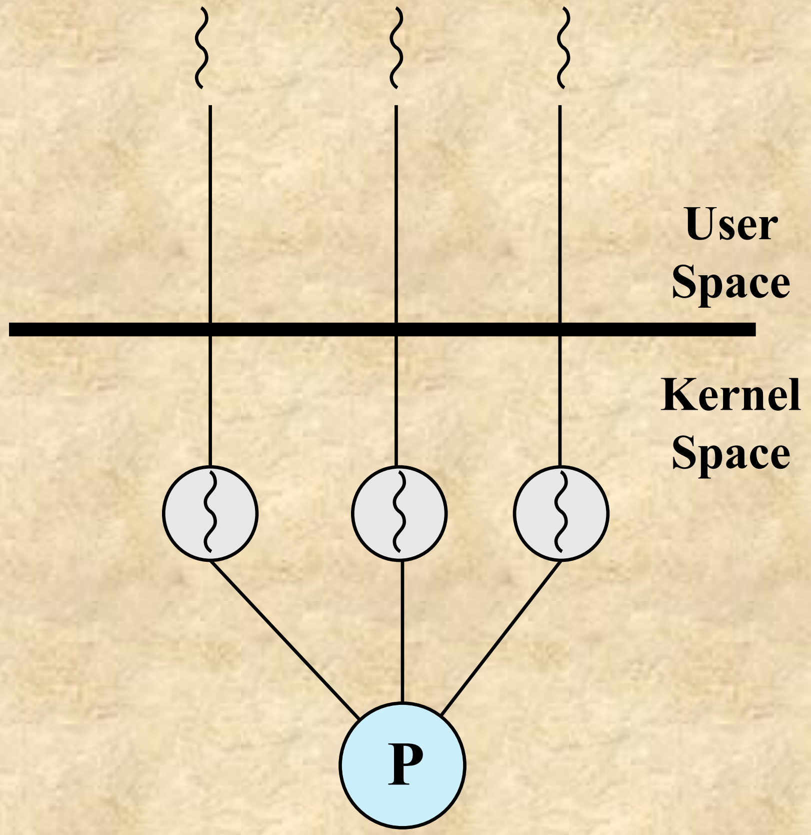
Jacketing

- Purpose is to convert a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads

- However, this approach eliminates the main advantage of threads

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the kernel
  - There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
  - 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
- Kernel routines themselves can be multithreaded

Disadvantage of KLTs

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

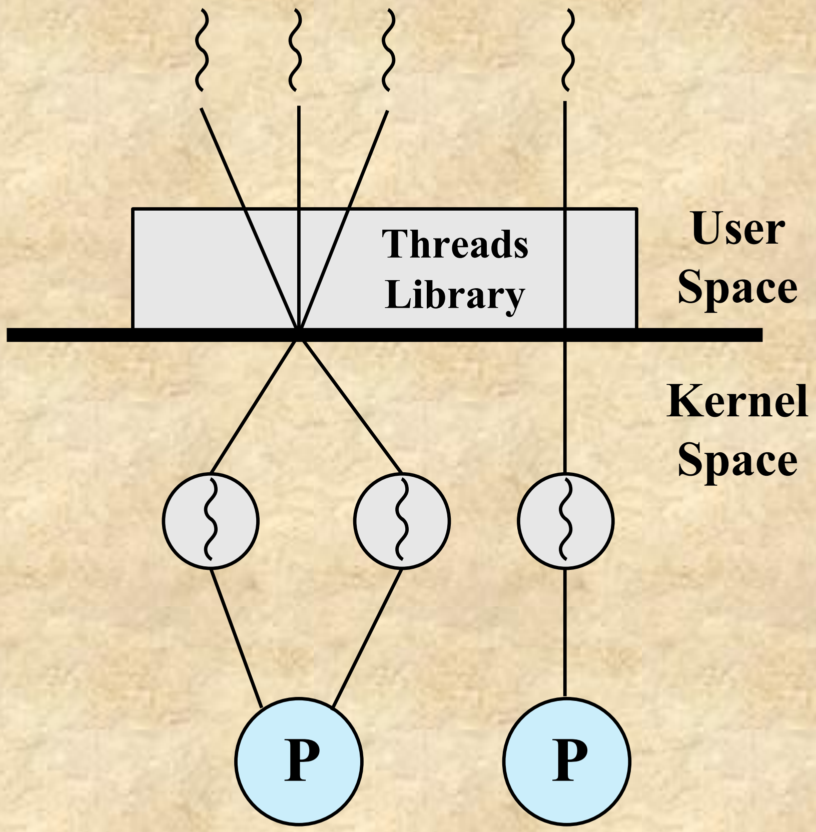
Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1
Thread and Process Operation Latencies (μs)

The principal disadvantage of the KLT approach compared to the ULT approach is that the transfer of control from one thread to another within the same process requires a mode switch to the kernel. To illustrate the differences, Table 4.1 shows the results of measurements taken on a uniprocessor VAX computer running a UNIX-like OS. The two benchmarks are as follows: Null Fork, the time to create, schedule, execute, and complete a process/thread that invokes the null procedure (i.e., the overhead of forking a process/thread); and Signal-Wait, the time for a process/thread to signal a waiting process/thread and then wait on a condition (i.e., the overhead of synchronizing two processes/threads together). We see that there is an order of magnitude or more of difference between ULTs and KLTs and similarly between KLTs and processes.

Combined Approaches

- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application
- Solaris is a good example



(c) Combined

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

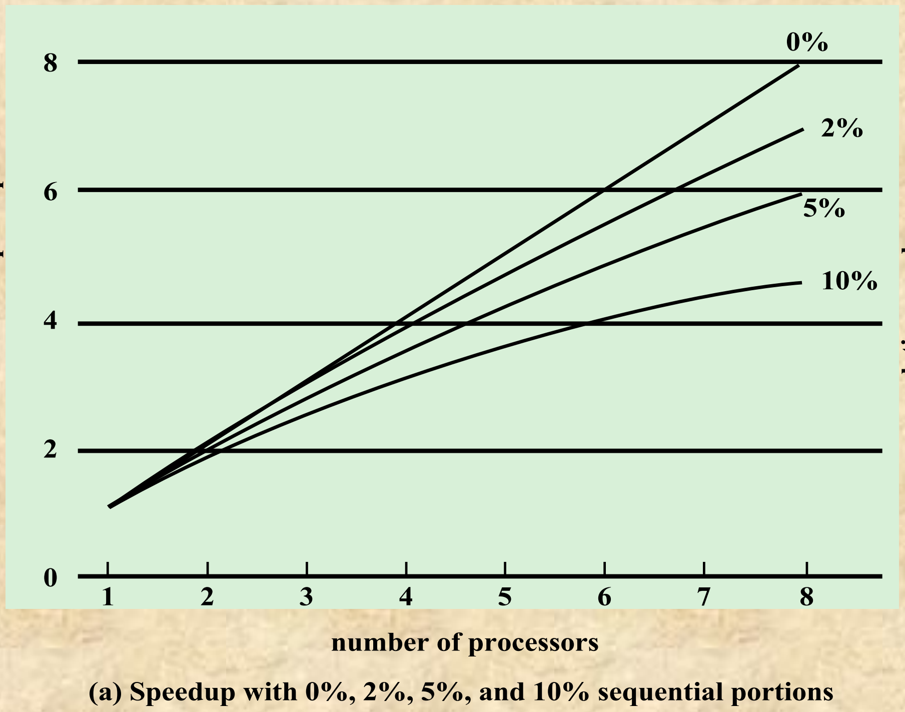
Table 4.2
Relationship between Threads and Processes

Amdahl's law

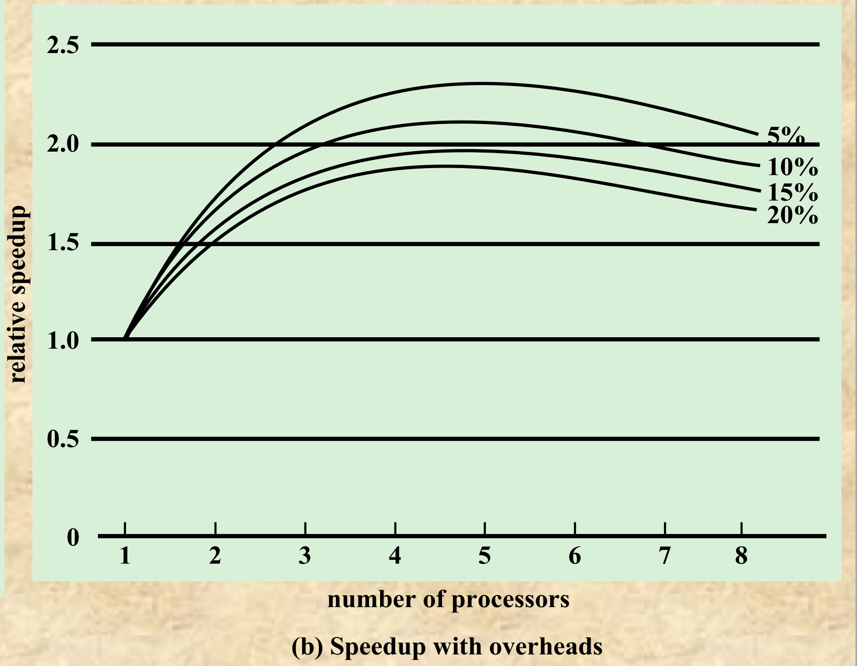
- The potential performance benefits of a multicore organization depend on the ability to effectively exploit the parallel resources available to the application.
- Let us focus first on a single application running on a multicore system. Amdahl's law (see Appendix E) states that:
- Speedup = time to execute program on a single processor / time to execute program on *N parallel processors*

$$= 1 / (1 - f) + f / N$$

- The law assumes a program in which a fraction (1 - *f*) *of the execution time* involves code that is inherently serial and a fraction *f that involves code that is infinitely* parallelizable with no scheduling overhead.



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



(b) Speedup with overheads

Figure 4.7 Performance Effect of Multiple Cores

Even a small amount of serial code has a noticeable impact. If only 10% of the code is inherently serial (*f = 0.9)* , *running the program on a* multicore system with eight processors yields a performance gain of only a factor of 4.7.

In addition, software typically incurs overhead as a result of communication and distribution of work to multiple processors and cache coherence overhead. This results in a curve where performance peaks and then begins to degrade because of the increased burden of the overhead of using multiple processors. Figure 4.7b , from [MCDO07], is a representative example.

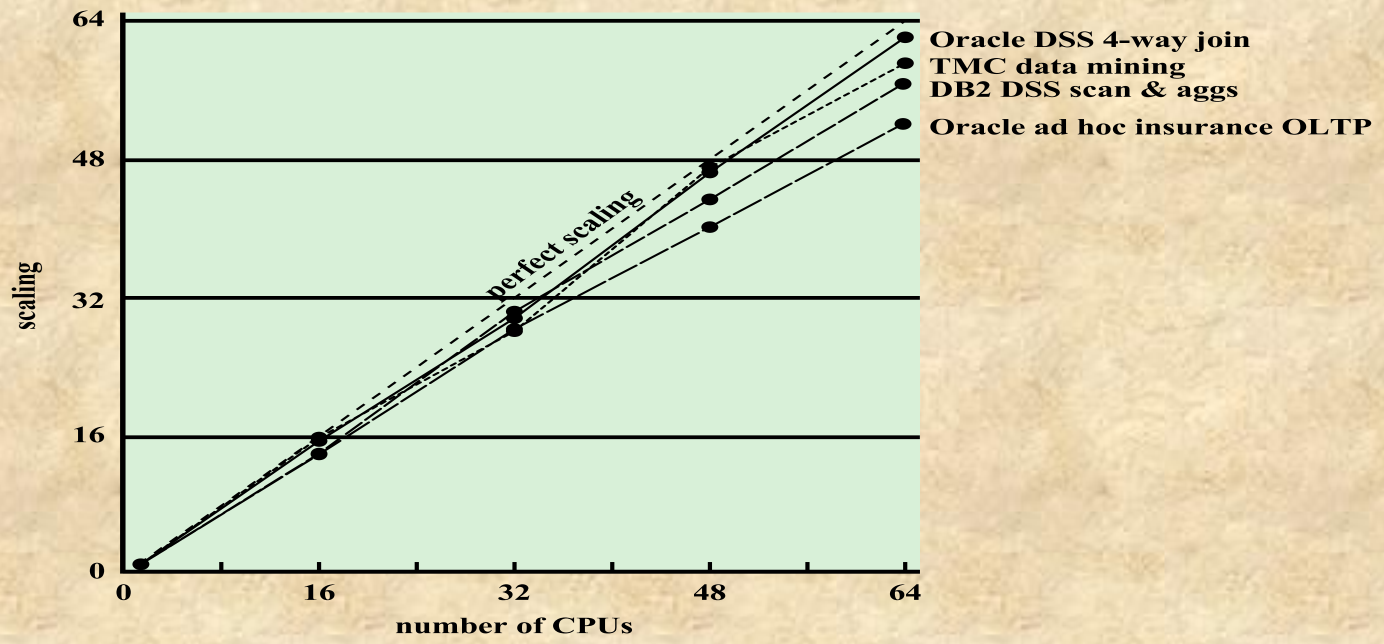


Figure 4.8 Scaling of Database Workloads on Multiple-Processor Hardware

However, software engineers have been addressing this problem and there are numerous applications in which it is possible to effectively exploit a multicore system. [MCDO07] reports on a set of database applications, in which great attention was paid to reducing the serial fraction within hardware architectures, operating systems, middleware, and the database application software. Figure 4.8 shows the result. As this example shows, database management systems and database applications are one area in which multicore systems can be used effectively. Many kinds of servers can also effectively use the parallel multicore organization, because servers typically handle numerous relatively independent transactions in parallel.

Applications That Benefit

- Multithreaded native applications
  - Characterized by having a small number of highly threaded processes
- Multiprocess applications
- Characterized by the presence of many single-threaded processes (Examples of multiprocess applica- tions include the Oracle database, SAP, and PeopleSoft)
- Java applications
  - All applications that use a Java 2 Platform, Enterprise Edition application server can immediately benefit from multicore technology
- Multi-instance applications
  - Multiple instances of the application in parallel

Application Example: Valve Game Software

- Valve is an entertainment and technology company that has developed a number of popular games, as well as the Source engine, one of the most widely played game engines available. Source is an animation engine used by Valve for its games and licensed for other game developers.
- In recent years, Valve has reprogrammed the Source engine software to use multithreading to exploit the power of multicore processor chips from Intel and AMD [REIM06].
- The revised Source engine code provides more powerful support for Valve games such as Half Life 2.

Valve Game Software

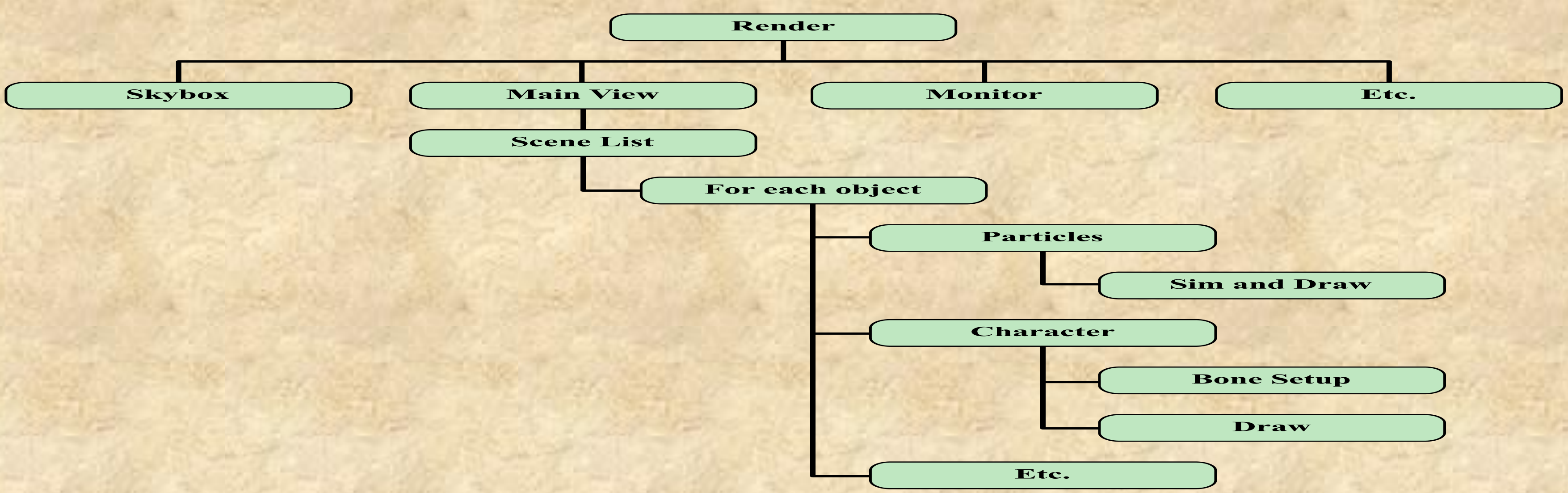


Figure 4.9 Hybrid Threading for Rendering Module

Figure 4.9 illustrates the thread structure for the rendering module. In this hierarchical structure, higher-level threads spawn lower-level threads as needed. The rendering module relies on a critical part of the Source engine, the world list, which is a database representation of the visual elements in the game's world.

Windows Process and Thread Management

- An **application** consists of one or more processes
- Each **process** provides the resources needed to execute a program
- A **thread** is the entity within a process that can be scheduled for execution
- A **job object** allows groups of process to be managed as a unit
- A **thread pool** is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application
- A **fiber** is a unit of execution that must be manually scheduled by the application
- **User-mode scheduling (UMS)** is a lightweight mechanism that applications can use to schedule their own threads

Management of Background Tasks and Application Lifecycles

- Beginning with Windows 8, and carrying through to Windows 10, developers are responsible for managing the state of their individual applications
- Previous versions of Windows always give the user full control of the lifetime of a process
- In the new Metro interface Windows takes over the process lifecycle of an application
  - A limited number of applications can run alongside the main app in the Metro UI using the SnapView functionality
  - Only one Store application can run at one time
- Live Tiles give the appearance of applications constantly running on the system
  - In reality they receive push notifications and do not use system resources to display the dynamic content offered

Metro Interface

- Foreground application in the Metro interface has access to all of the processor, network, and disk resources available to the user
    - All other apps are suspended and have no access to these resources
- When an app enters a suspended mode, an event should be triggered to store the state of the user’s information
    - This is the responsibility of the application developer
- Windows may terminate a background app
    - You need to save your app’s state when it’s suspended, in case Windows terminates it so that you can restore its state later
    - When the app returns to the foreground another event is triggered to obtain the user state from memory

Windows Process

Important characteristics of Windows processes are:

- Windows processes are implemented as objects
- A process can be created as a new process or a copy of an existing process
- An executable process may contain one or more threads
- Both process and thread objects have built-in synchronization capabilities

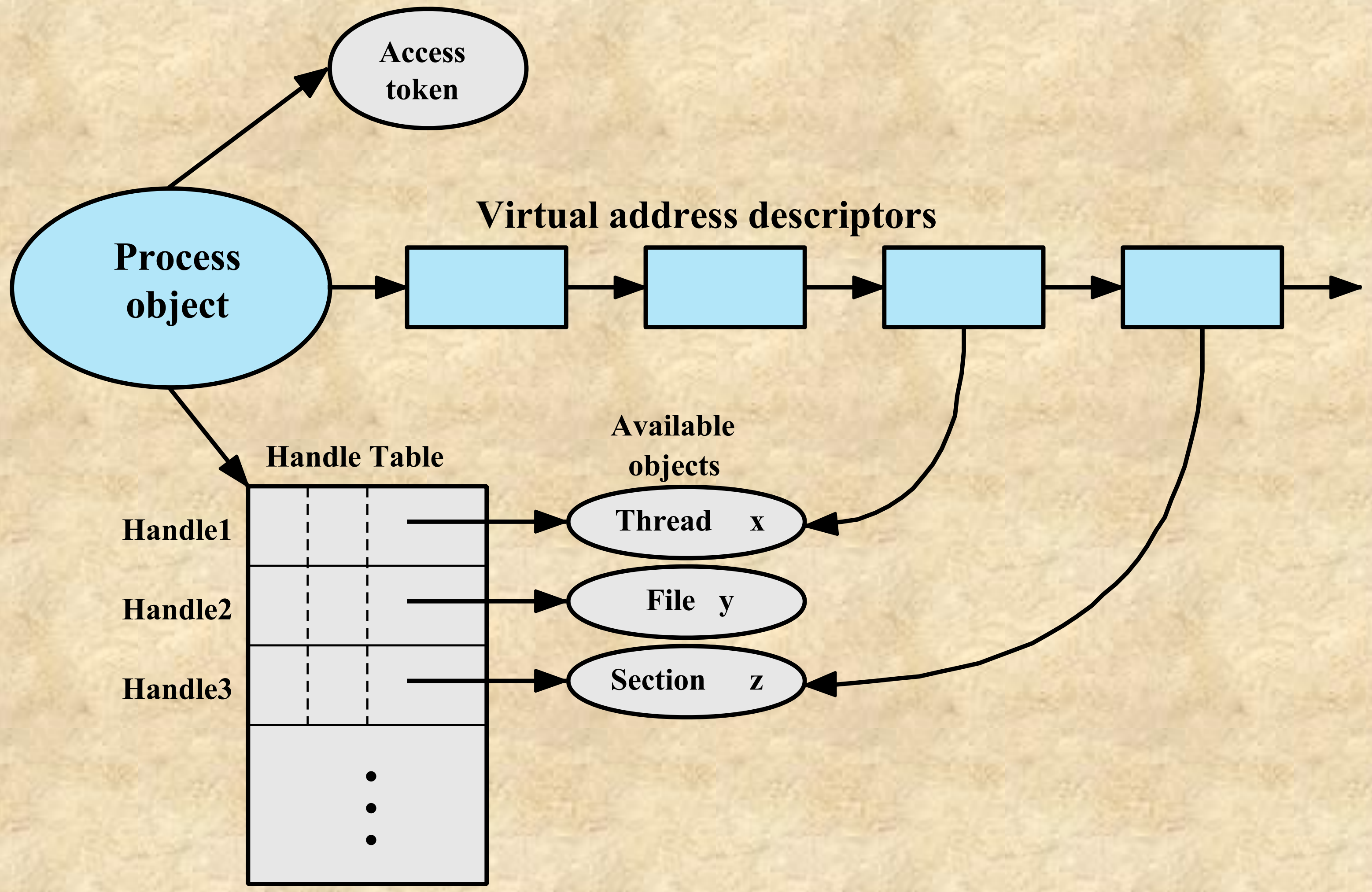


Figure 4.10 A Windows Process and Its Resources

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

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

(Table is on page 171 in textbook)

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

(Table is on page 171 in textbook)

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

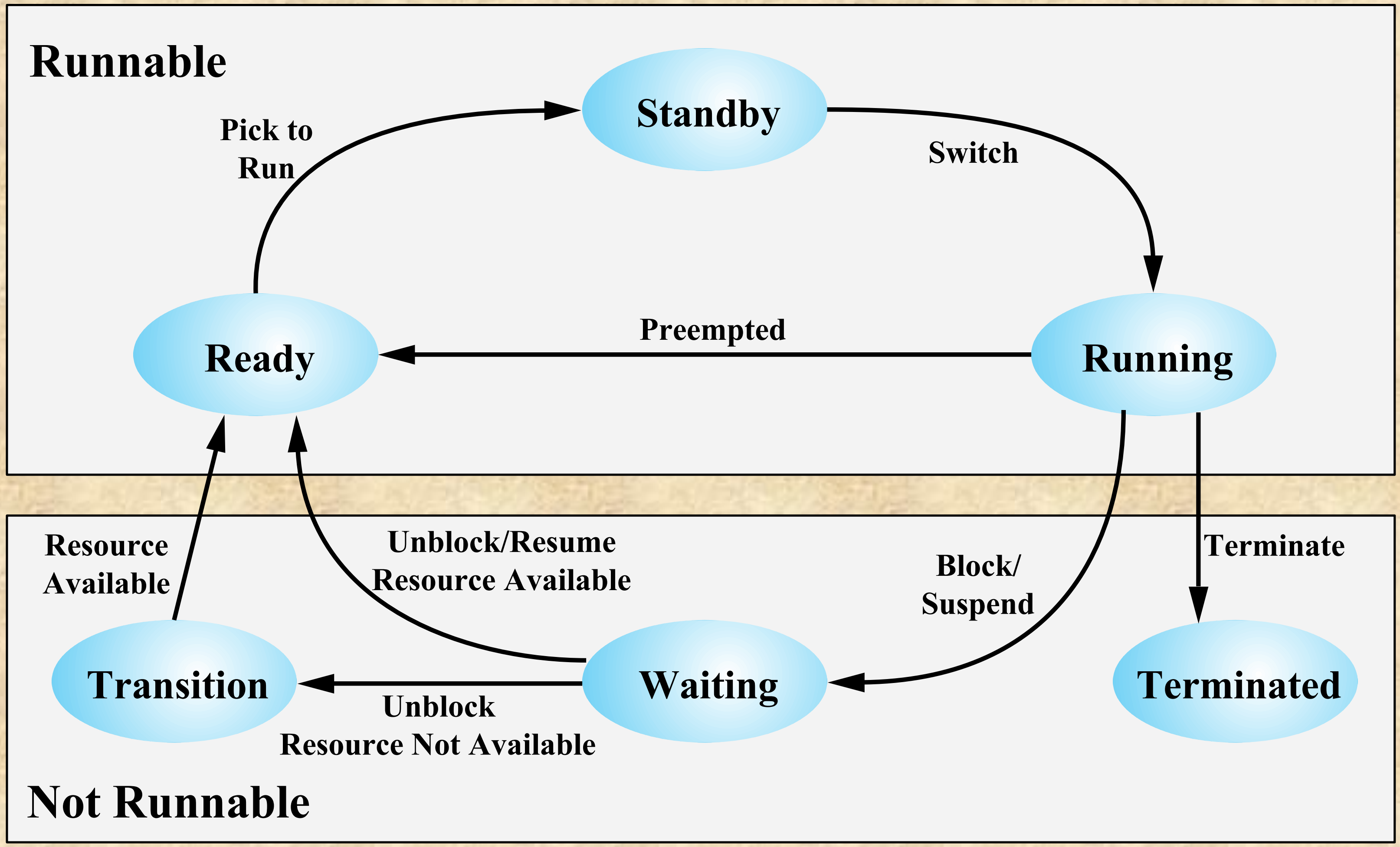


Figure 4.11 Windows Thread States

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

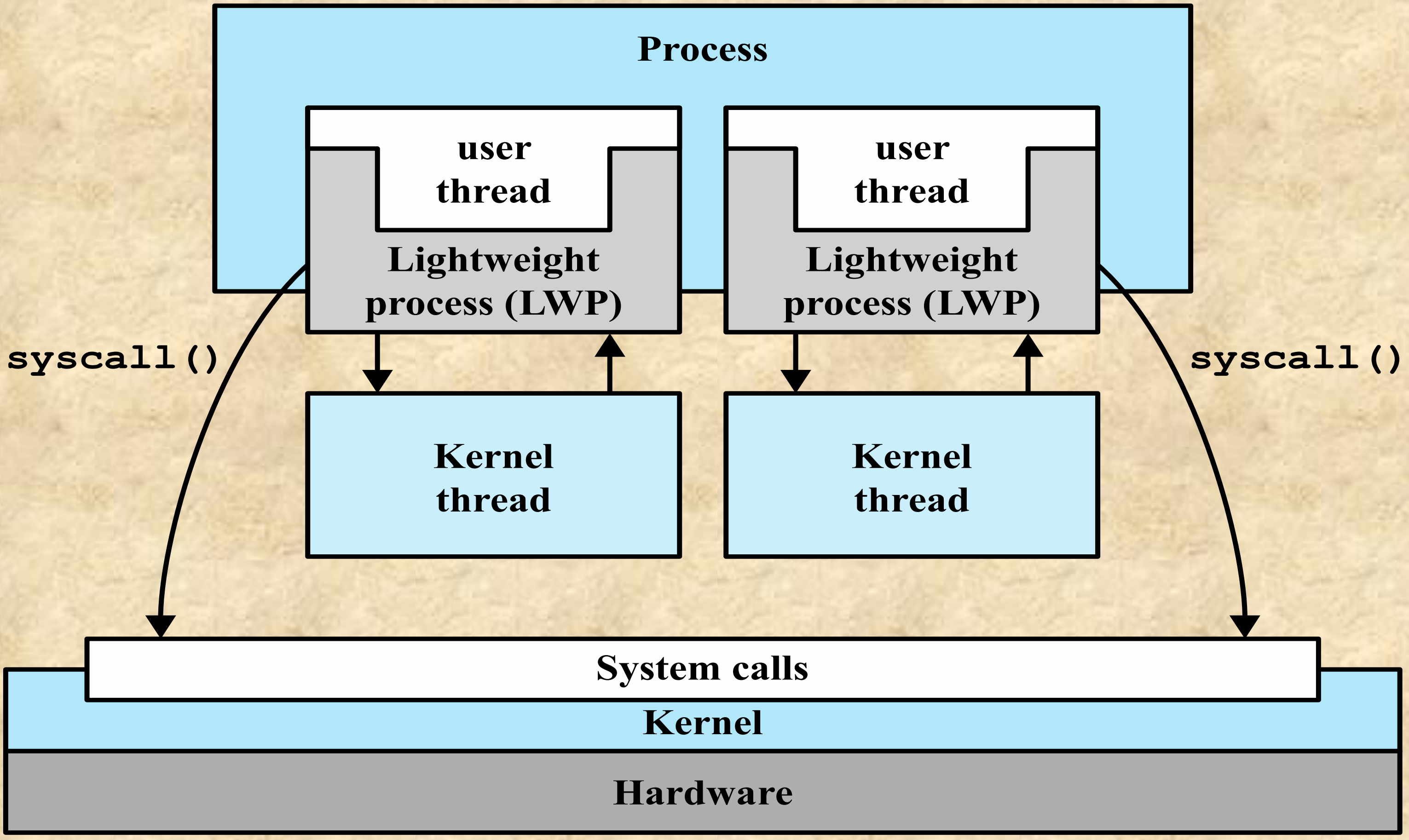


Figure 4.12 Processes and Threads in Solaris

Figure 4.12 illustrates the relationship among these four entities.

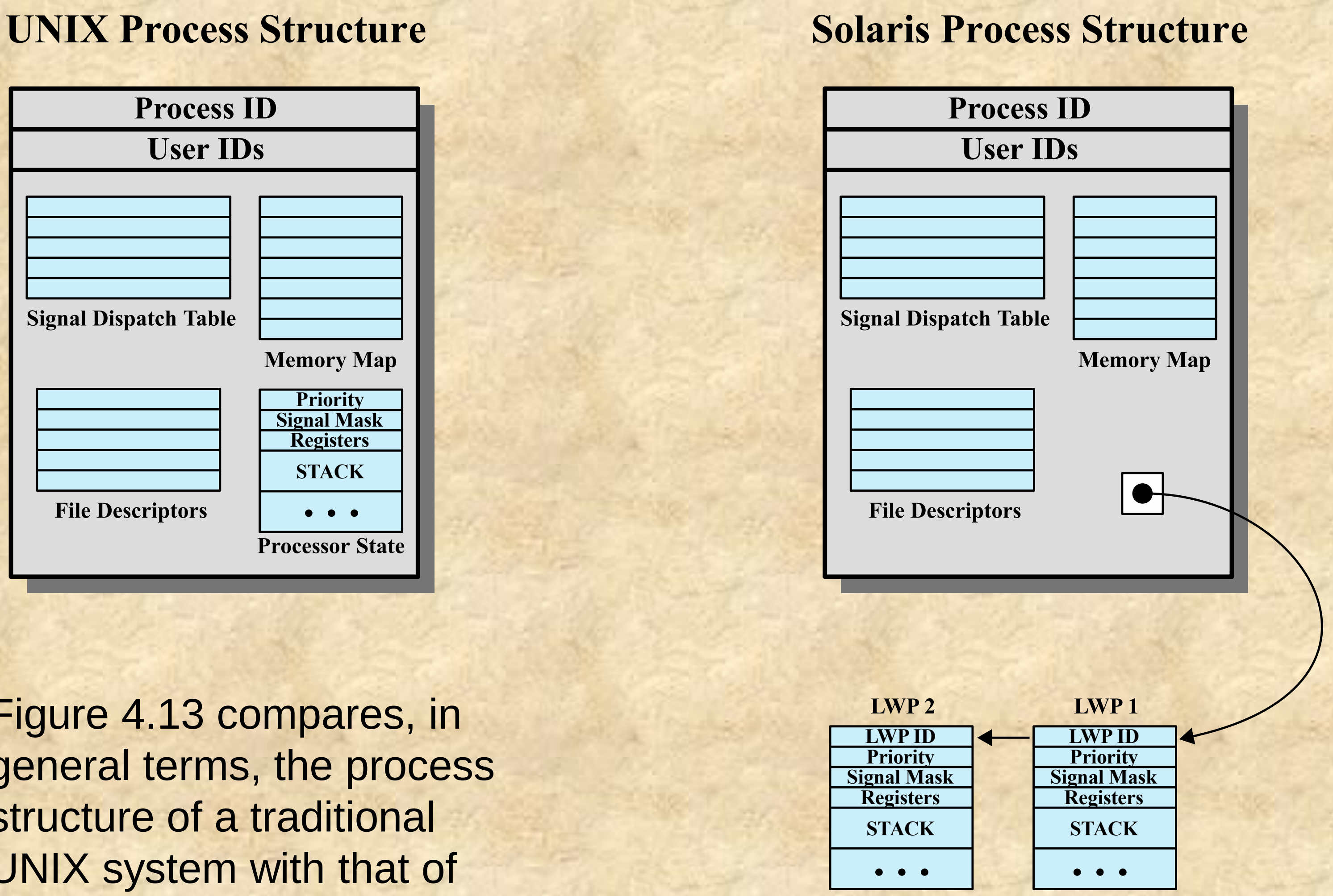


Figure 4.13 compares, in general terms, the process structure of a traditional UNIX system with that of Solaris

Figure 4.13 Process Structure in Traditional UNIX and Solaris [LEWI96]

A Lightweight Process (LWP) Data Structure Includes:

- An LWP identifier
- The priority of this LWP and hence the kernel thread that supports it
- A signal mask that tells the kernel which signals will be accepted
- Saved values of user-level registers
- The kernel stack for this LWP, which includes system call arguments, results, and error codes for each call level
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

- RUN: The thread is runnable; that is, the thread is ready to execute.
- ONPROC: The thread is executing on a processor.
- SLEEP: The thread is blocked.
- STOP: The thread is stopped.
- ZOMBIE: The thread has terminated.
- FREE: Thread resources have been released and the thread is awaiting removal from the OS thread data structure.

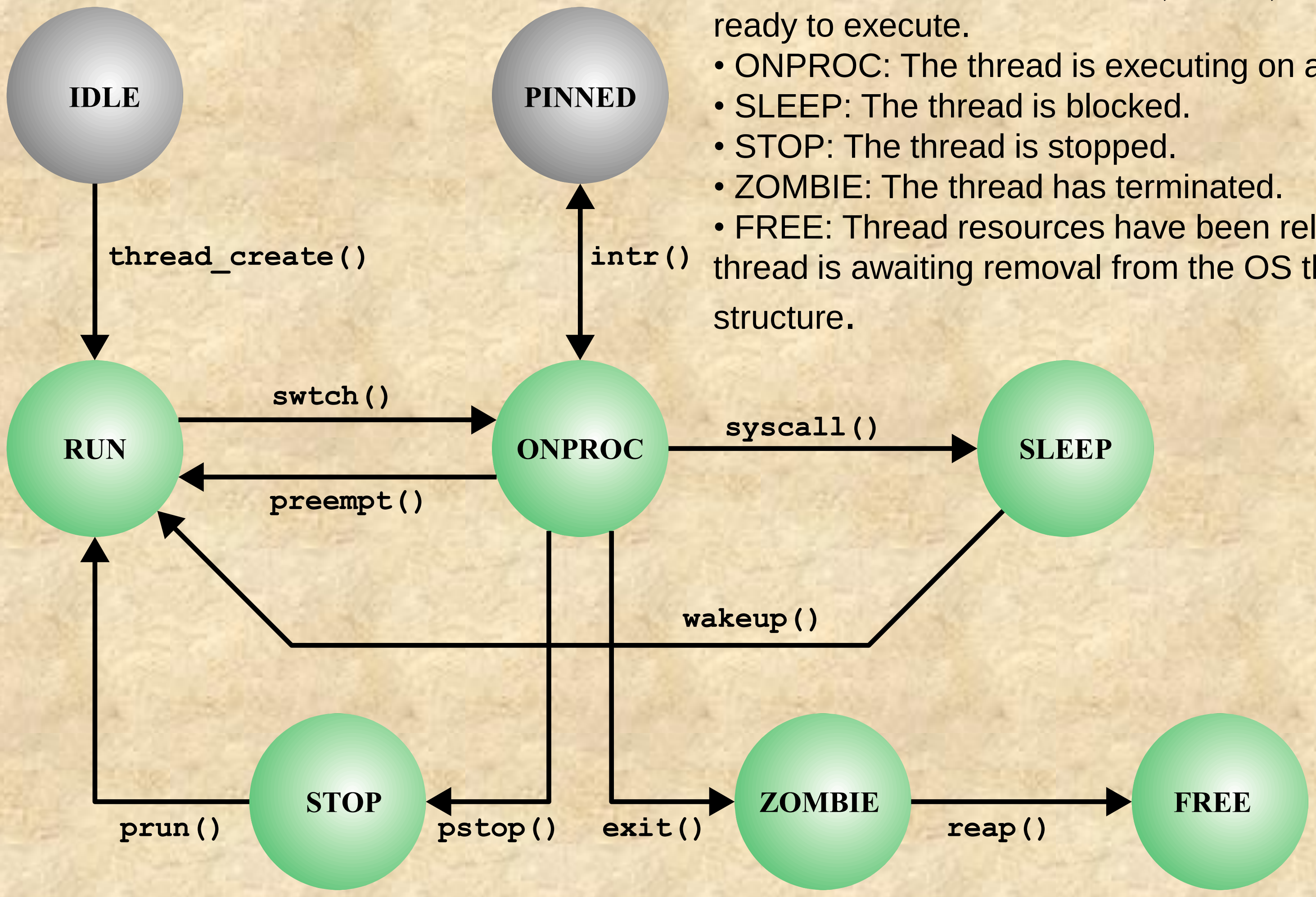


Figure 4.14 Solaris Thread States

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 unifies these two concepts into a single model, namely kernel threads, and the mechanisms for scheduling and executing kernel threads
  - To do this, interrupts are converted to kernel threads

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

task_struct data structure

• **State:**
The execution state of the process (executing, ready, suspended, stopped, zombie). This is described subsequently.
• **Scheduling information:**
Information needed by Linux to schedule processes. A process can be normal or real time and has a priority. Real-time processes are scheduled before normal processes, and within each category, relative priorities can be used. A counter keeps track of the amount of time a process is allowed to execute.

• **Identifiers:**
Each process has a unique process identifier and also has user and group identifiers. A group identifier is used to assign resource access privileges to a group of processes.

• **Interprocess communication:**
Linux supports the IPC mechanisms found in UNIX SVR4, described in Chapter 6.

task_struct data structure

• **Links:**
 Each process includes a link to its parent process, links to its siblings (processes with the same parent), and links to all of its children.

Times and timers:
Includes process creation time and the amount of processor time so far consumed by the process. A process may also have associated one or more interval timers.

• **File system:**
 Includes pointers to any files opened by this process, as well as pointers to the current and the root directories for this process.

• **Address space:**
 Defines the virtual address space assigned to this process.

• **Processor-specific context:**
The registers and stack information that constitute the context of this process.

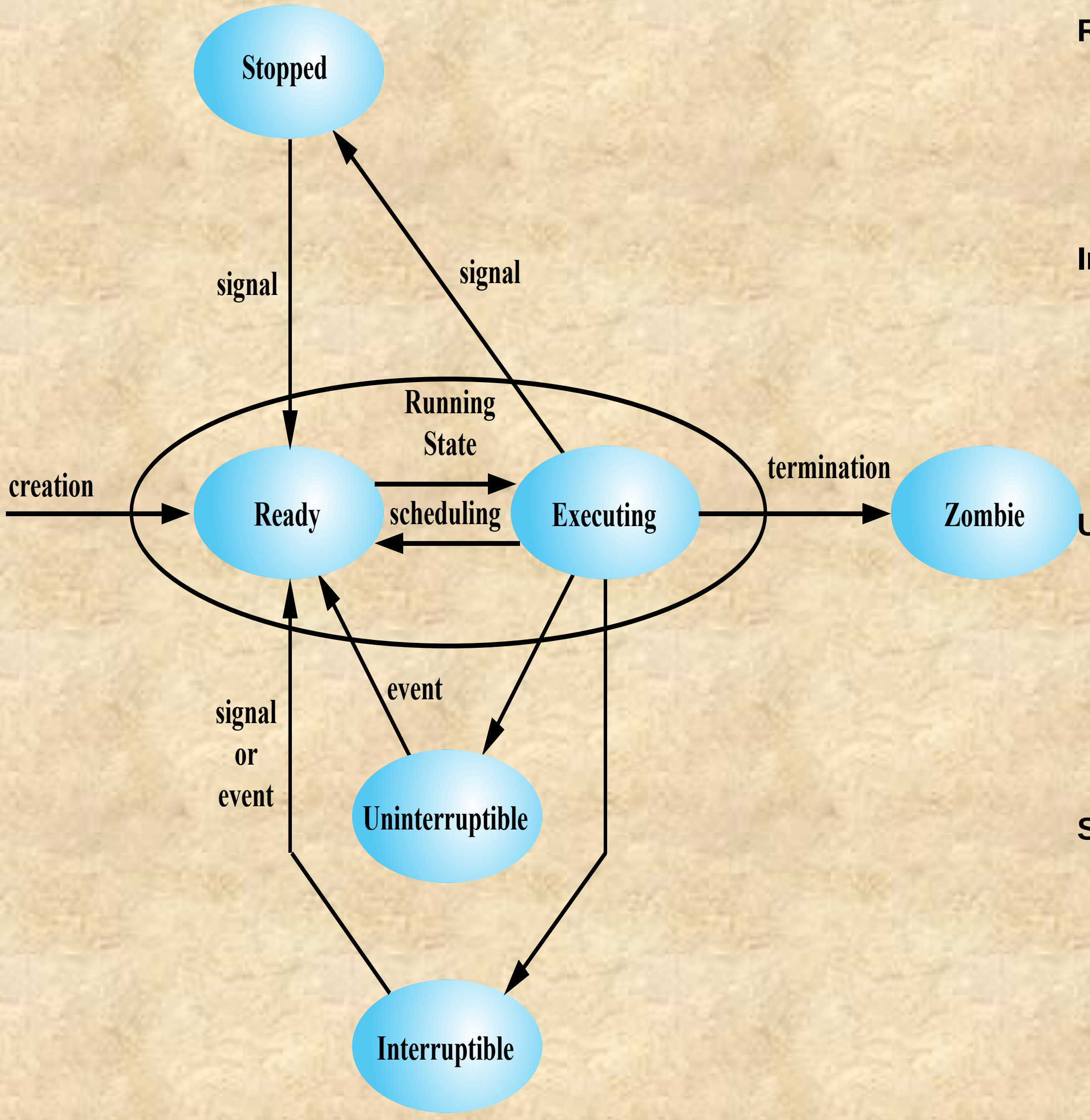


Figure 4.15 Linux Process/Thread Model

Running:
- Corresponds to two states.
  - -A Running process is either executing or
  - - it is ready to execute.

Interruptible:
- A blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.

Uninterruptible:
- Another blocked state.
- The difference between the Interruptible state is that in this state, a process is waiting directly on hardware conditions and therefore will not handle any signals.

Stopped:
- The process has been halted and can only resume by positive action from another process.
- E.G., a process that is being debugged can be put into the Stopped state.

Zombie:
- The process has been terminated but, for some reason, still must have its task structure in the process table.

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

Traditional UNIX systems support a single thread of execution per process, while modern UNIX systems typically provide support for multiple kernel-level threads per process. We have seen that modern versions of UNIX offer kernel-level threads. Linux provides a unique solution in that it does not recognize a distinction between threads and processes. Using a mechanism similar to the lightweight processes of Solaris, user-level threads are mapped into kernel-level processes. Multiple user-level threads that constitute a single user-level process are mapped into Linux kernel-level processes that share the same group ID. This enables these processes to share resources such as files and memory and to avoid the need for a context switch when the scheduler switches among processes in the same group

Linux Namespaces

- Associated with each process in Linux are a set of namespaces
- A namespace enables a process to have a different view of the system than other processes that have other associated namespaces
- There are currently six namespaces in Linux
  - Mnt (Mount namespace)
  - Pid (Process ID namespace)
  - Net (network namespace)
  - Ipc (Interprocess communication namespace)
  - Uts (Unix timesharing namespace)
  - User (User namespace)

Android Process and Thread Management

- An Android application is the software that implements an app
- Each Android application consists of one or more instance of one or more of four types of application components
- Each component performs a distinct role in the overall application behavior, and each component can be activated independently within the application and even by other applications
- Four types of components:
  - Activities
  - Services
  - Content providers
  - Broadcast receivers

Activitities

- Activities: An activity corresponds to a single screen visible as a user interface. For example, an e-mail application might have one activity that shows a list of new e-mails, another activity to compose an e-mail, and another activity for reading e-mails.
- Although the activities work together to form a cohesive user experience in the e-mail application, each one is independent of the others.
- Android makes a distinction between internal and exported activities. Other apps may start exported activities, which generally include the main screen of the app.
- However, other apps cannot start the internal activities. For example, a camera application can start the activity in the e-mail application that composes new mail, in order for the user to share a picture.

Services

- Services are typically used to perform background operations that take a considerable amount of time to finish. This ensures faster responsiveness, for the main thread (a.k.a. UI thread) of an application, with which the user is directly interacting.
- For example, a service might create a thread to play music in the background while the user is in a different application, or it might create a thread to fetch data over the network without blocking user interaction with an activity.
- A service may be invoked by an application. Additionally, there are system services that run for the entire lifetime of the Android system, such as Power Manager, Battery, and Vibrator services. These system services create threads that are part of the System Server process.
-

Content providers

- A content provider acts as an interface to application data that can be used by the application. One category of managed data is private data, which is used only by the application containing the content provider.
- For example the NotePad application uses a content provider to save notes. The other category is shared data, accessible by multiple applications.
- This category includes data stored in file systems, an SQLite database, on the Web, or any other persistent storage location your application can access

Broadcast receivers

- A broadcast receiver responds to system-wide broadcast announcements.
- A broadcast can originate from another application, such as to let other applications know that some data has been downloaded to the device and is available for them to use, or from the system (for example, a low-battery warning).

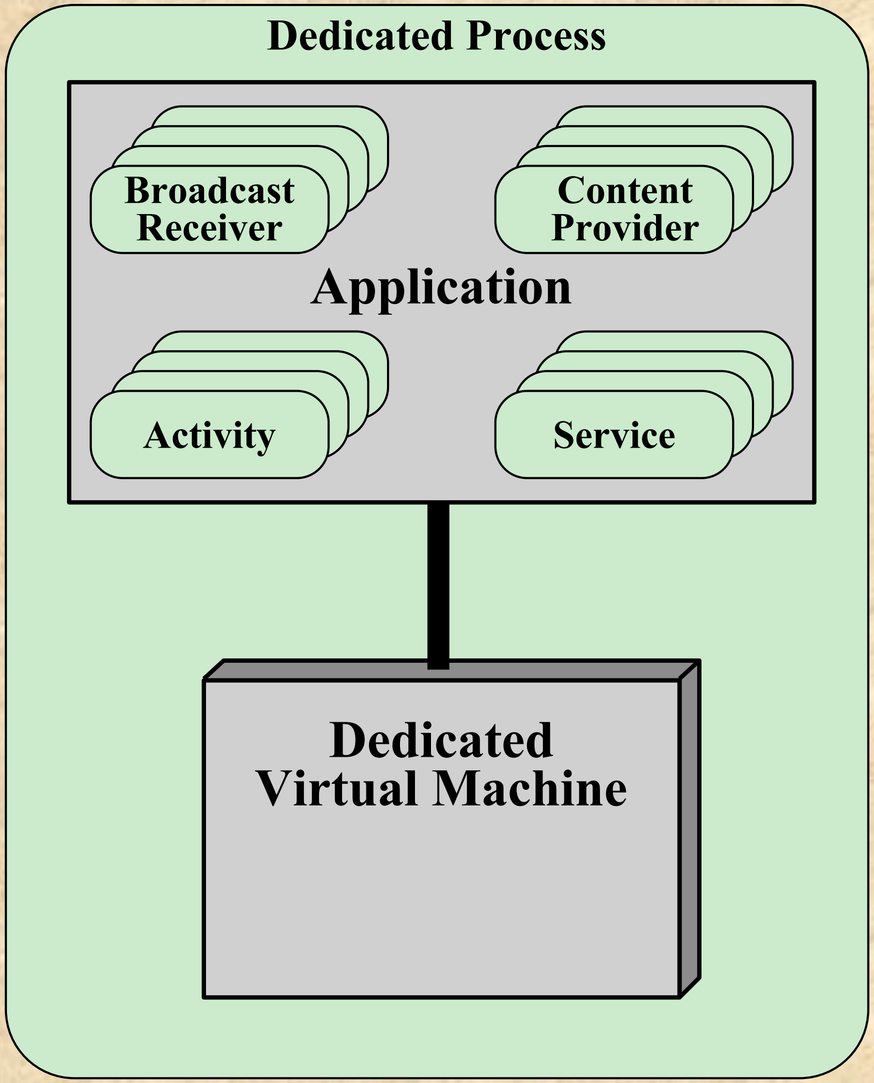


Figure 4.16 Android Application

- Each application runs on its own dedicated virtual machine and its own single process that encompasses the application and its virtual machine (Figure 4.16).
- This approach, referred to as the sandboxing model, isolates each application.
- Thus, one application cannot access the resources of the other without permission being granted.
- Each application is treated as a separate Linux user with its own unique user ID, which is used to set file permissions.

Activities

- An Activity is an application component that provides a screen with which users can interact in order to do something
- Each Activity is given a window in which to draw its user interface
- The window typically fills the screen, but may be smaller than the screen and float on top of other windows
- An application may include multiple activities
- When an application is running, one activity is in the foreground, and it is this activity that interacts with the user
- The activities are arranged in a last-in-first-out stack in the order in which each activity is opened
- If the user switches to some other activity within the application, the new activity is created and pushed on to the top of the back stack, while the preceding foreground activity becomes the second item on the stack for this application

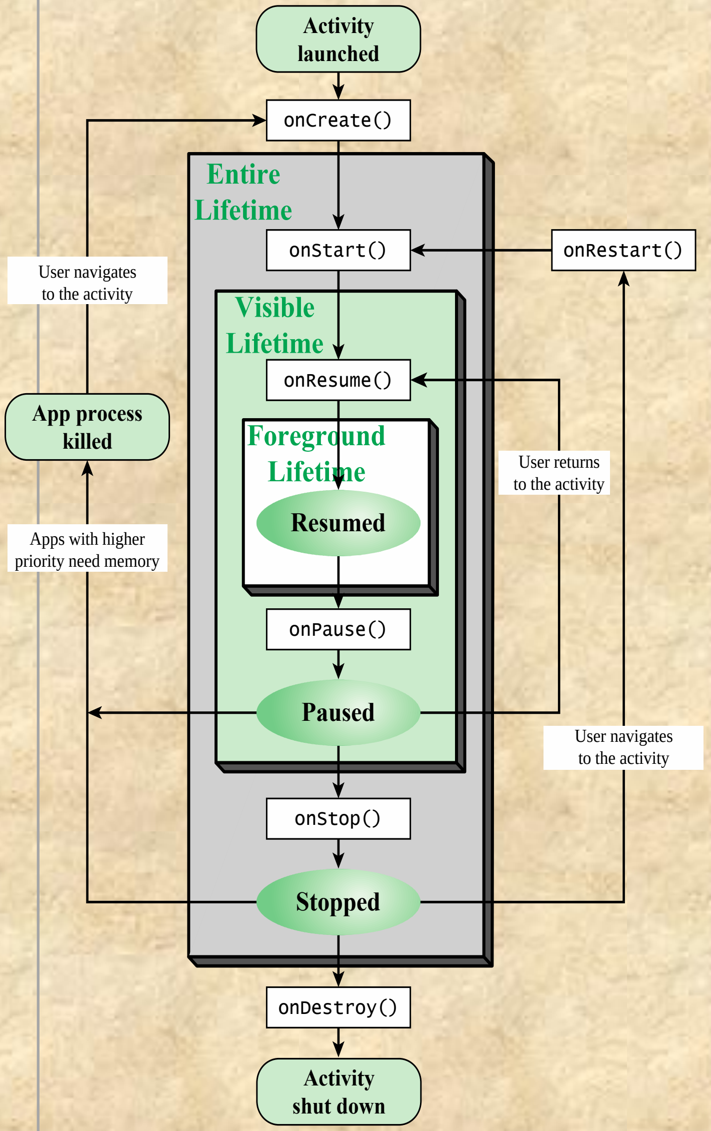


Figure 4.17 Activity State Transition Diagram

Figure 4.17 provides a simplified view of the state transition diagram of an activity. Keep in mind that there may be multiple activities in the application, each one at its own particular point on the state transition diagram. When a new activity is launched, the application software performs a series of system calls to the Activity Manager (Figure 2.20): onCreate() does the static setup of the activity, including any data structure initialization; onStart() makes the activity visible to the user on the screen; onResume() passes control to the activity so that user input goes to the activity. At this point the activity is in the Resumed state. This is referred to as the foreground lifetime of the activity. During this time, the activity is in front of all other activities on screen and has user input focus.

A user action may invoke another activity within the application. For example, during the execution of the e-mail application, when the user selects an e-mail, a new activity opens to view that e-mail. The system responds to such an activity with the onPause() system call, which places the currently running activity on the stack, putting it in the Paused state. The application then creates a new activity, which will enter the Resumed state.

At any time, a user may terminate the currently running activity by means of the Back button, closing a window, or some other action relevant to this activity. The application then invokes onStop(0) to stop the activity. The application then pops the activity that is on the top of the stack and resumes it. The Resumed and Paused states together constitute the visible lifetime of the activity. During this time, the user can see the activity on-screen and interact with it.

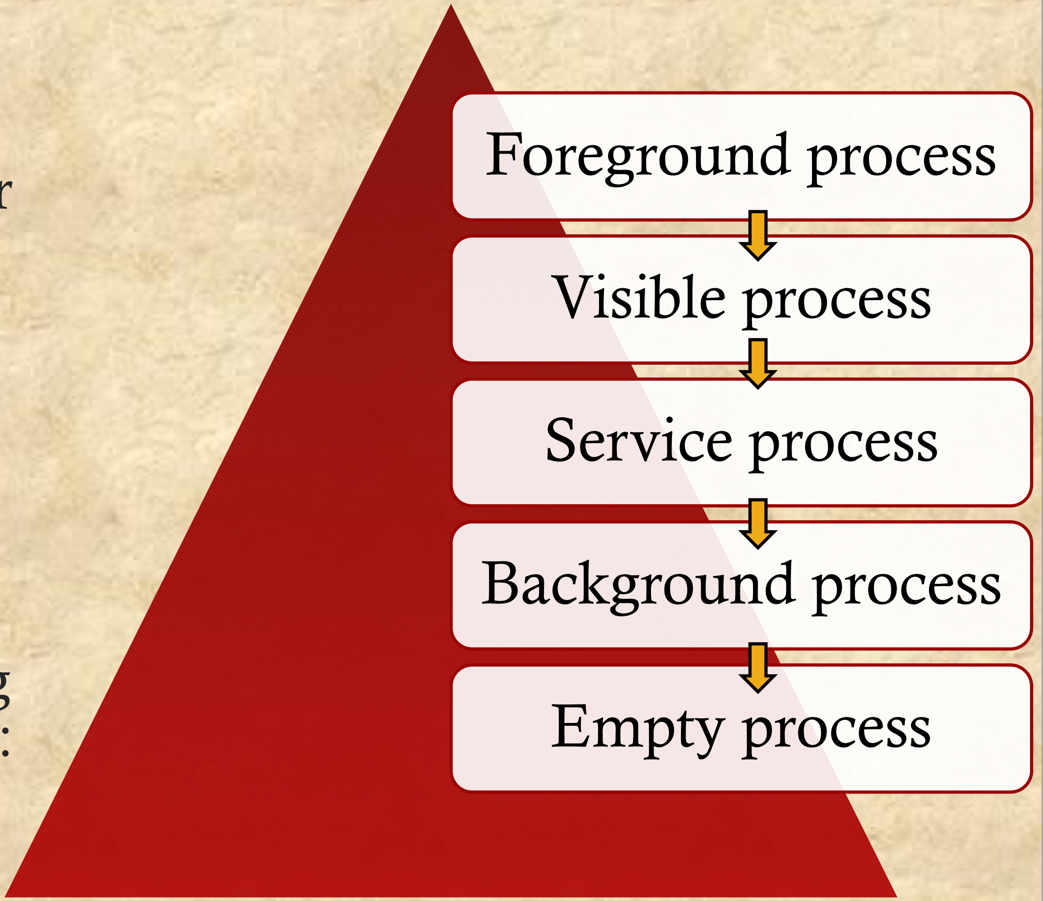
If the user leaves one application to go to another, for example, by going to the Home screen, the currently running activity is paused and then stopped. When the user resumes this application, the stopped activity, which is on top of the back stack, is restarted and becomes the foreground activity for the application.

Processes and Threads

- The default allocation of processes and threads to an application is a single process and a single thread. All of the components of the application run on the single thread of the single process for that application.
- To avoid slowing down the user interface when slow and/or blocking operations occur in a component, the developer can create multiple threads within a process and/or multiple processes within an application. In any case, all processes and their threads for a given application execute within the same virtual machine.
- In order to reclaim memory in a system that is becoming heavily loaded, the system may kill one or more processes. As was discussed in the preceding section, when a process is killed, one or more of the activities supported by that process are also killed.
- A precedence hierarchy is used to determine which process or processes to kill in order to reclaim needed resources.

Processes and Threads

- A precedence hierarchy is used to determine which process or processes to kill in order to reclaim needed resources
- Processes are killed beginning with the lowest precedence first
- The levels of the hierarchy, in descending order of precedence are:



Mac OS X Grand Central Dispatch (GCD)

- Mac OS X Grand Central Dispatch (GCD) provides a pool of available threads
- Designers can designate portions of applications, called *blocks,* that can be dispatched independently and run concurrently
- Concurrency is based on the number of cores available and the thread capacity of the system

Blocks

- A simple extension to a language. Here is a simple example of a block definition:

```
x = ^{ printf("hello world\n"); }
```

- A block is denoted by a caret at the start of the function, which is enclosed in curly brackets.
- A block defines a self-contained unit of work
- Enables the programmer to encapsulate complex functions
- Blocks are scheduled and dispatched by queues
- Dispatched on a first-in-first-out basis
- Can be associated with an event source, such as a timer, network socket, or file descriptor

Summary

- Processes and threads
  - Multithreading
  - Thread functionality
- Types of threads
  - User level and kernel level threads
- Multicore and multithreading
  - Performance of Software on Multicore
- Windows process and thread management
  - Management of background tasks and application lifecycles
  - Windows process
  - Process and thread objects
  - Multithreading
  - Thread states
  - Support for OS subsystems
- Solaris thread and SMP management
  - Multithreaded architecture
  - Motivation
  - Process structure
  - Thread execution
  - Interrupts as threads
- Linux process and thread management
  - Tasks/threads/namespaces
- Android process and thread management
  - Android applications
  - Activities
  - Processes and threads
- Mac OS X grand central dispatch