Operating Systems: Internals and Design Principles

Chapter 4 Threads

Seventh Edition

By William Stallings

Operating Systems: Internals and Design Principles

The basic idea is that the several components in any complex system will perform particular subfunctions that contribute to the overall function.

—THE SCIENCES OF THE ARTIFICIAL,

Herbert Simon

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

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

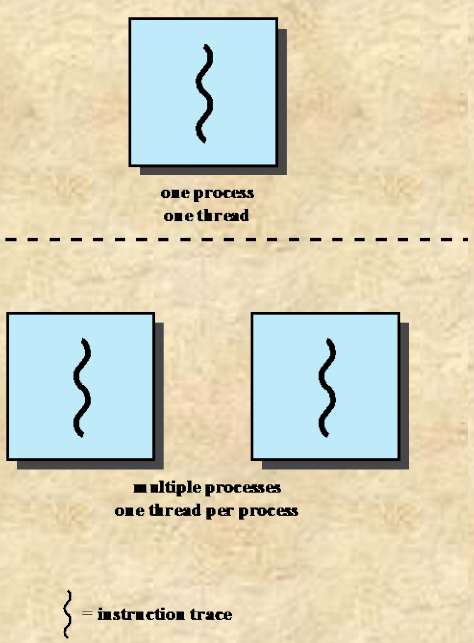


Figure 4.1 Threads and Processes

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

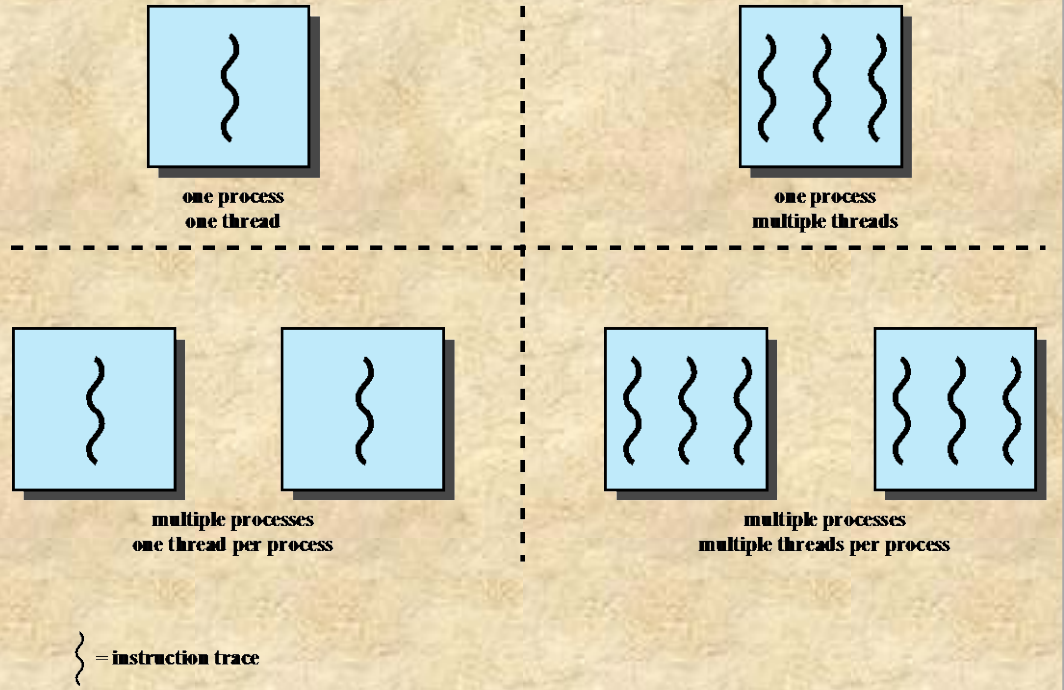


Figure 4.1 Threads and Processes

Multithreading

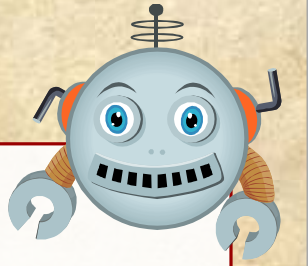
Refers to the ability of an operating system to support multiple threads of execution within a single process

Processes

- The unit or resource allocation and a unit of protection
- 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

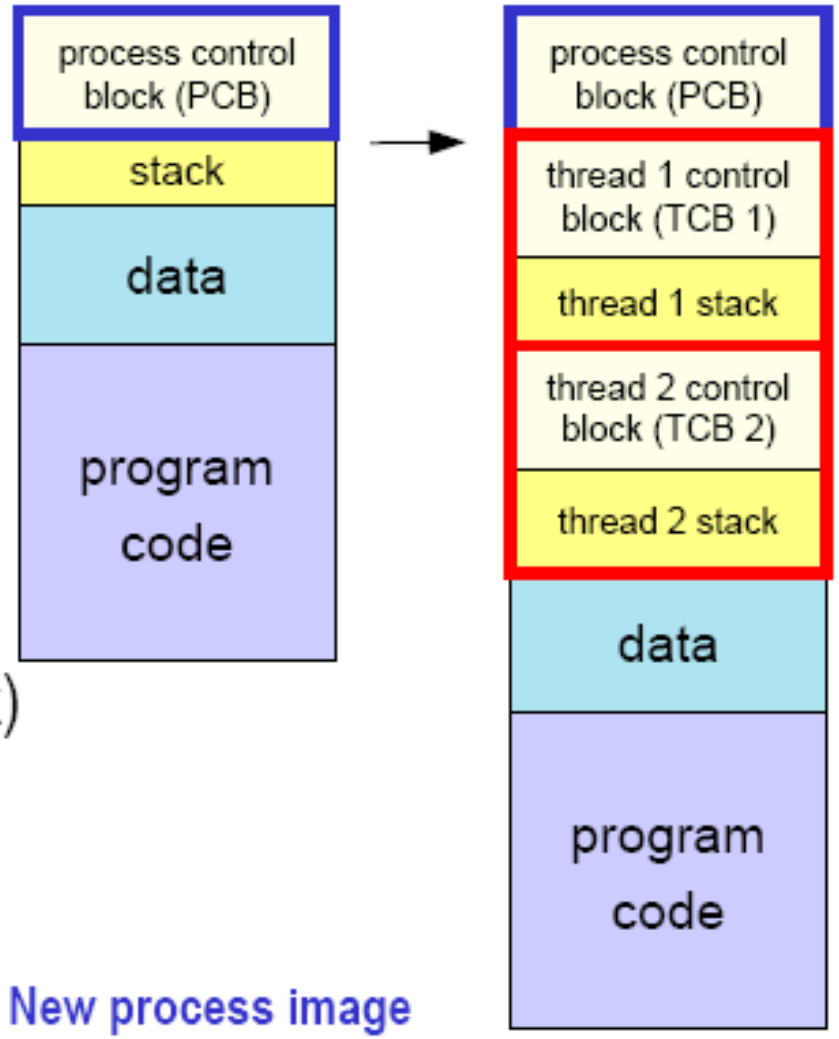
Each thread has:



- an execution state (Running, Ready, etc.)
- saved thread context when not running
- 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)

Execution and Scheduling

- **Multithreading requires changes in the process description model**
  - ✓ each thread of execution receives its own control block and stack
    - own execution state ("Running", "Blocked", etc.)
    - own copy of CPU registers
    - own execution history (stack)
  - ✓ the process keeps a global control block listing resources currently used



Execution and Scheduling

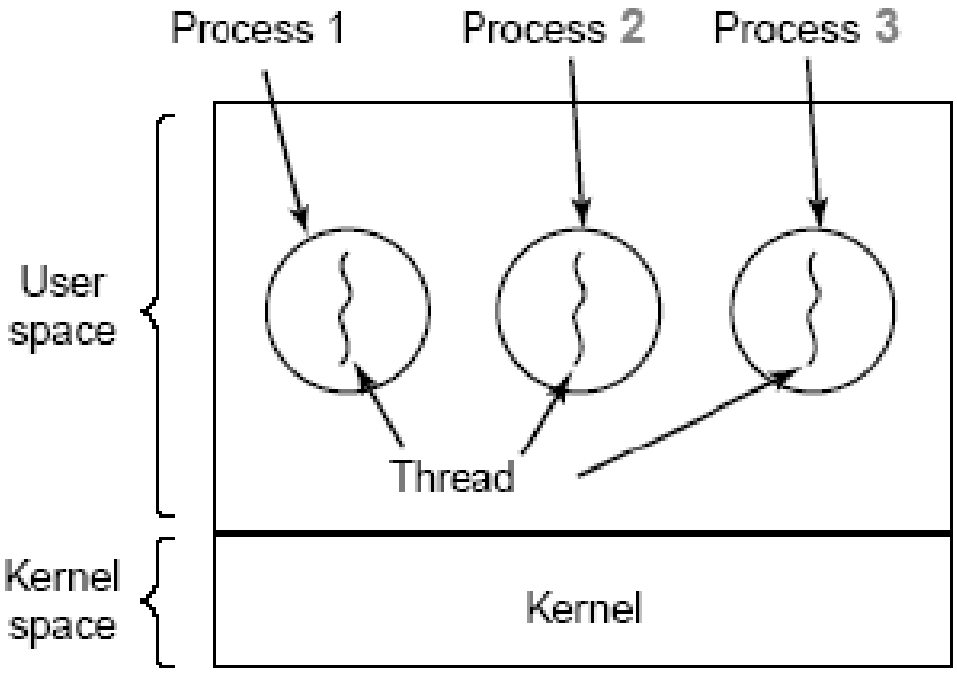
- **Per-process items and per-thread items in the control block structures**
  - ✓ process identification data + thread identifiers
    - numeric identifiers of the process, the parent process, the user, etc.
  - ✓ CPU state information
    - user-visible, control & status registers
    - stack pointers
  - ✓ process control information
    - scheduling: state, priority, awaited event
    - used memory and I/O, opened files, etc.
    - pointer to next PCB

Execution & Scheduling

- **Multithreaded process model**
  - all threads share the same address space and resources
  - spawning a new thread only involves allocating a new stack and a new CPU state block

Execution & Scheduling

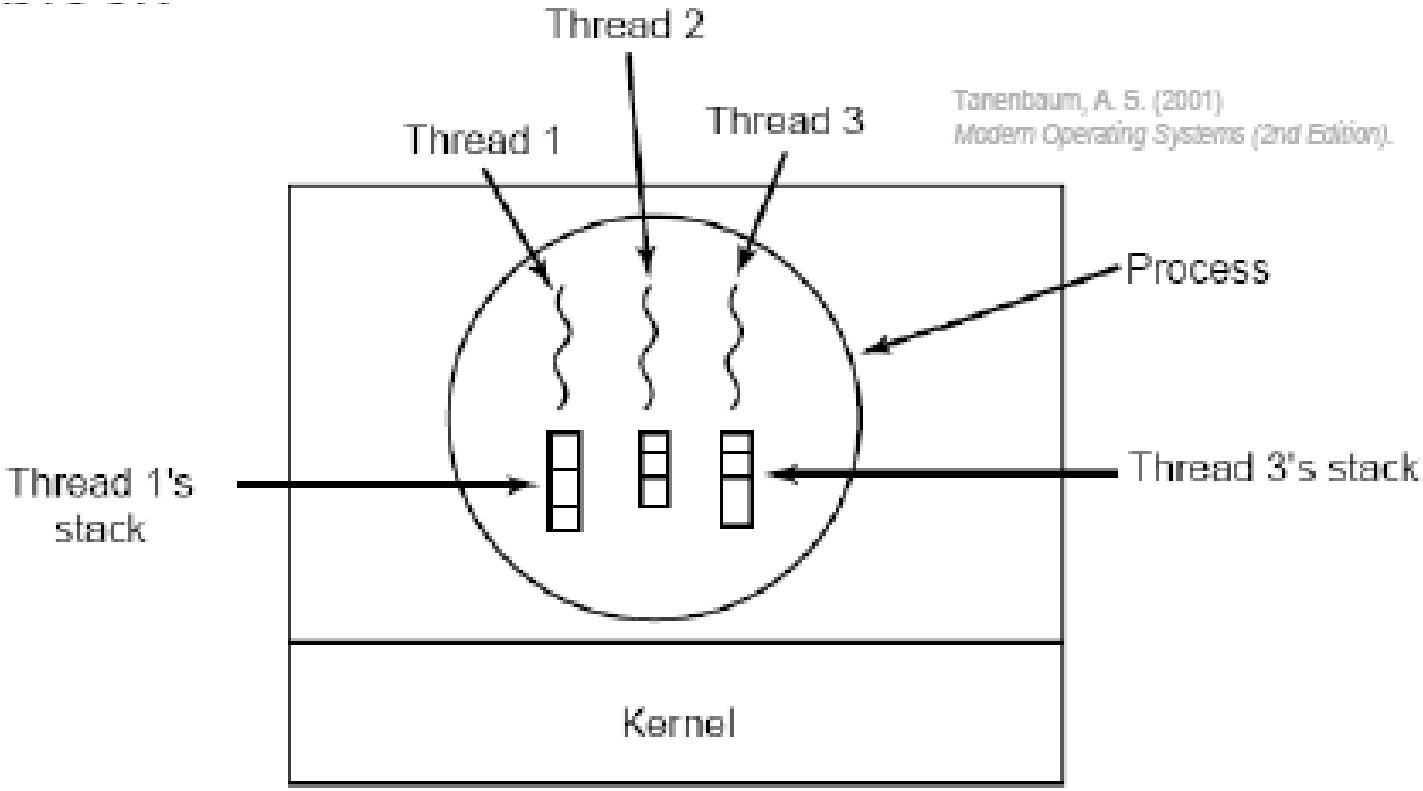
Single-threaded and multithreaded process models (in abstract space)



(a) Three processes with one thread

Execution & Scheduling

Single-threaded and multithreaded process models (in abstract space)



(a) One process with three threads

Threads

- An execution state (running, ready, etc.)
- Saved thread context when not running
- Has 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 vs. Processes

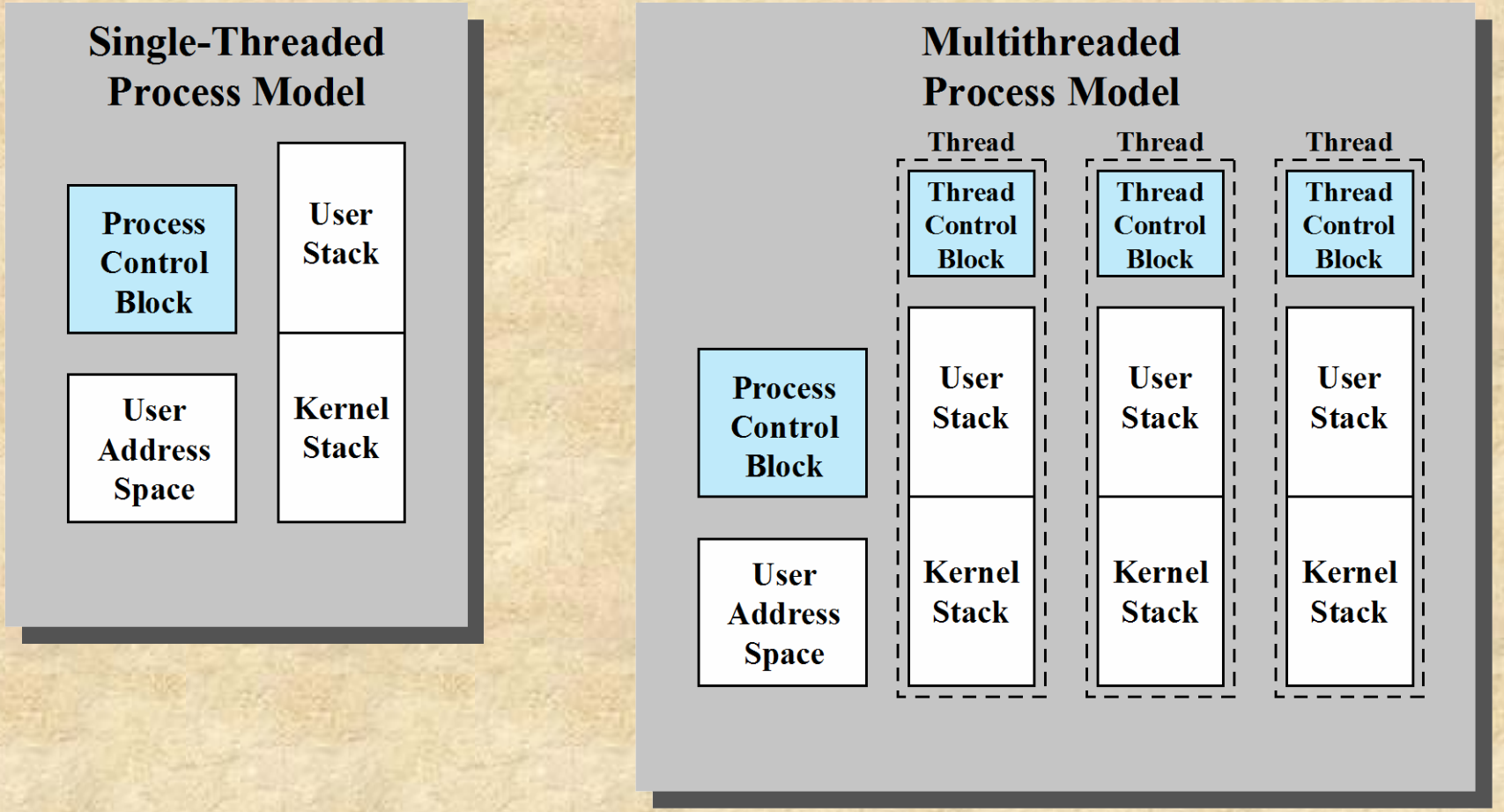
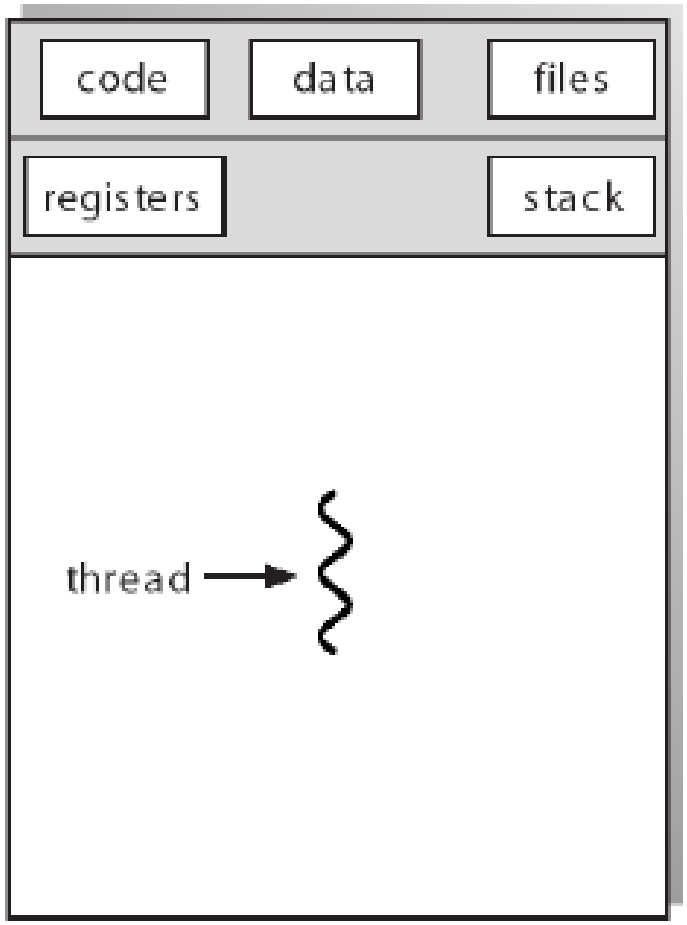
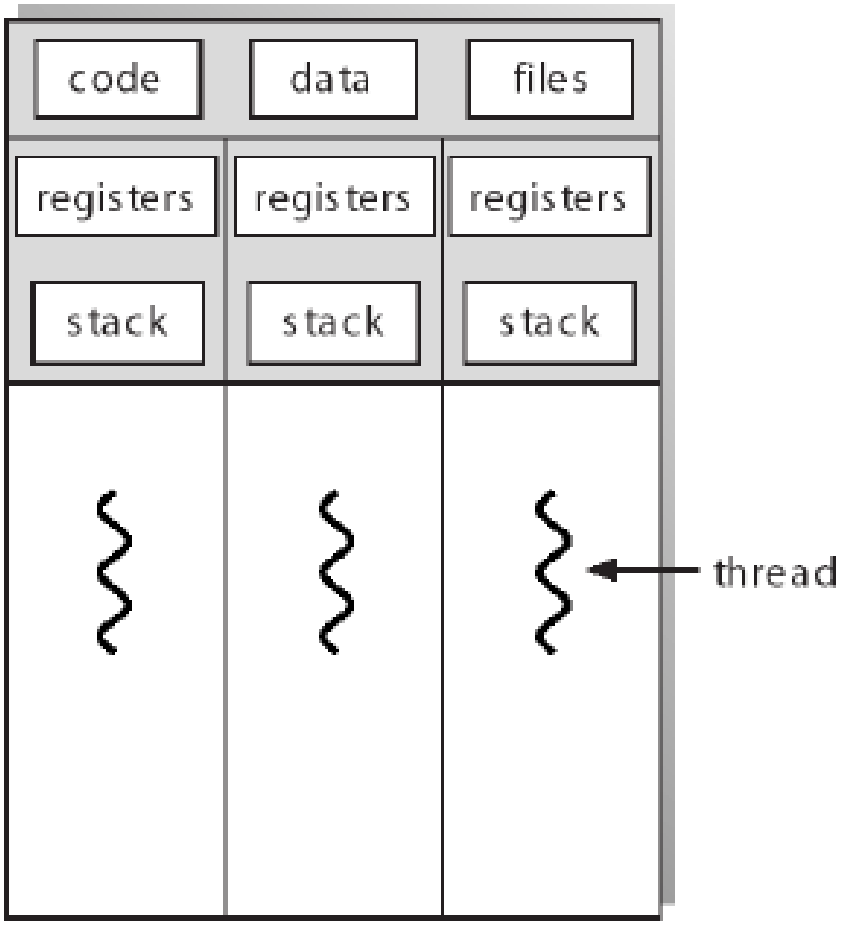


Figure 4.2 Single Threaded and Multithreaded Process Models

Multithreaded Process Model



single-threaded process



multithreaded process

Silberschatz, A., Galvin, P. B. and Gagne. G. (2003) *Operating Systems Concepts with Java (6th Edition)*.

Single-threaded and multithreaded process models (in abstract space)

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:

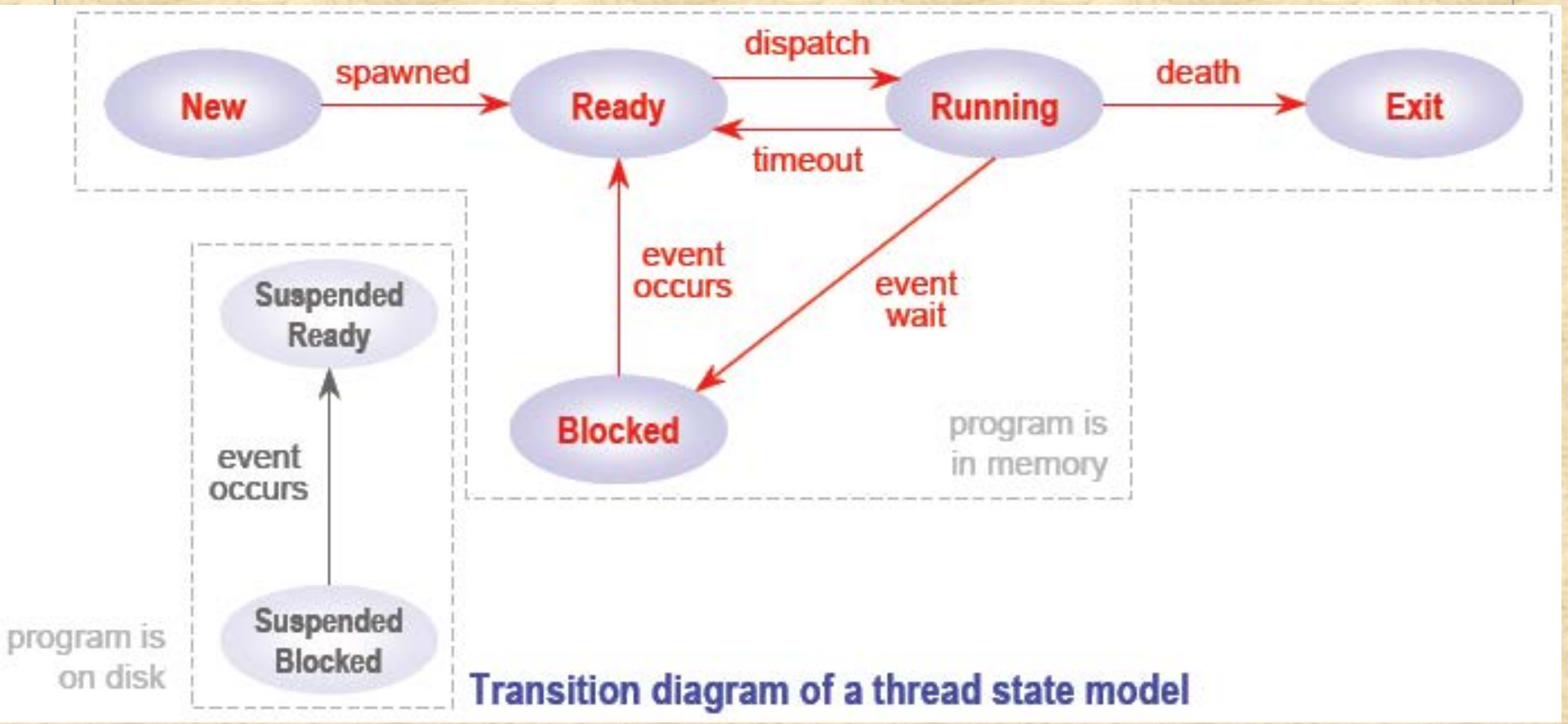
- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish

Thread States

- threads (like processes) can be ready, running or blocked
- threads can't be suspended ("swapped out"), only processes can



Transition diagram of a thread state model

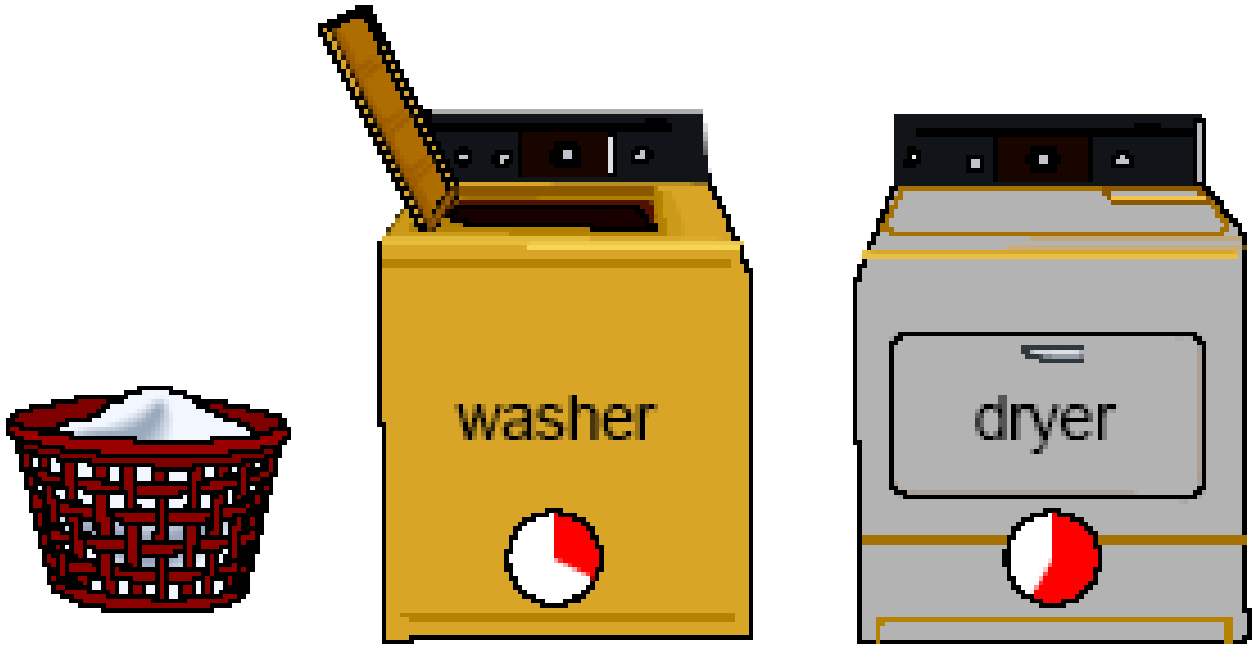
Threads

- Actions that affect all of the 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

Threads

It's the same old throughput story, again

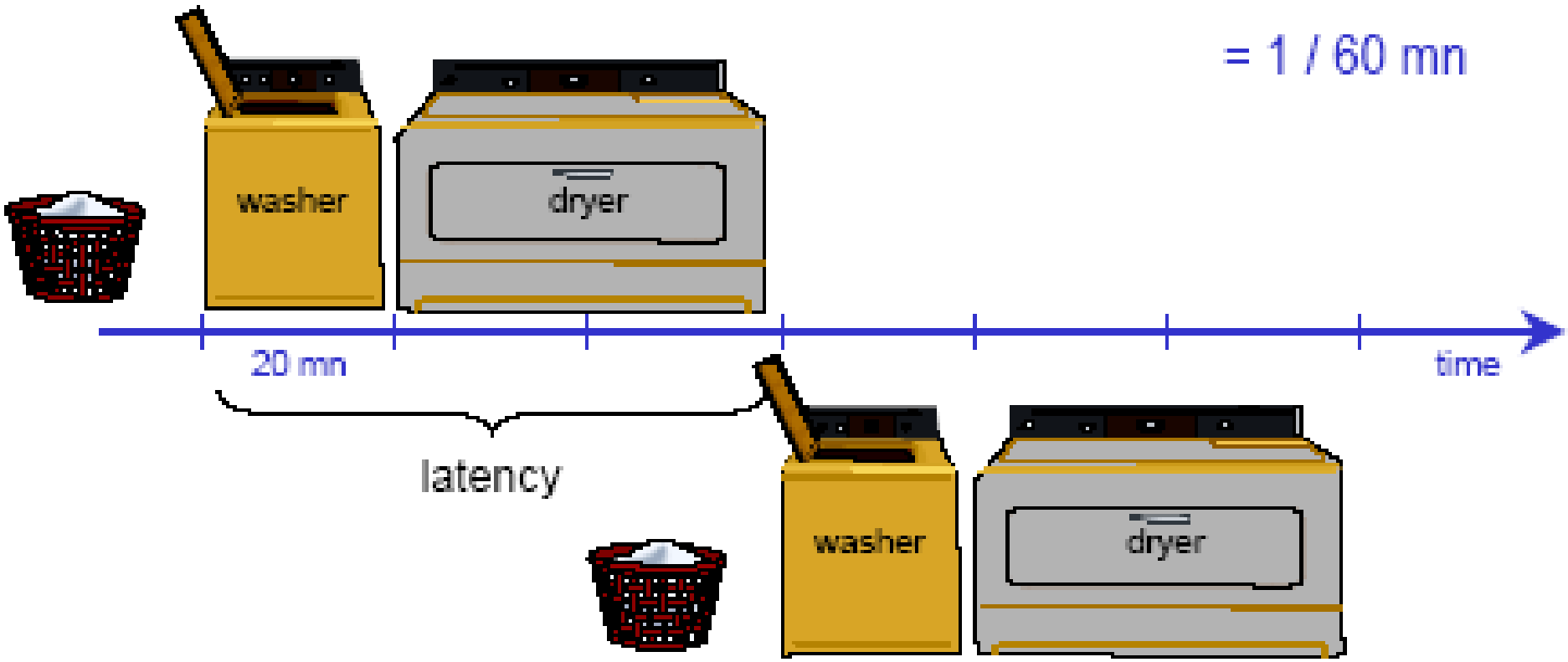
- In the laundry room
  - ✓ the washing machine takes 20 minutes
  - ✓ the dryer takes 40 minutes



Threads

It's the same old throughput story, again

- Doing two loads in a sequence
  - ✓ **latency** = time for one execution to complete = 60 mn
  - ✓ **throughput** = rate of completed executions = 2 / 120 mn
    = 1 / 60 mn

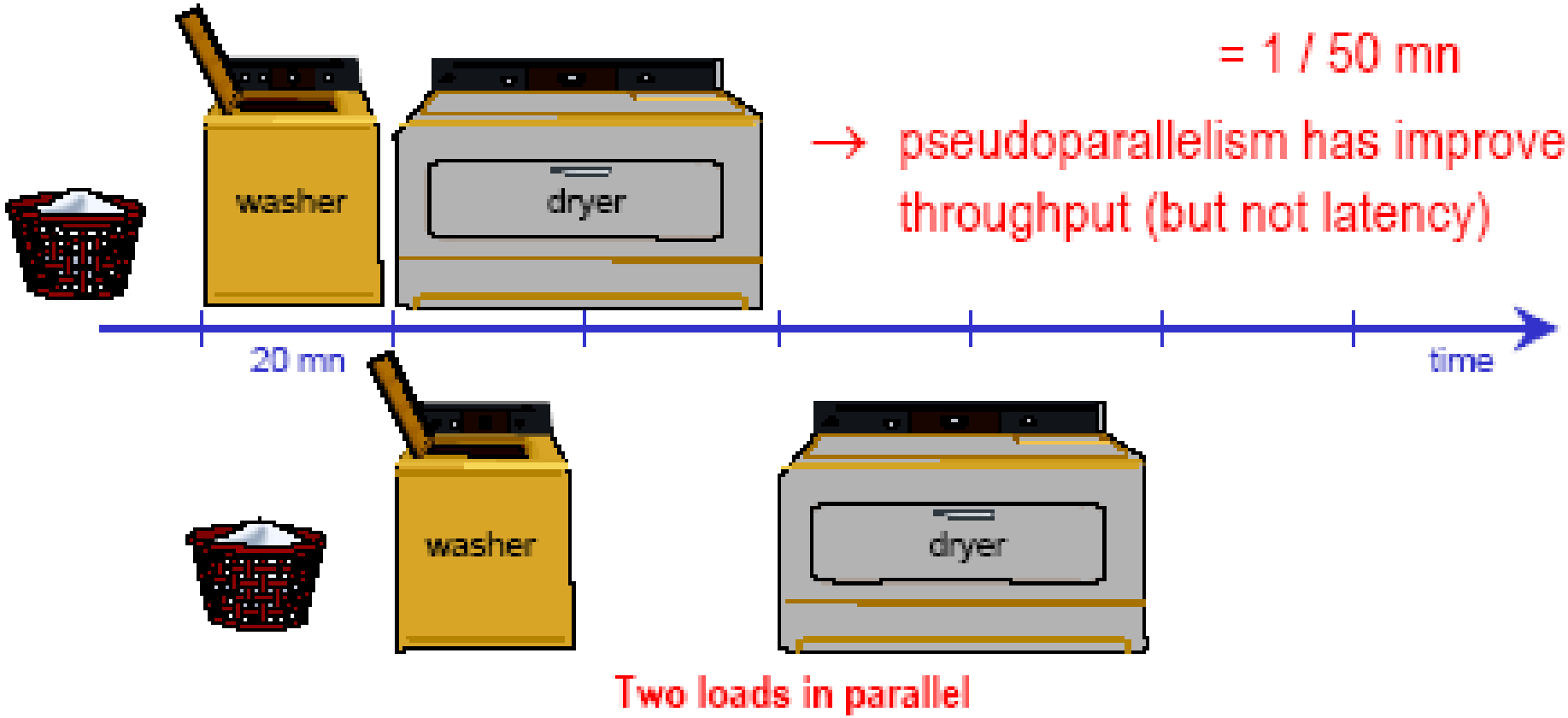


Two loads in a sequence

Threads

It's the same old throughput story, again

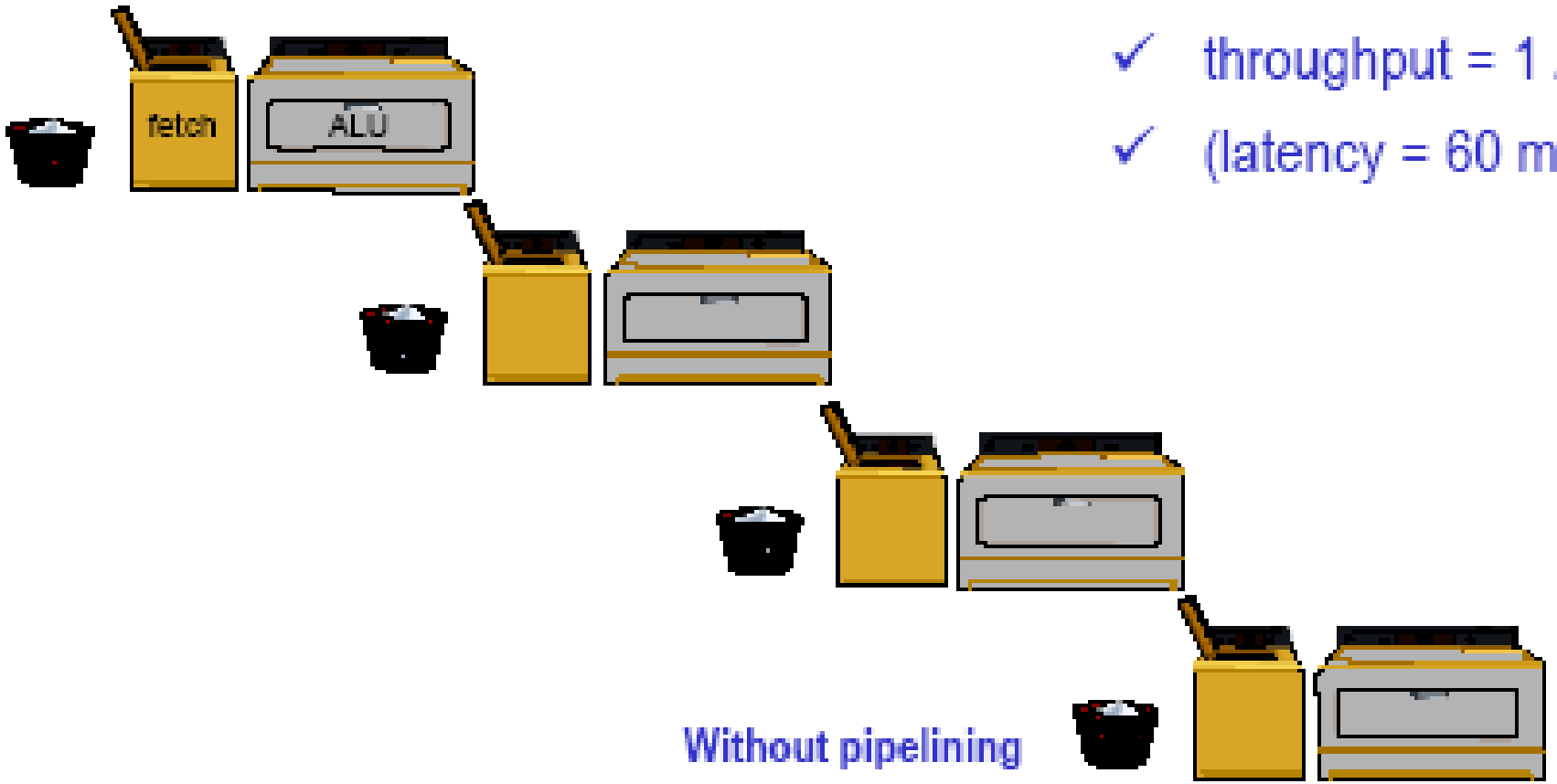
- Doing two loads in (pseudo)parallel
  - ✓ **latency** = time for one execution to complete = 60 to 80 mn
  - ✓ **throughput** = rate of completed executions = 2 / 100 mn



Threads

It's the same old throughput story, again

- This is the principle used in processor pipelining
  - ✓ here, washer & dryer are regularly clocked stages
  - ✓ without pipelining: throughput is 1 over the sum of all stages
    - ✓ throughput = 1 / 60 mn
    - ✓ (latency = 60 mn)

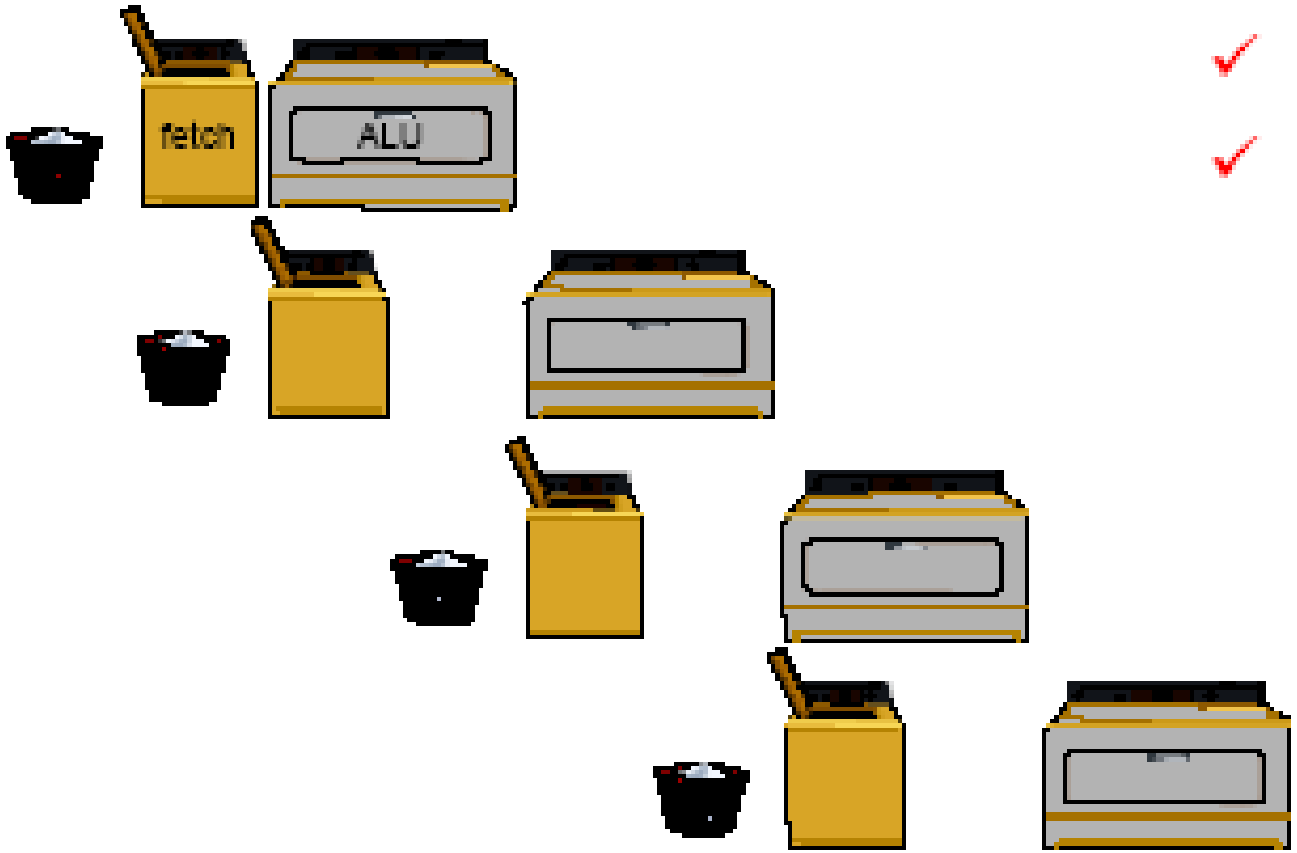


Without pipelining

Threads

It's the same old throughput story, again

- **This is the principle used in processor pipelining**
  - ✓ here, washer & dryer are regularly clocked stages
  - ✓ with pipelining: throughput is only 1 over the longest stage
    - ✓ throughput = 1 / 40 mn
    - ✓ (but latency = 80 mn)

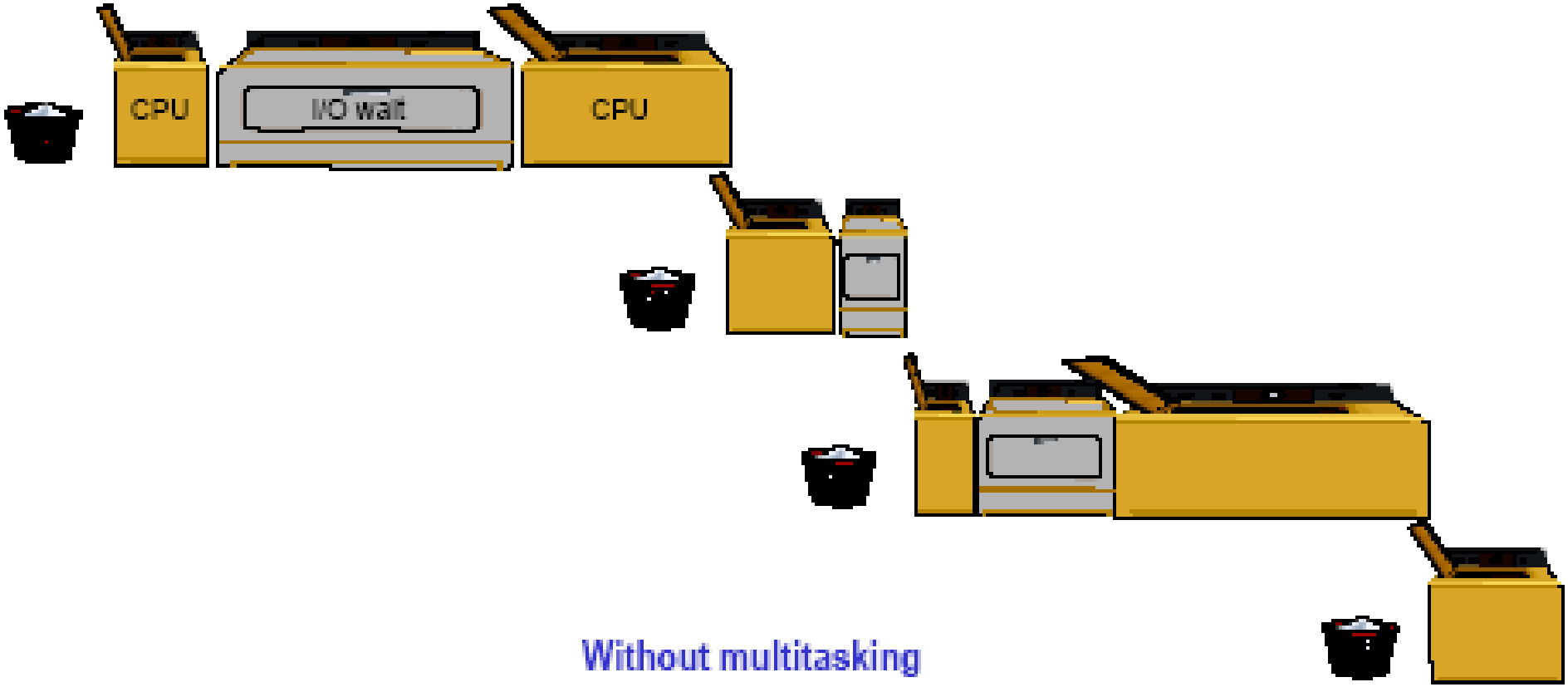


With pipelining

Threads

It's the same old throughput story, again

- **This is also the principle used in multitasking**
  - ✓ here, the washer is the CPU and the dryer is one I/O device
  - ✓ wash & dry times may vary with loads and repeat in any order

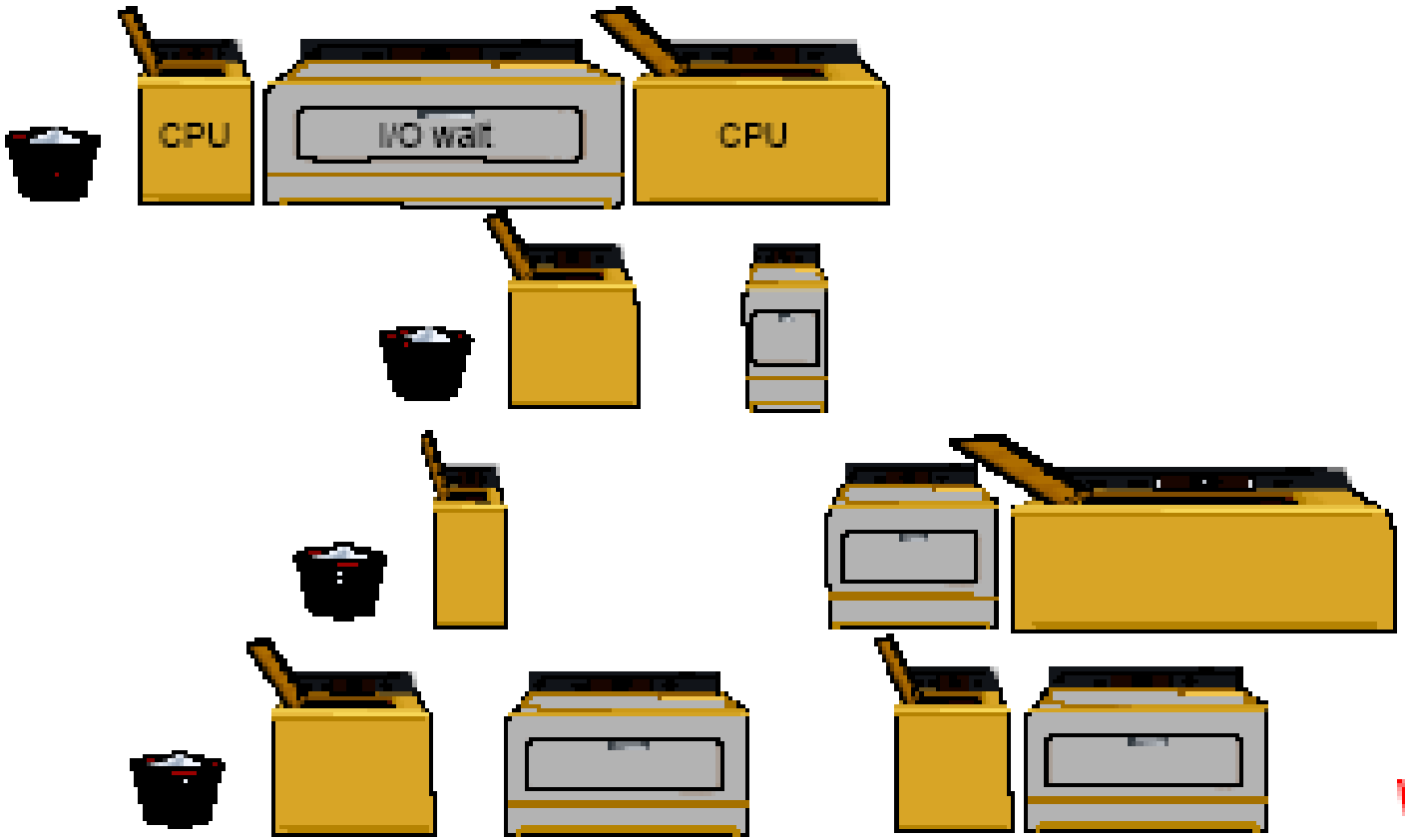


Without multitasking

Threads

It's the same old throughput story, again

- **This is also the principle used in multitasking**
  - ✓ thanks to multitasking, throughput (CPU utilization) is much higher (but the total time to complete a process is also longer)

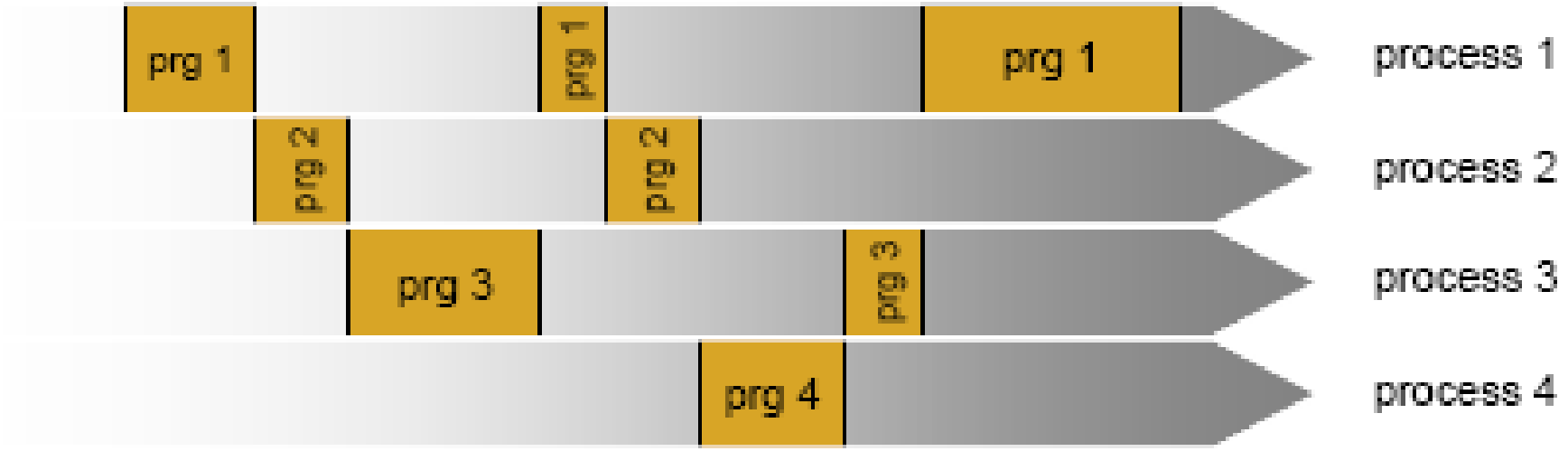


With multitasking

Threads

It's the same old throughput story, again

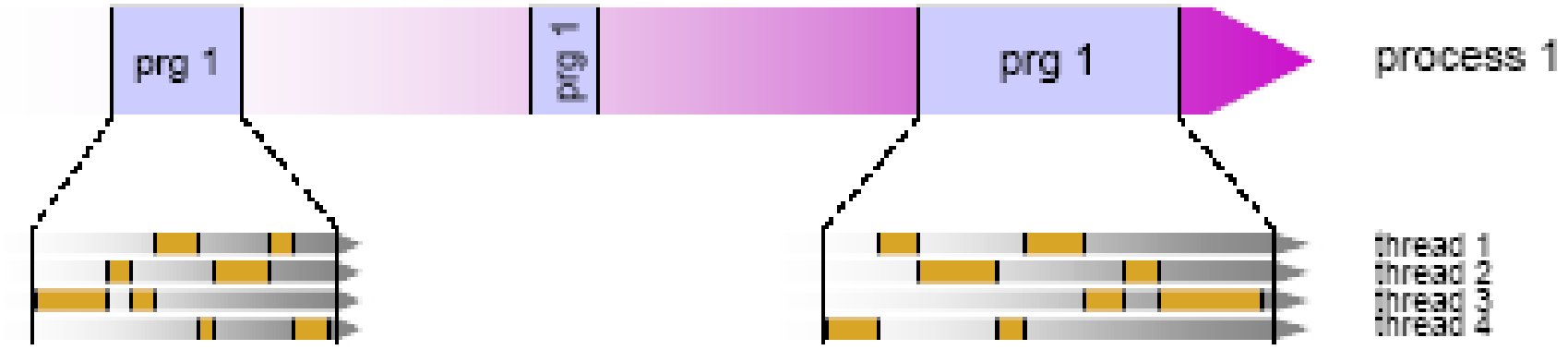
- This is also the principle used in multitasking



Threads

It's the same old throughput story, again

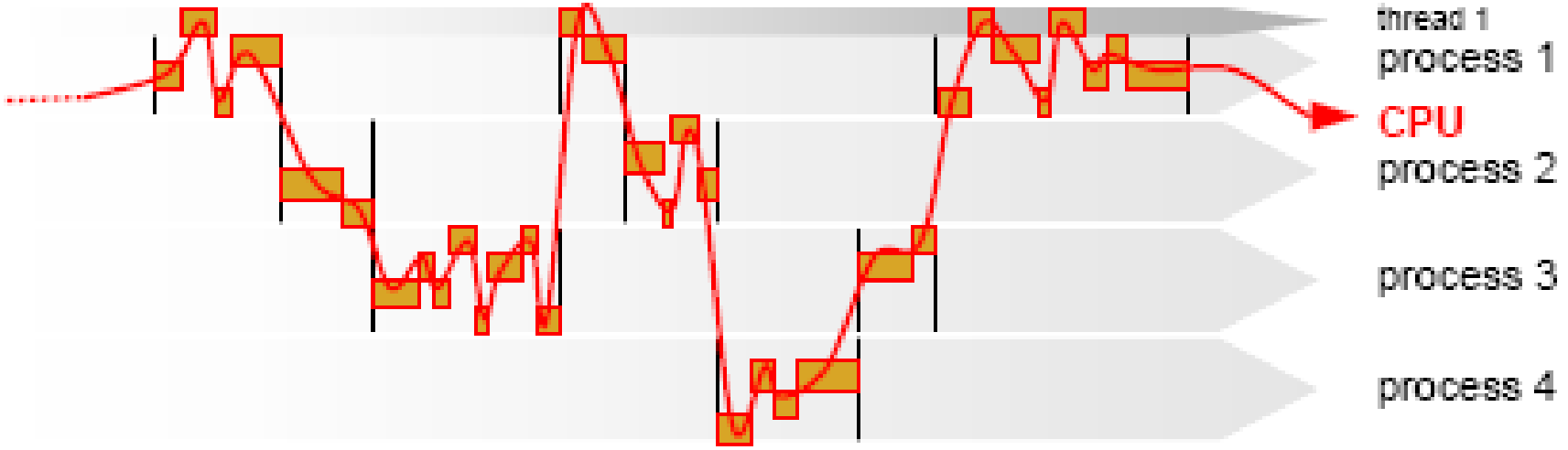
- **And, naturally, the same idea applies in multithreading**
  - ✓ multithreading is basically the same as multitasking at a finer level of temporal resolution (and within the same address space)
  - ✓ the same illusion of parallelism is achieved at a finer grain



Threads

It's the same old throughput story, again

- **And, naturally, the same idea applies in multithreading**
  - ✓ in a single-processor system, there is still only one CPU (washing machine) going through all the threads of all the processes



Multithreading

Threads

It's the same old throughput story, again

- **From processes to threads: a shift of levels**
  - container paradigm
    - there can be multiple processes running in one computer
    - there can be multiple threads running in one process
  - resource sharing paradigm
    - multiple processes share hardware resources: CPU, physical memory, I/O devices
    - multiple threads share process-owned resources: memory address space, opened files, etc.

Threads

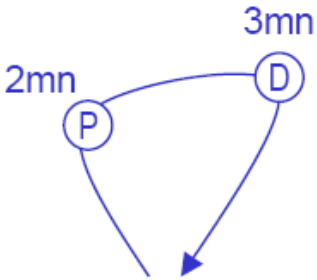
It's the same old throughput story, again

- **From processes to threads: a shift of levels**
  - container paradigm
    - there can be multiple processes running in one computer
    - there can be multiple threads running in one process
  - resource sharing paradigm
    - multiple processes share hardware resources: CPU, physical memory, I/O devices
    - multiple threads share process-owned resources: memory address space, opened files, etc.

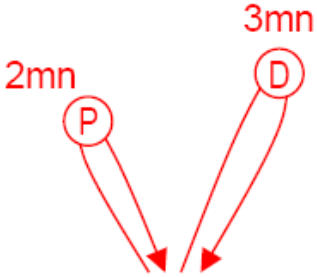
Threads

It's the same old throughput story, again

- **Illustration: two shopping scenarios**
  - ✓ Single-threaded shopping



    - you are in the grocery store
    - first you go to produce and grab salad and apples, then you go to dairy and grab milk, butter and cheese
    - it took you about 1mn x 5 items = 5mn
  - ✓ Multithreaded shopping



    - you take your two kids with you to the grocery store
    - you send them off in two directions with two missions, one toward produce, one toward dairy
    - you wait for their return (at the slot machines) for a maximum duration of about 1mn x 3 items = 3mn

Threads

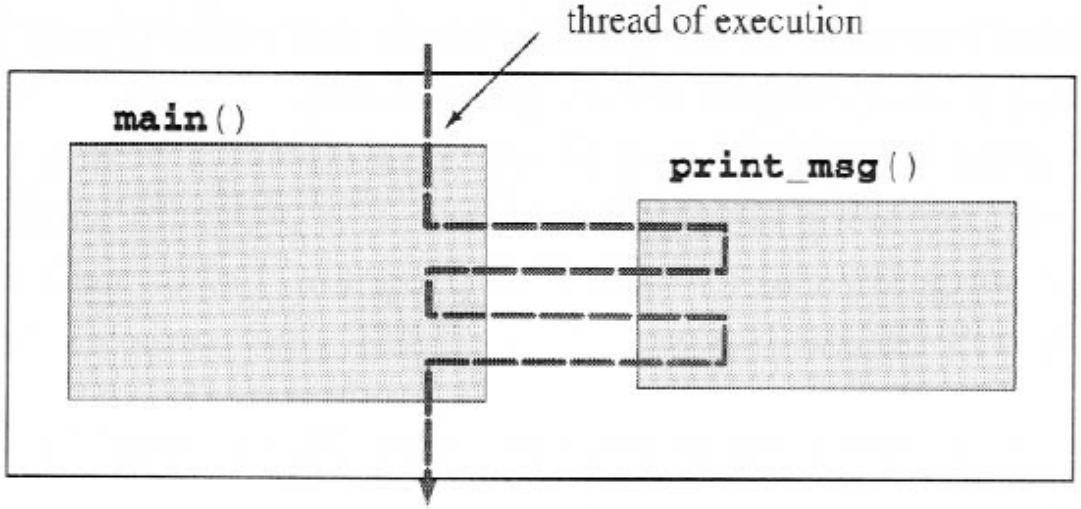
It's the same old throughput story, again

```
void main ( … ) {
    int i;
    char * produce[] = {“salad”, “apples”, NULL};
    char * diary[] = {“milk”, “butter”, “cheese”, NULL};

    print_msg(produce);
    print_msg(diary);
}

void *print_msg(char *item) {
    int i;
    while (items[i] != NULL) {
            printf(“ grabbing the %s…”, (char *) (item[i++]));
            fflush (stdout);
            sleep (1);
    }
    return NULL;
}
```

Threads

It's the same old throughput story, again

- **Results of single-threaded shopping**
  - ✓ total duration ≈ 5 seconds; outcome is deterministic



one process,
two functions,
one thread

Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

```
> ./single_shopping
grabbing the salad...
grabbing the apples...
grabbing the milk...
grabbing the butter...
grabbing the cheese...
>
```

Single-threaded shopping diagram and output

Threads

It's the same old throughput story, again

```
void main(...)
{
    char *produce[] = { "salad", "apples", NULL };
    char *dairy[] = { "milk", "butter", "cheese", NULL };
    void *print_msg(void *);
    pthread_t th1, th2;

    pthread_create(&th1, NULL, print_msg, (void *)produce);
    pthread_create(&th2, NULL, print_msg, (void *)dairy);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
}

void *print_msg(void *items)
{
    int i = 0;
    while (items[i] != NULL) {
        printf("grabbing the %s...", (char *)(items[i++]));
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}
```

send the kids off!

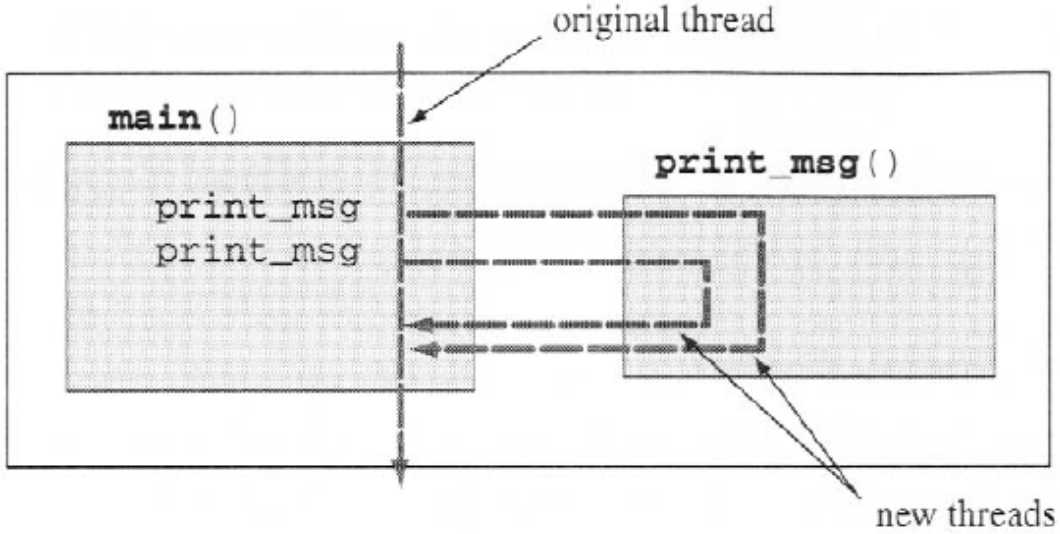
wait for their return

Multithreaded shopping code

Threads

It's the same old throughput story, again

- **Results of multithreaded shopping**
  - ✓ total duration ≈ 3 seconds; outcome is nondeterministic



Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```

```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

Multithreaded shopping diagram and possible outputs

Threads

Practical Uses of Multithreading

- **System calls for thread creation and termination wait**
  - ✓
```
err = pthread_create(pthread_t  *th,
                     pthread_attr_t  *attr,
                     void  *(*func)(void *),
                     void  *arg)
```

    creates a new thread of execution and calls `func(arg)` within that thread; the new thread can be given specific attributes `attr` or default attributes `NULL`

  - ✓
```
err = pthread_join(pthread_t  th,
                   void  **retval)
```

    blocks the calling thread until the thread specified by `th` terminates; the return value from `th` can be stored in `retval`

Uses of Threads in a Single-User Multiprocessing System

- Foreground to background work
  - one thread displays menus and read user input
  - another thread executes user commands
- Asynchronous processing
  - Asynchronous elements in the program can be implemented as threads
  - Handle external, surprise events such as client requests
- Speed execution
  - on a multiprocessor system, multiple threads from the same process may be able to execute simultaneously
  - “stagger” and overlap CPU execution time and I/O wait time
- Modular program structure

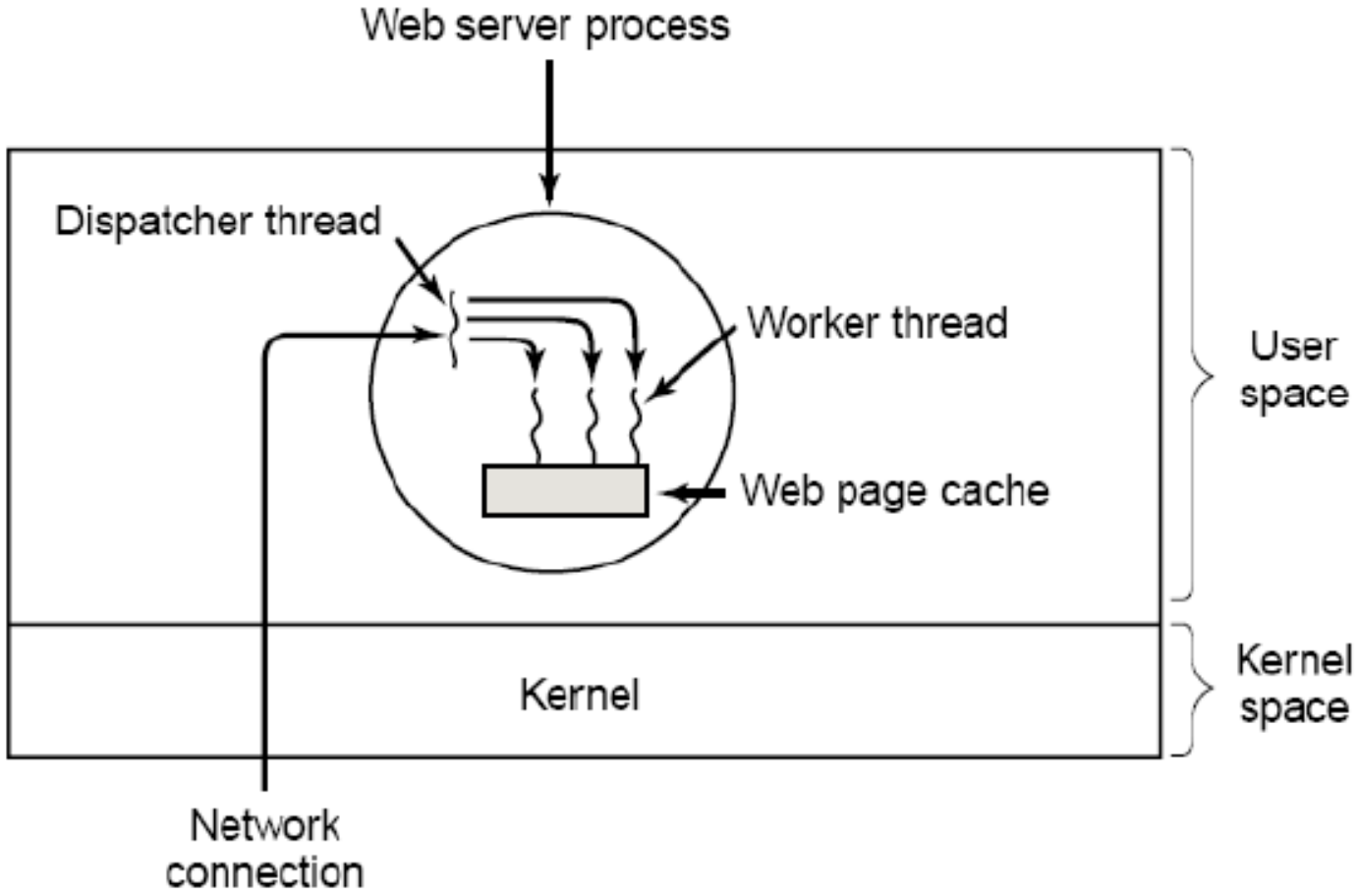
Examples of real-world multithreaded applications

- Web client (browser)
  - must download page components (images, styles, etc.) simultaneously; cannot wait for each image in series
- Web server
  - must serve pages to hundreds of Web clients simultaneously; cannot process requests one by one
- word processor, spreadsheet
  - provides uninterrupted GUI service to the user while reformatting or saving the document in the background

→*again, same principles as time-sharing (illusion of interactivity while performing other tasks), this time inside the same process*

Web server

- ✓ as each new request comes in, a “dispatcher thread” spawns a new “worker thread” to read the requested file (worker threads may be discarded or recycled in a “thread pool”)

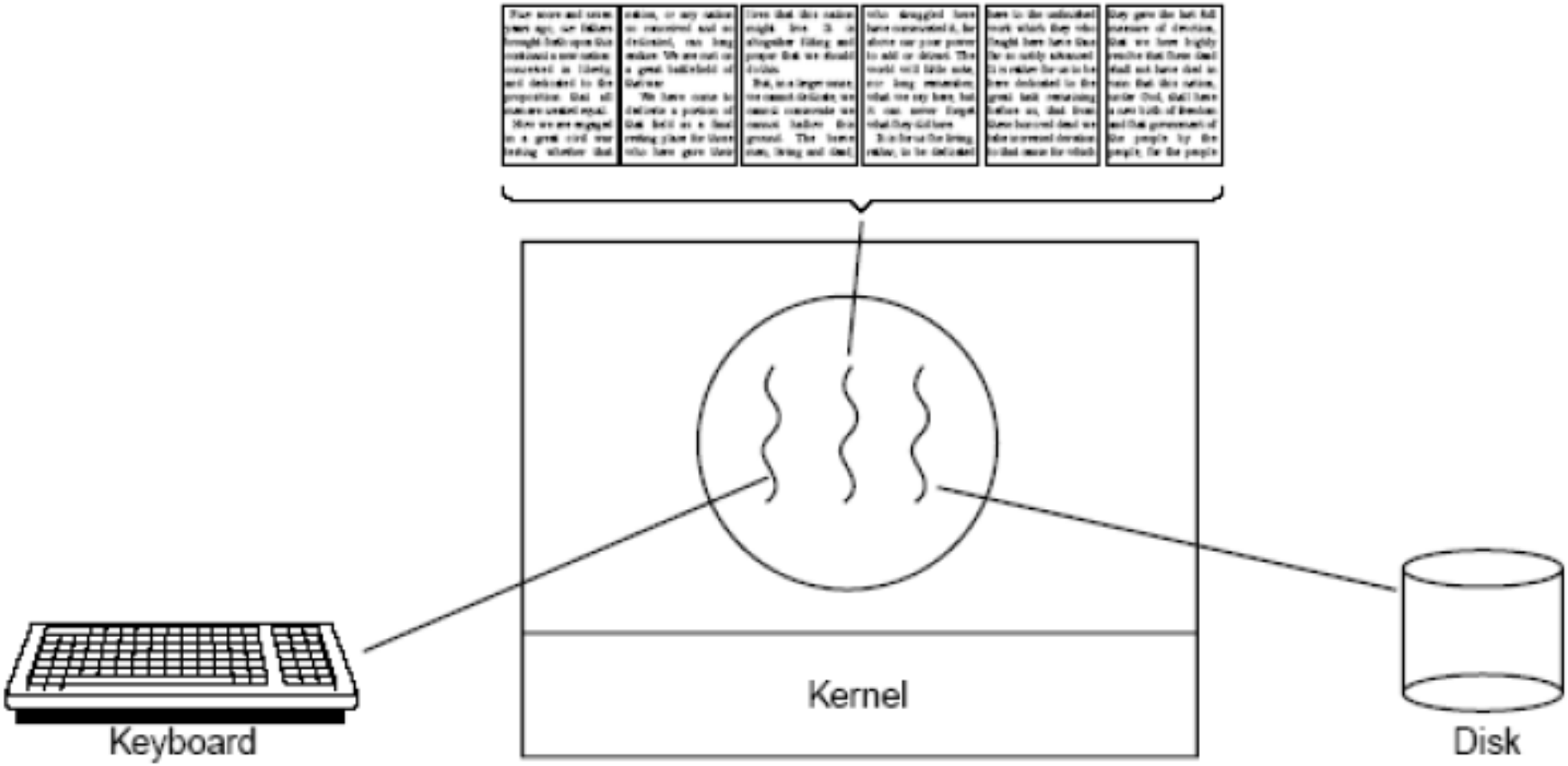


Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

A multithreaded Web server

Word processor

- ✓ one thread listens continuously to keyboard and mouse events to refresh the GUI; a second thread reformats the document (to prepare page 600); a third thread writes to disk periodically



A word processor with three threads

Adobe PageMaker

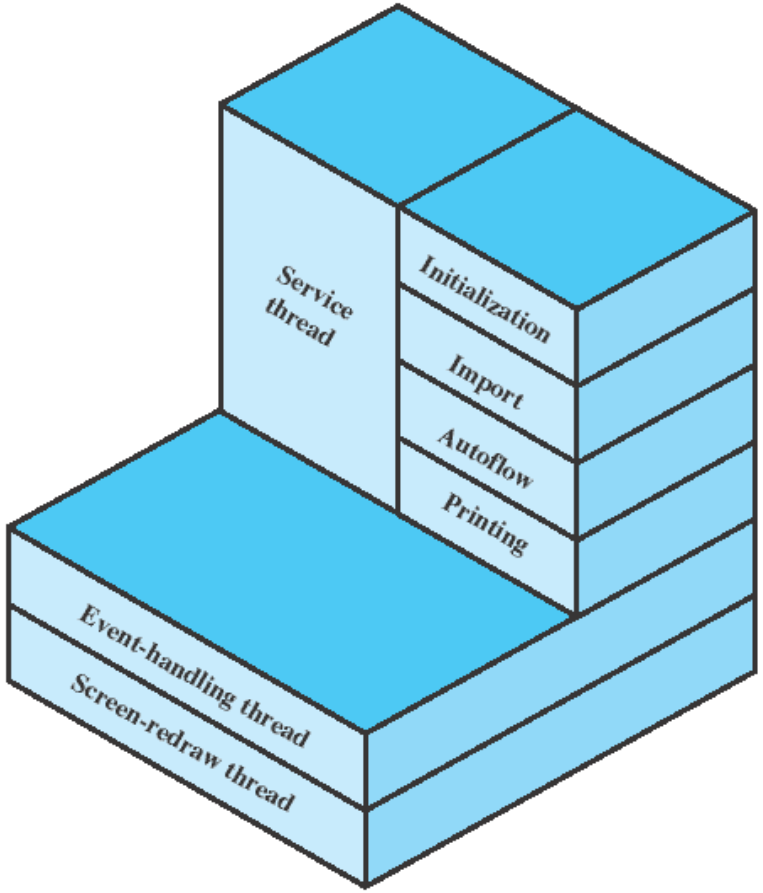
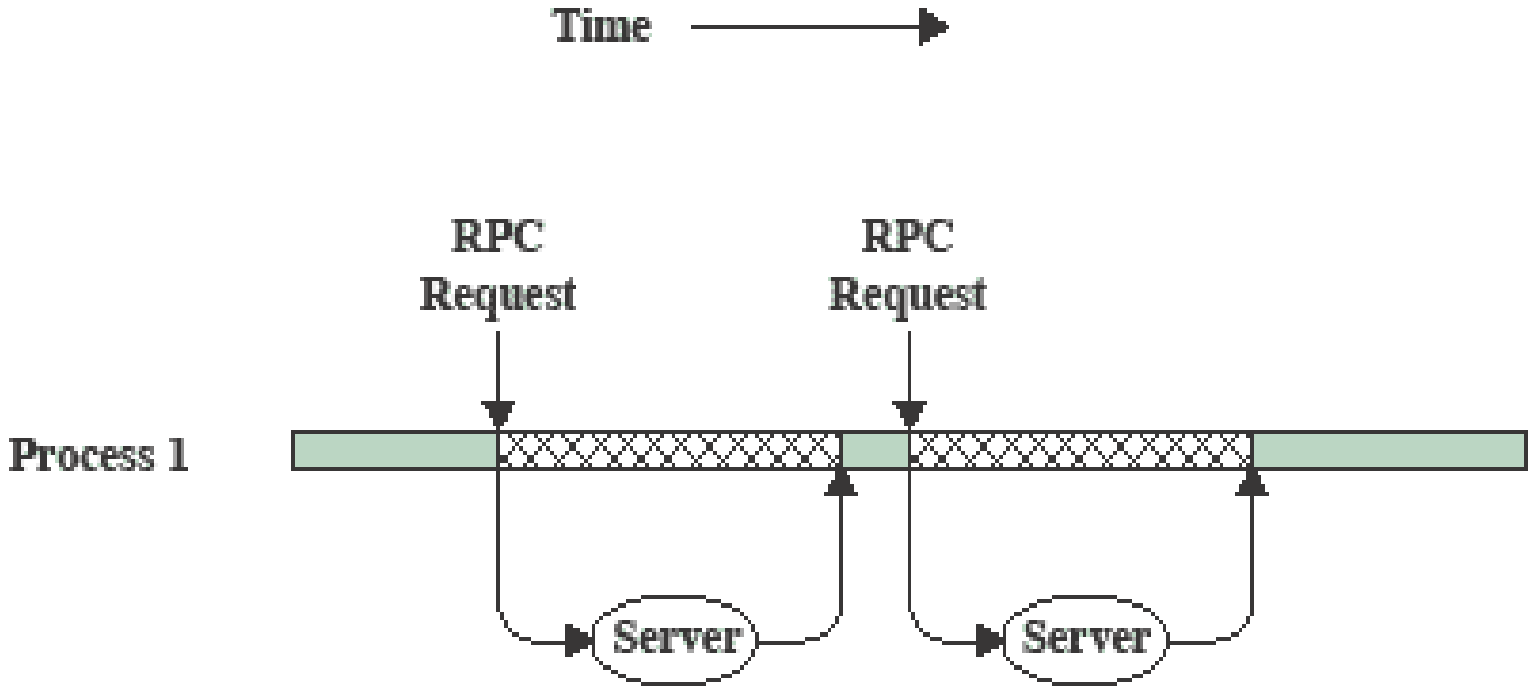


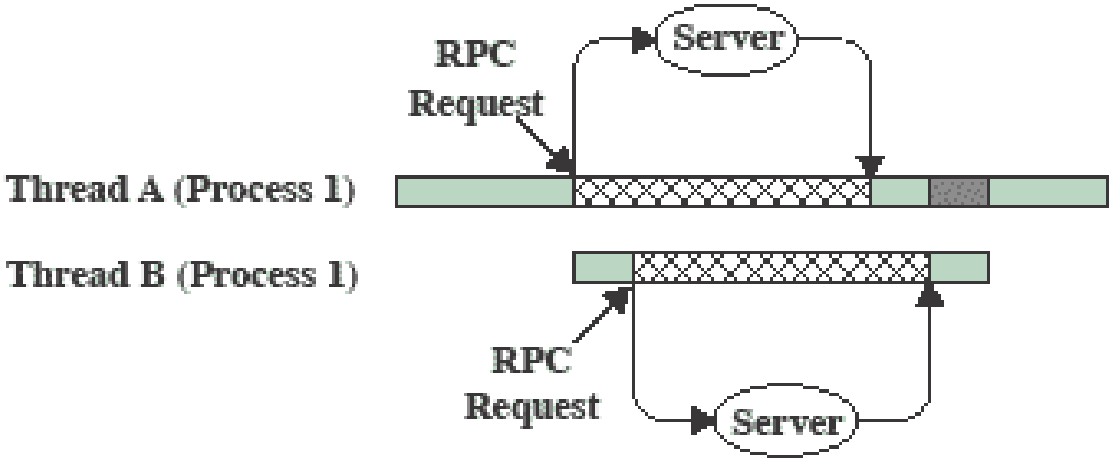
Figure 4.5 Thread Structure for Adobe PageMaker

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

Multithreading on a Uniprocessor

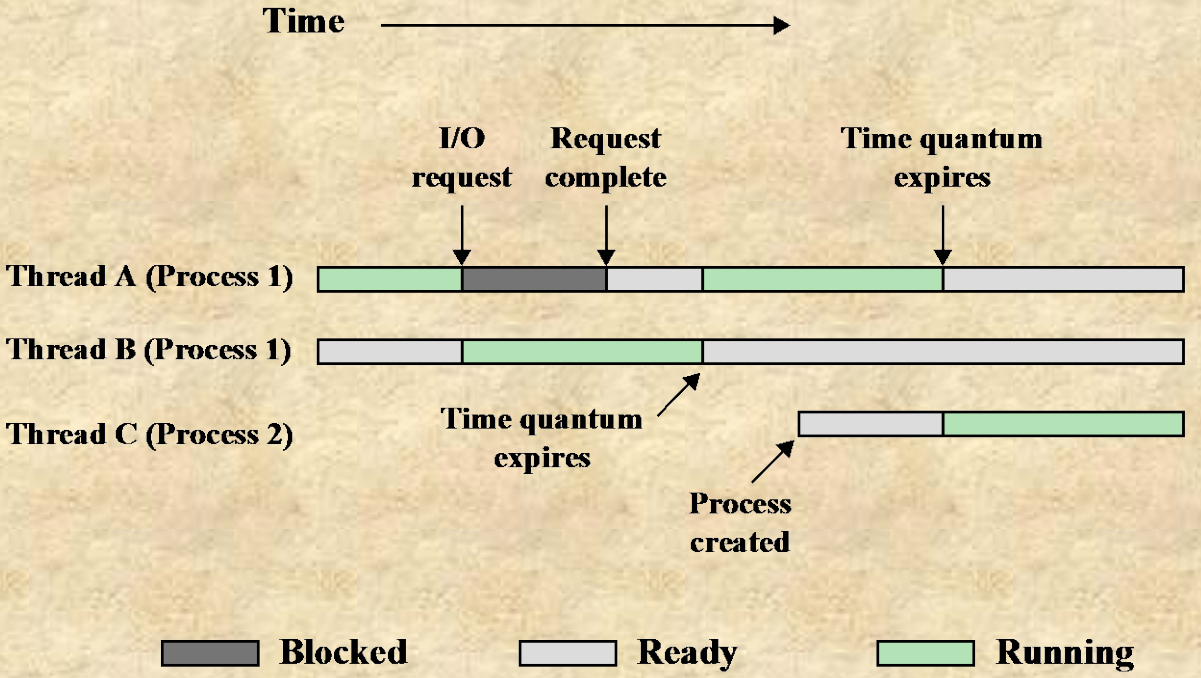


Figure 4.4 Multithreading Example on a Uniprocessor

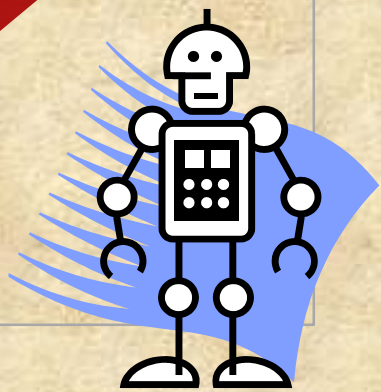
Thread Synchronization

- 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

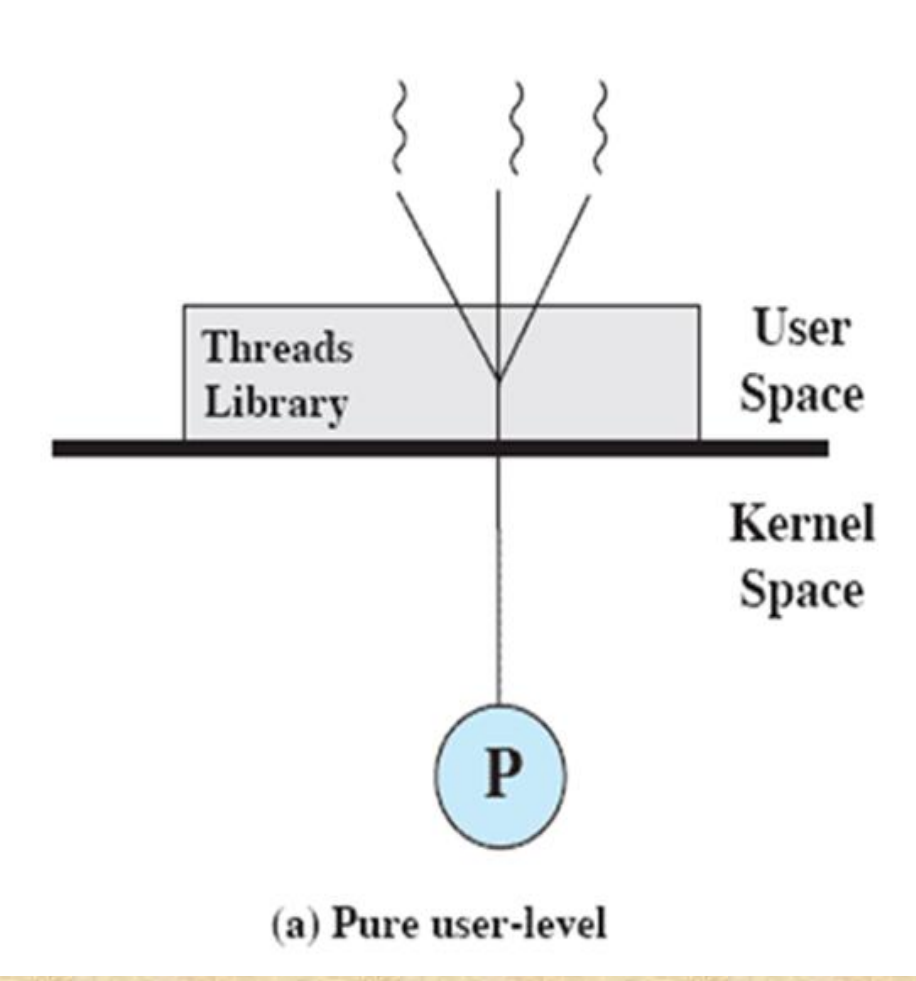
User Level Thread (ULT)

Kernel level Thread (KLT)



User-Level Threads (ULTs)

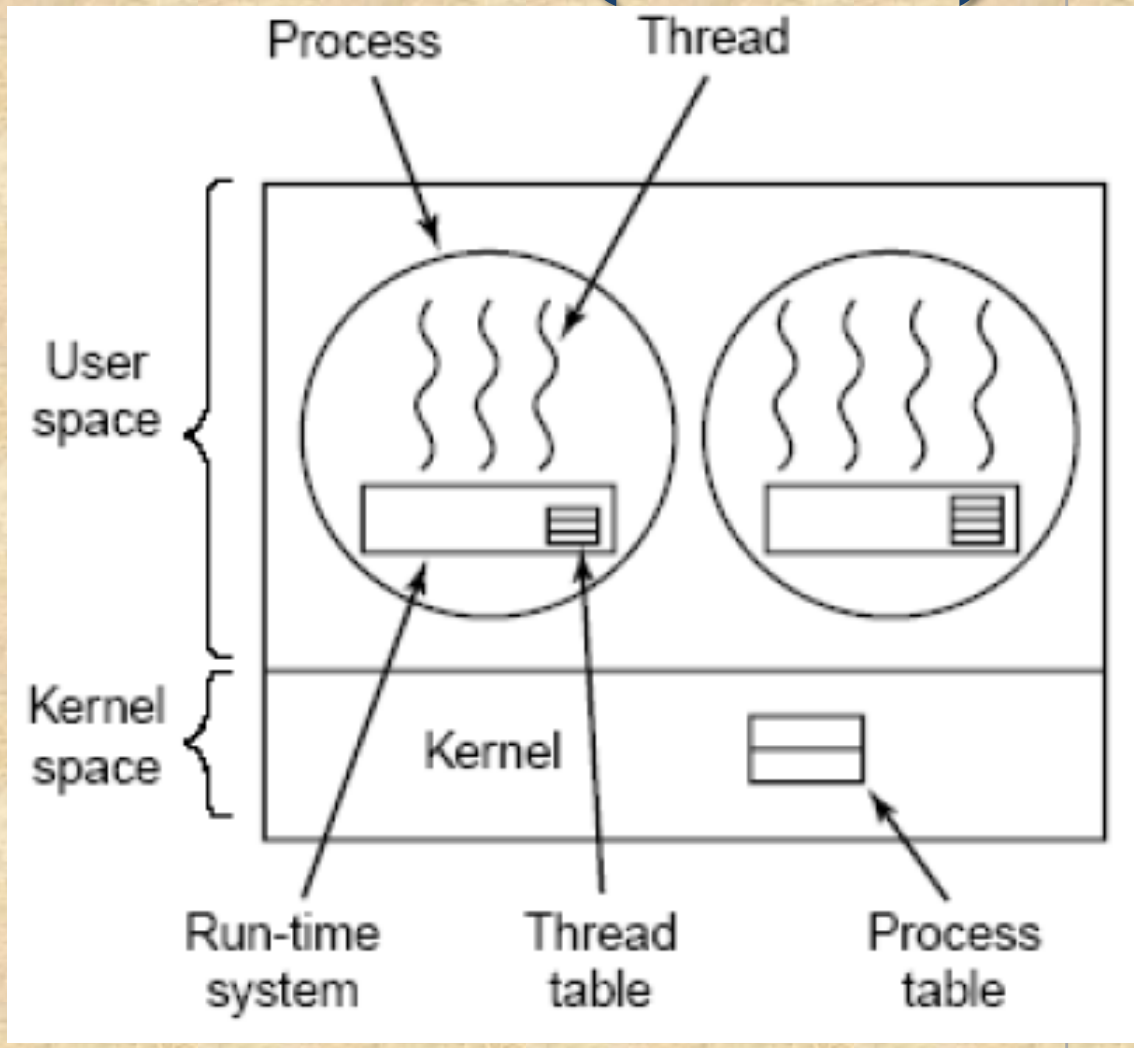
- All thread management is done by the application
- The kernel is not aware of the existence of threads
  - Kernel only knows about processes
  - Each user process manages its own private thread table



(a) Pure user-level

User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads
  - Kernel only knows about processes
  - Each user process manages its own private thread table



Relationships Between ULT States and Process States

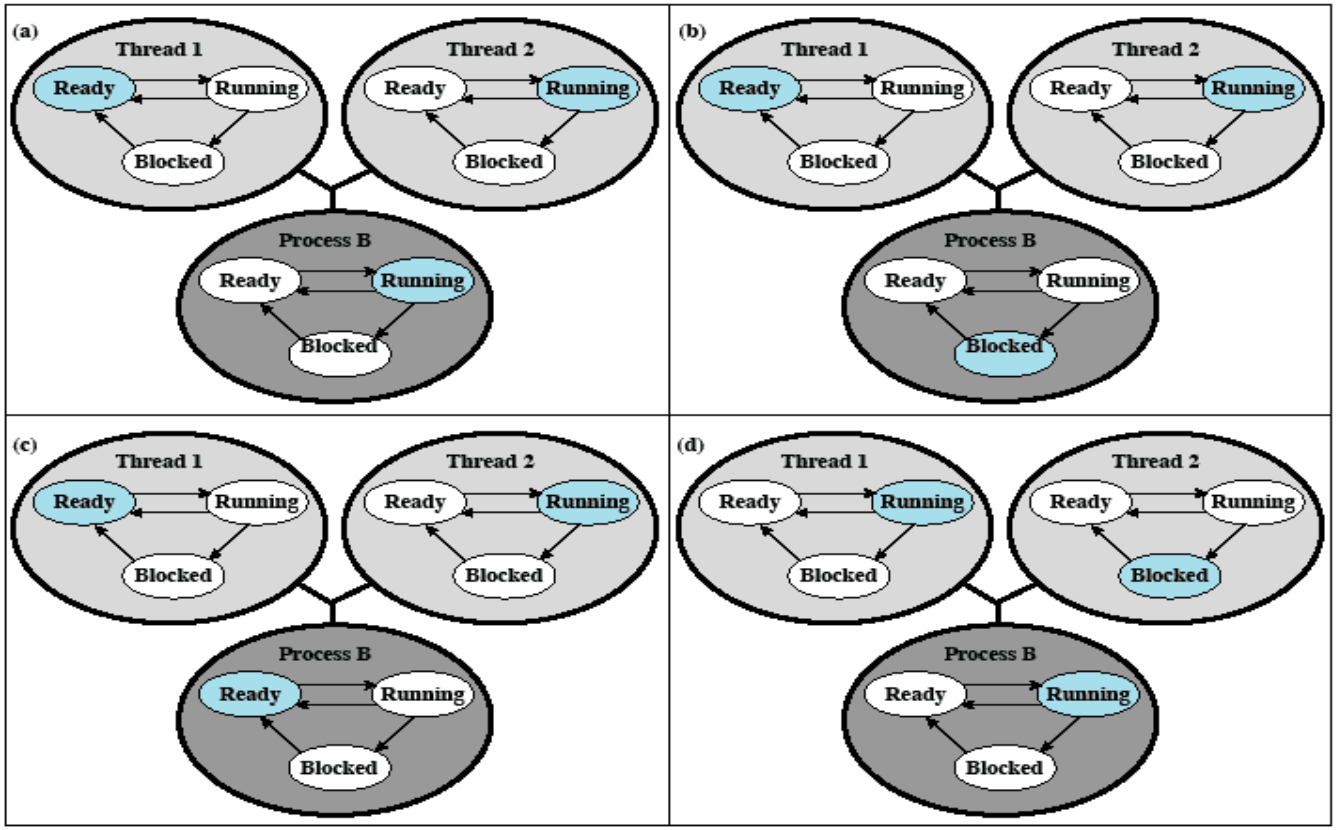
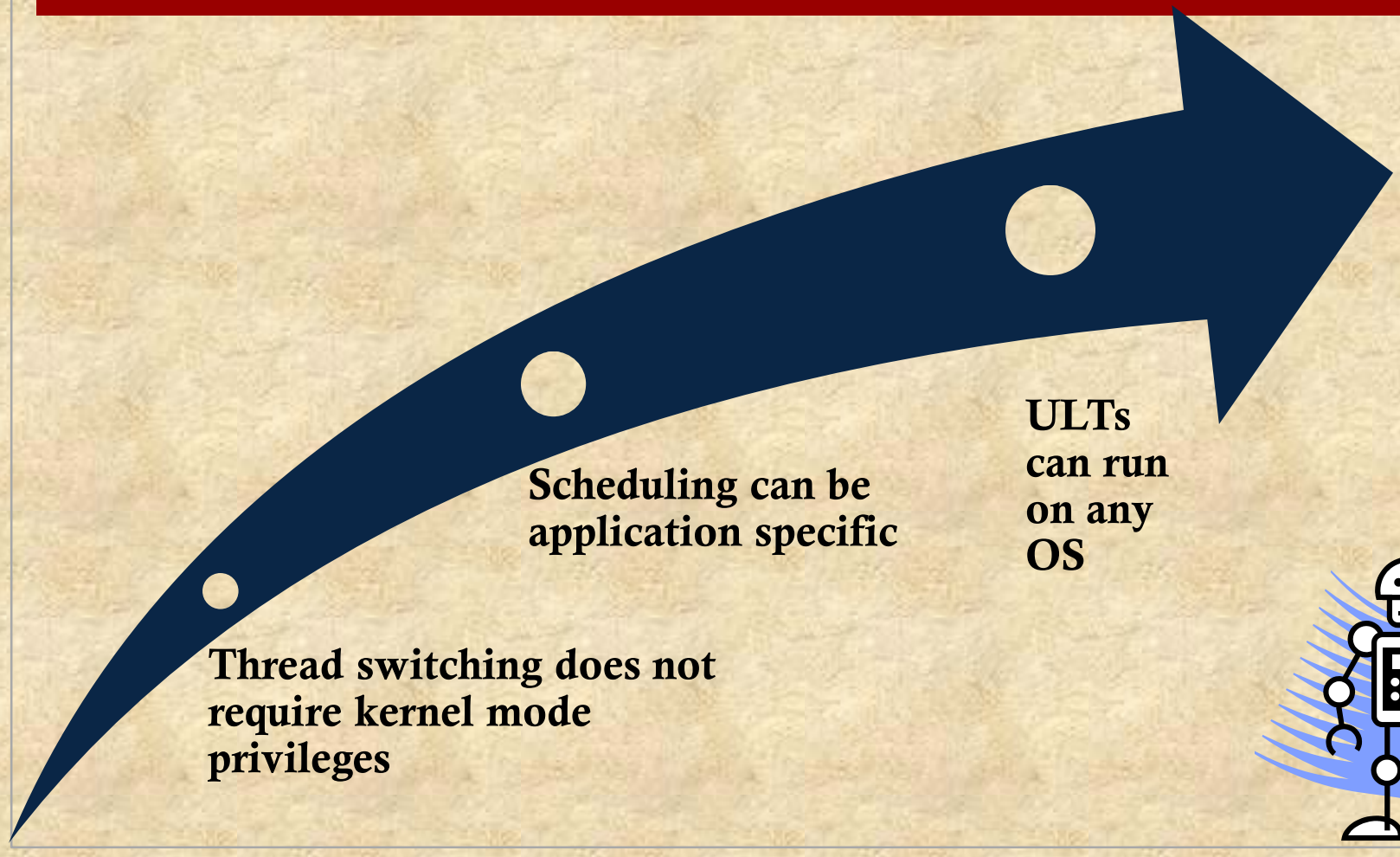


Figure 4.6 Examples of the Relationships between User-Level Thread States and Process States

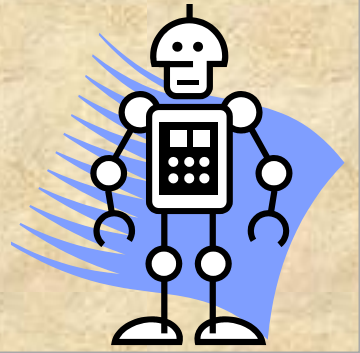
Advantages of ULTs



Thread switching does not require kernel mode privileges

Scheduling can be application specific

ULTs can run on any OS



Disadvantages of ULTs

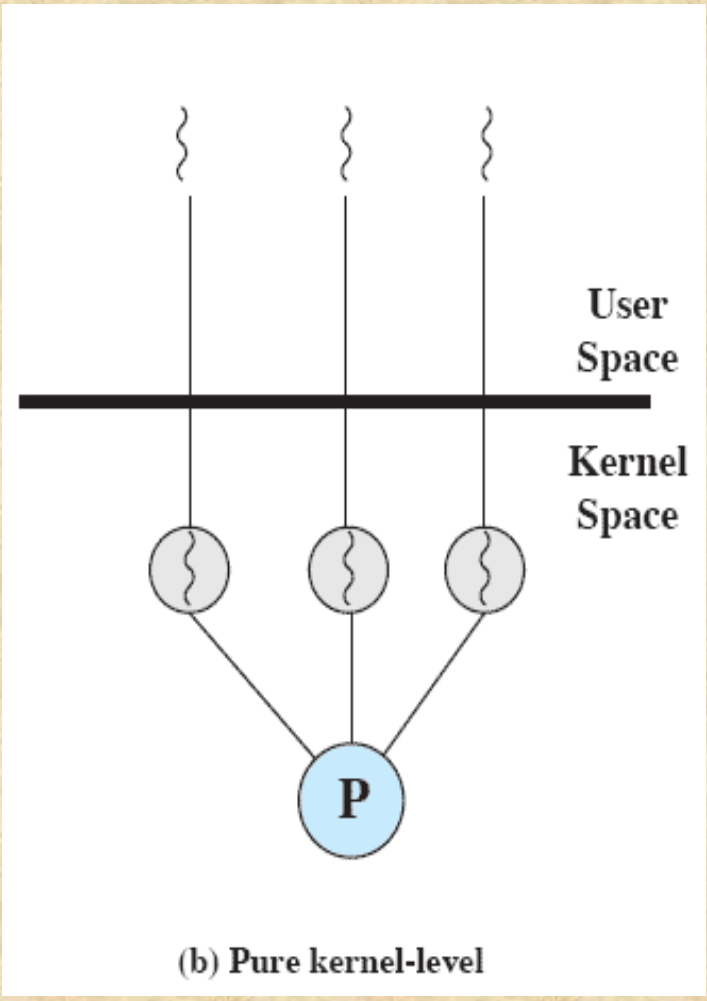
- In a typical OS many system calls are **blocking**
  - as a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked
- In a pure ULT strategy, a multithreaded application **cannot** take advantage of multiprocessing (multi-processor or multi-core)

Overcoming ULT Disadvantages

(1) Jacketing -- converts a blocking system call into a non-blocking system call

(2) Writing an application as multiple processes rather than multiple threads

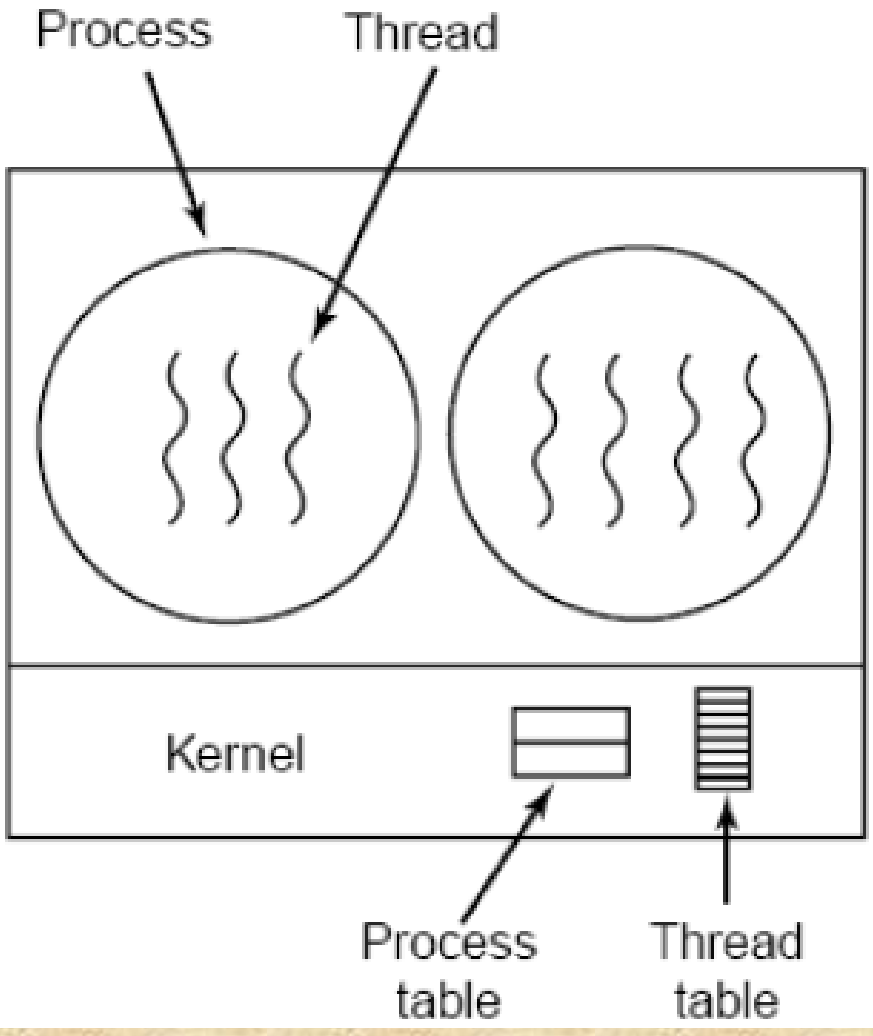
Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- Thread management is done by the kernel
  - no thread management is done by the application
  - Creating and destroying threads are system calls
  - Kernel maintains context information for the process and the threads
    - API to kernel thread facility
  - MS Windows is an example of this approach

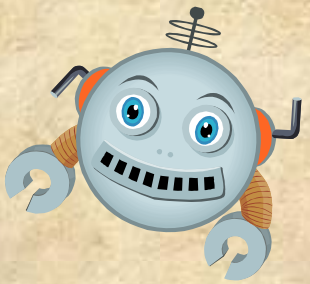
Kernel-Level Threads (KLTs)



- Thread management is done by the kernel
  - no thread management is done by the application
  - Creating and destroying threads are system calls
  - Kernel maintains context information for the process and the threads
    - API to kernel thread facility
  - MS 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 can be multithreaded

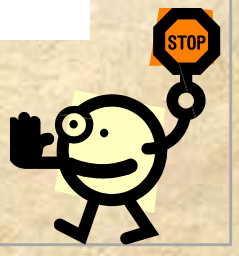


Disadvantage of KLTs

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

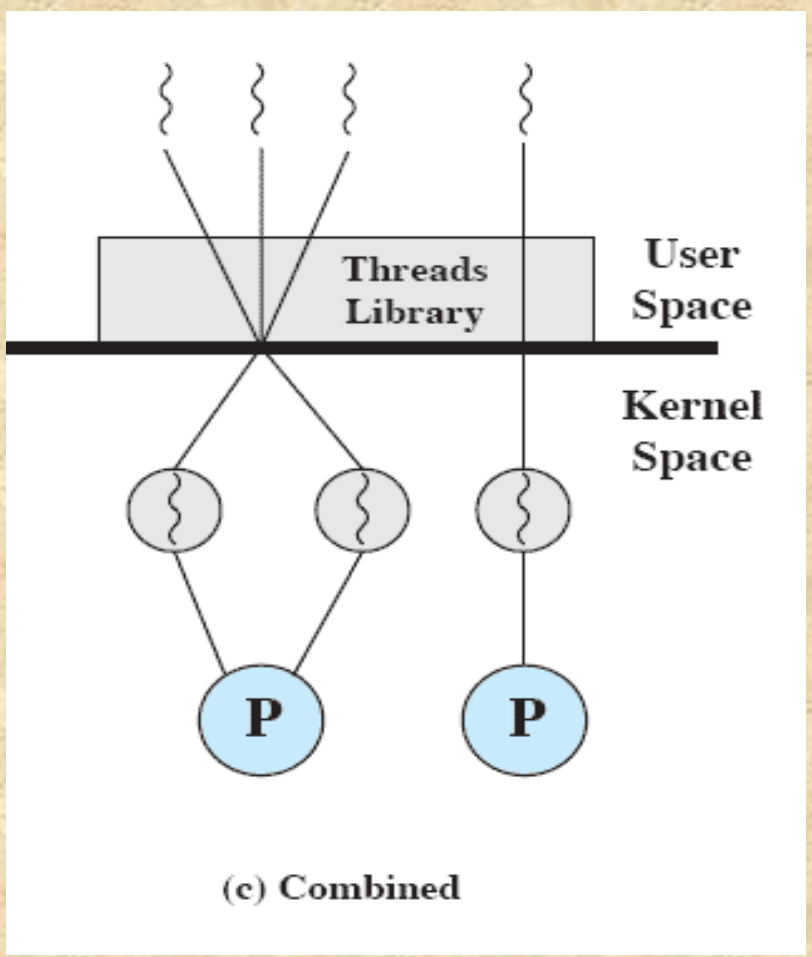
Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1 Thread and Process Operation Latencies (μs)



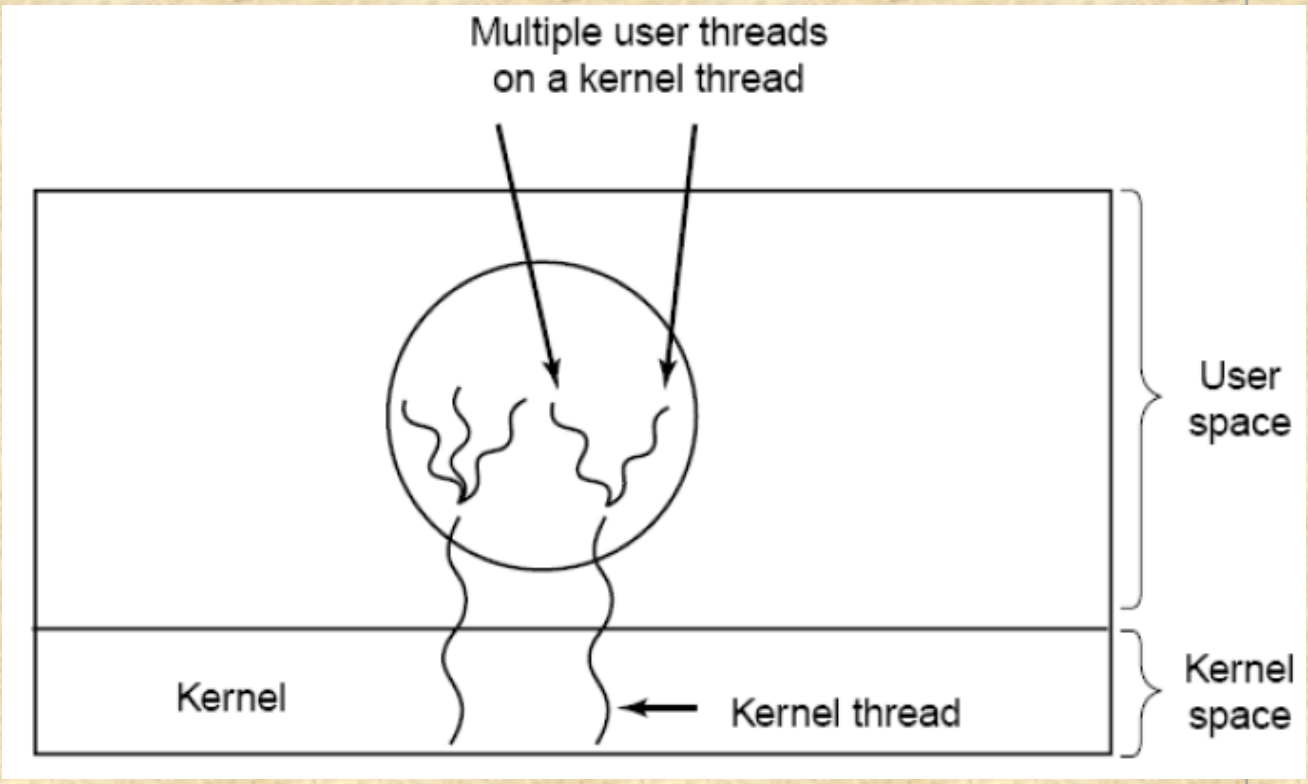
Combined Approaches

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



Combined Approaches

- Solaris is an example
- Multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs



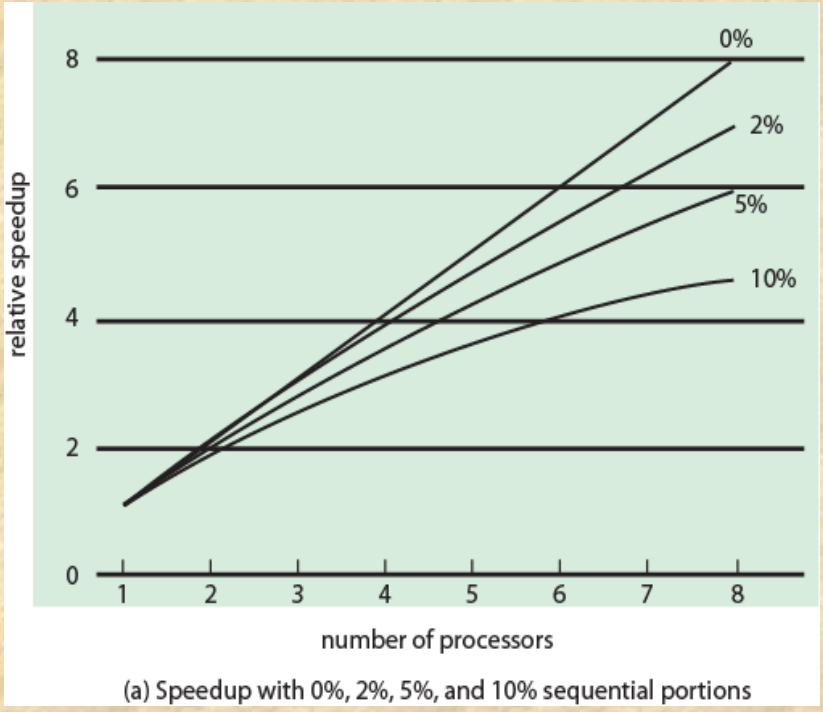
Relationship Between Threads and Processes

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

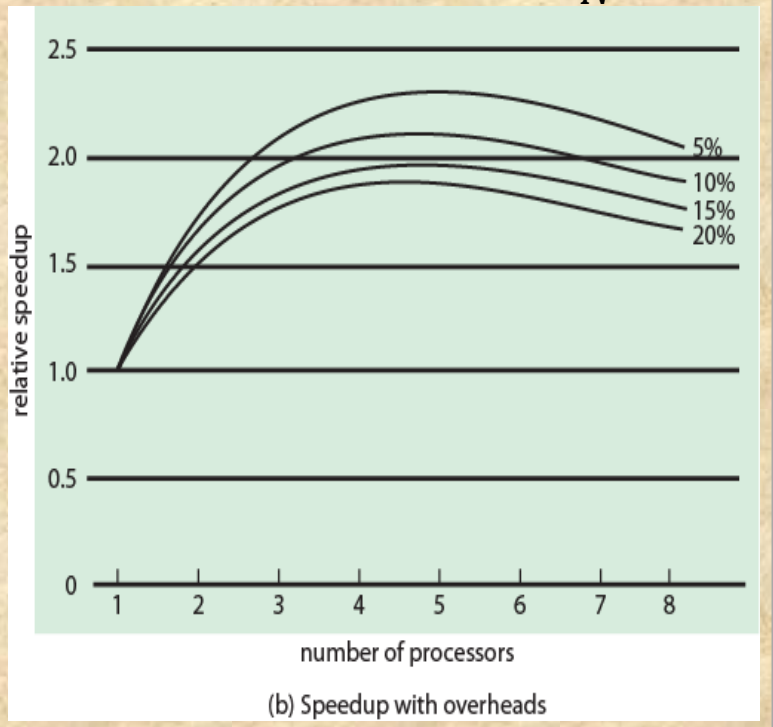
Performance Effect of Multiple Cores

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} = \frac{1}{(1-f) + \frac{f}{N}}$$



(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

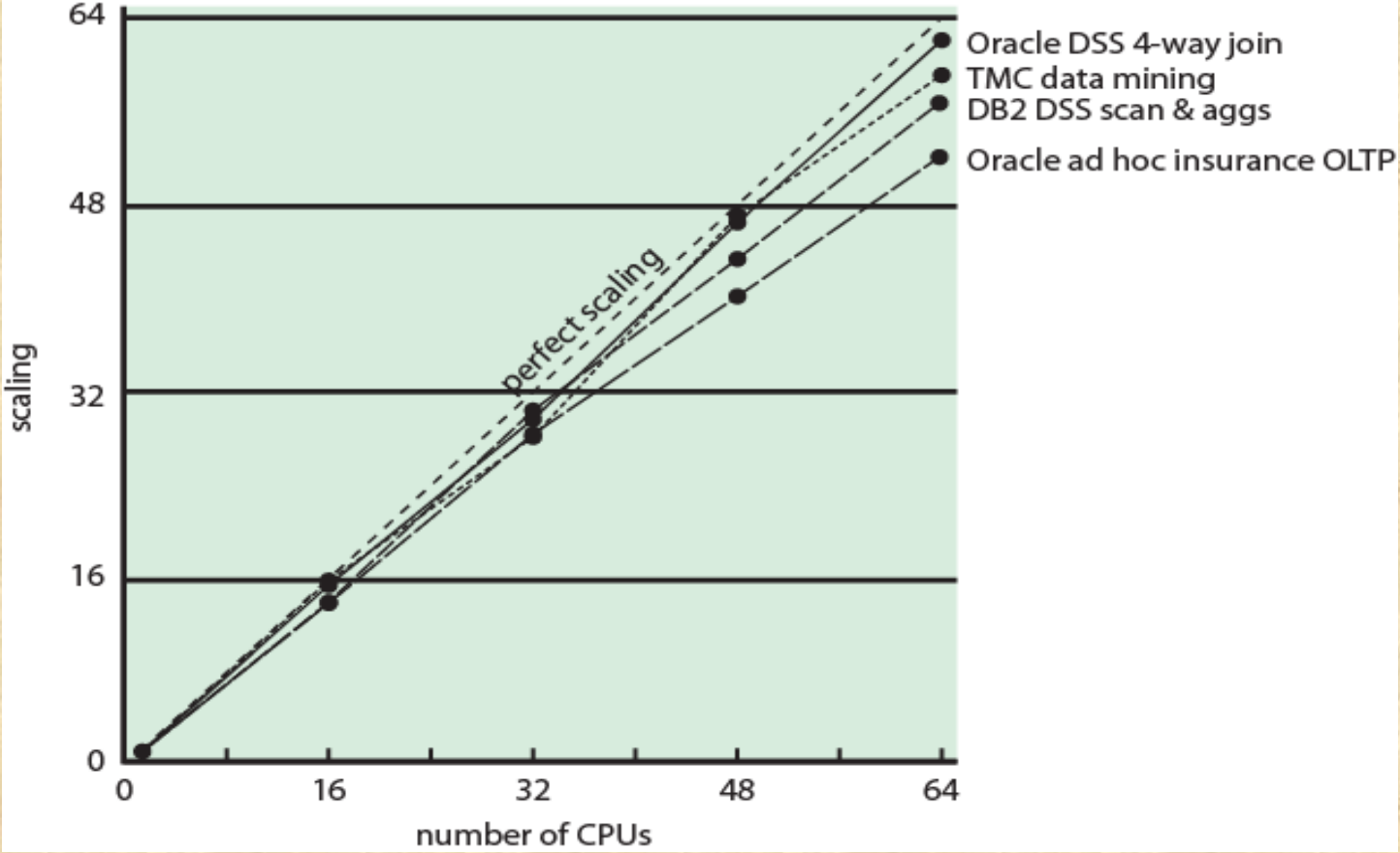


Figure 4.8 Scaling of Database Workloads on Multiple Processor Hardware

Applications That Benefit

- 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
- Multiinstance applications
  - multiple instances of the application in parallel

Valve Game Software

- Valve has developed the Source engine
  - an animation engine used in its games and licensed for other developers
  - Recently rewritten to use multi-threading
  - higher-level threads spawn lower-level threads as needed
  - Different threads responsible for rendering different objects as viewed from multiple angles

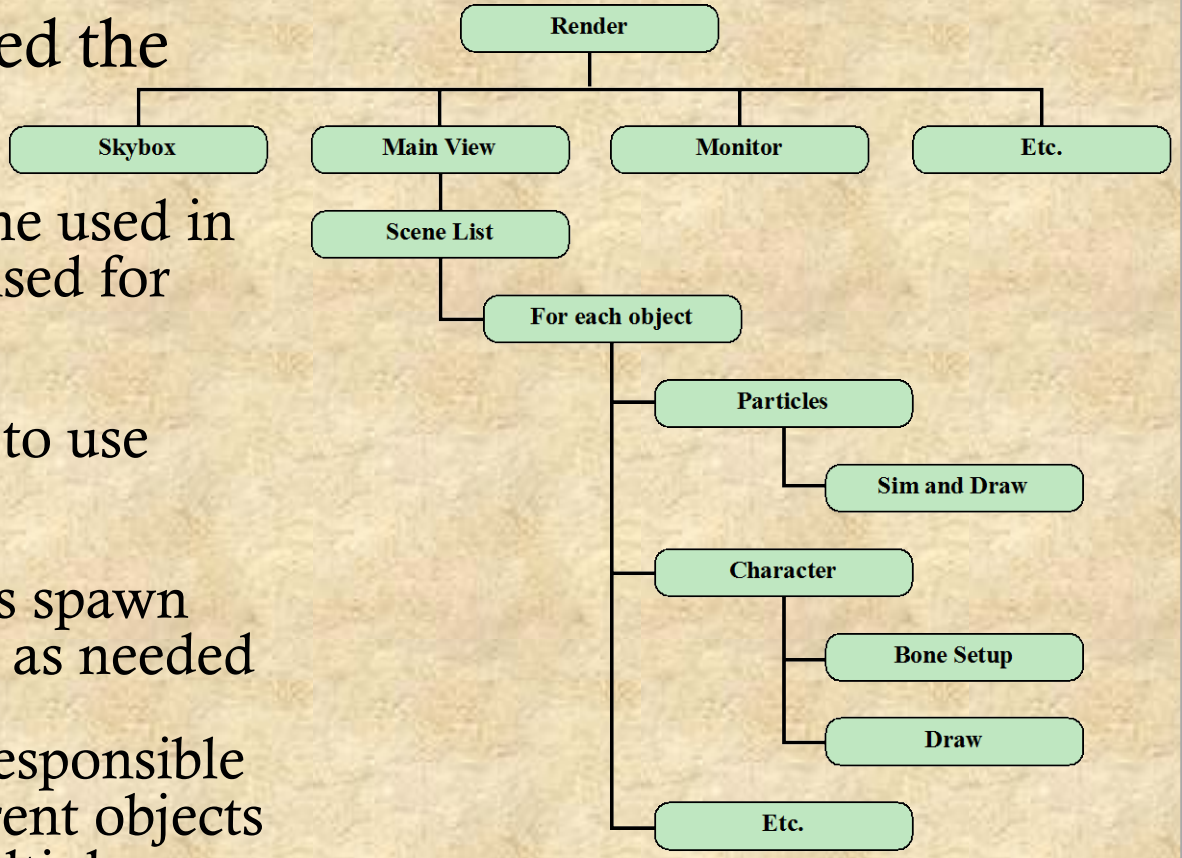


Figure 4.9 Hybrid Threading for Rendering Module

Windows 8 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

Changes in Windows 8

- Changes the traditional Windows approach to managing background tasks and application lifecycles
- Developers are now responsible for managing the state of their individual applications
- In the new Metro interface Windows 8 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 8 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
- Background apps only receive 1 processor second per processor hour
  - Background apps may never actually run

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 process
- an executable process may contain one or more threads
- both processes and thread objects have built-in synchronization capabilities

Relationship Between Process and Resource

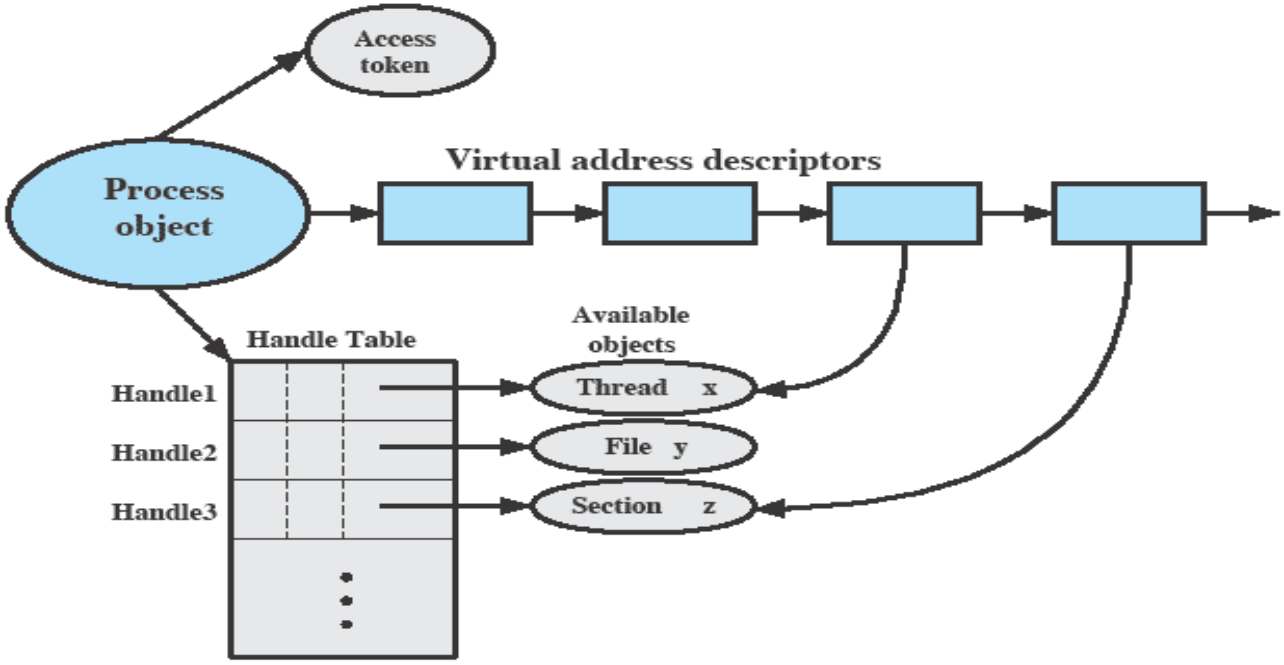


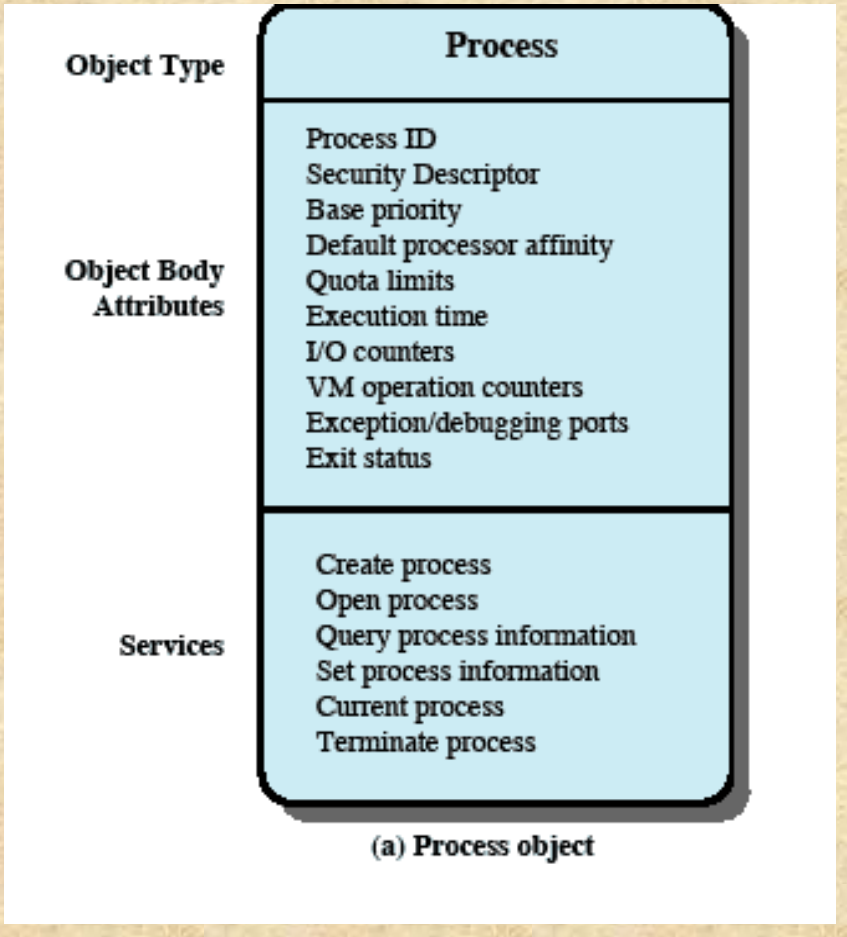
Figure 4.10 A Windows Process and Its Resources

Process and Thread Objects

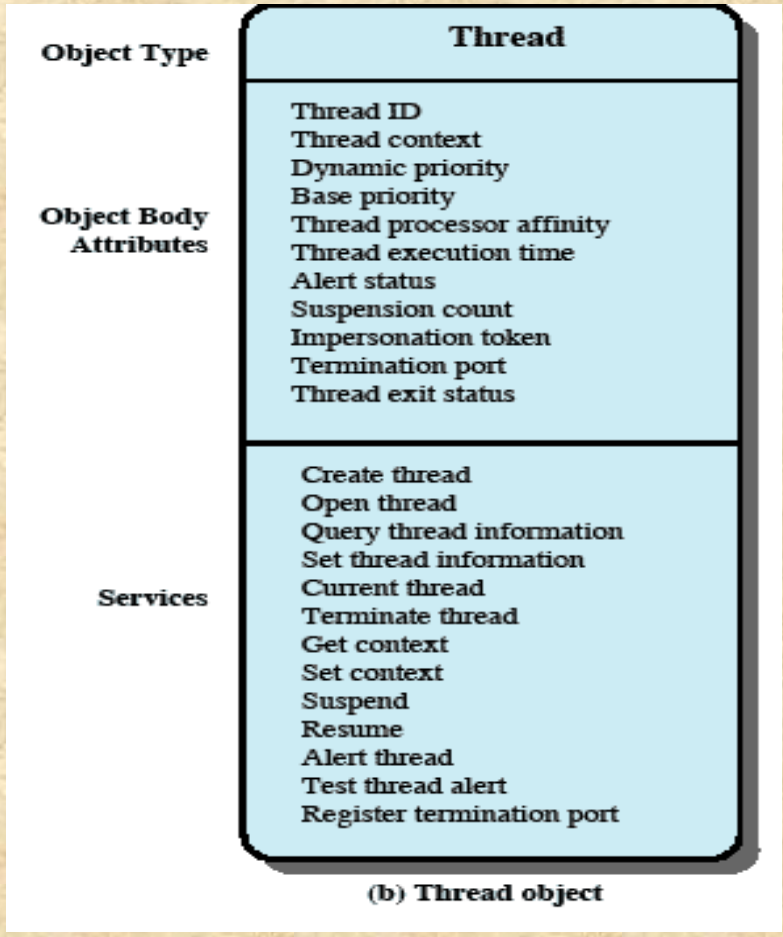
Windows makes use of two types of process-related objects:

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

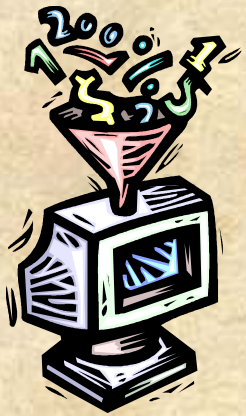
Windows Process and Thread Objects



(a) Process object



(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

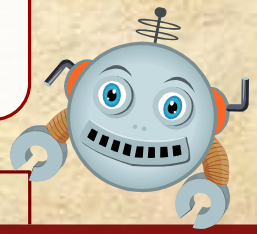
Windows Thread Object Attributes

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 b een 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

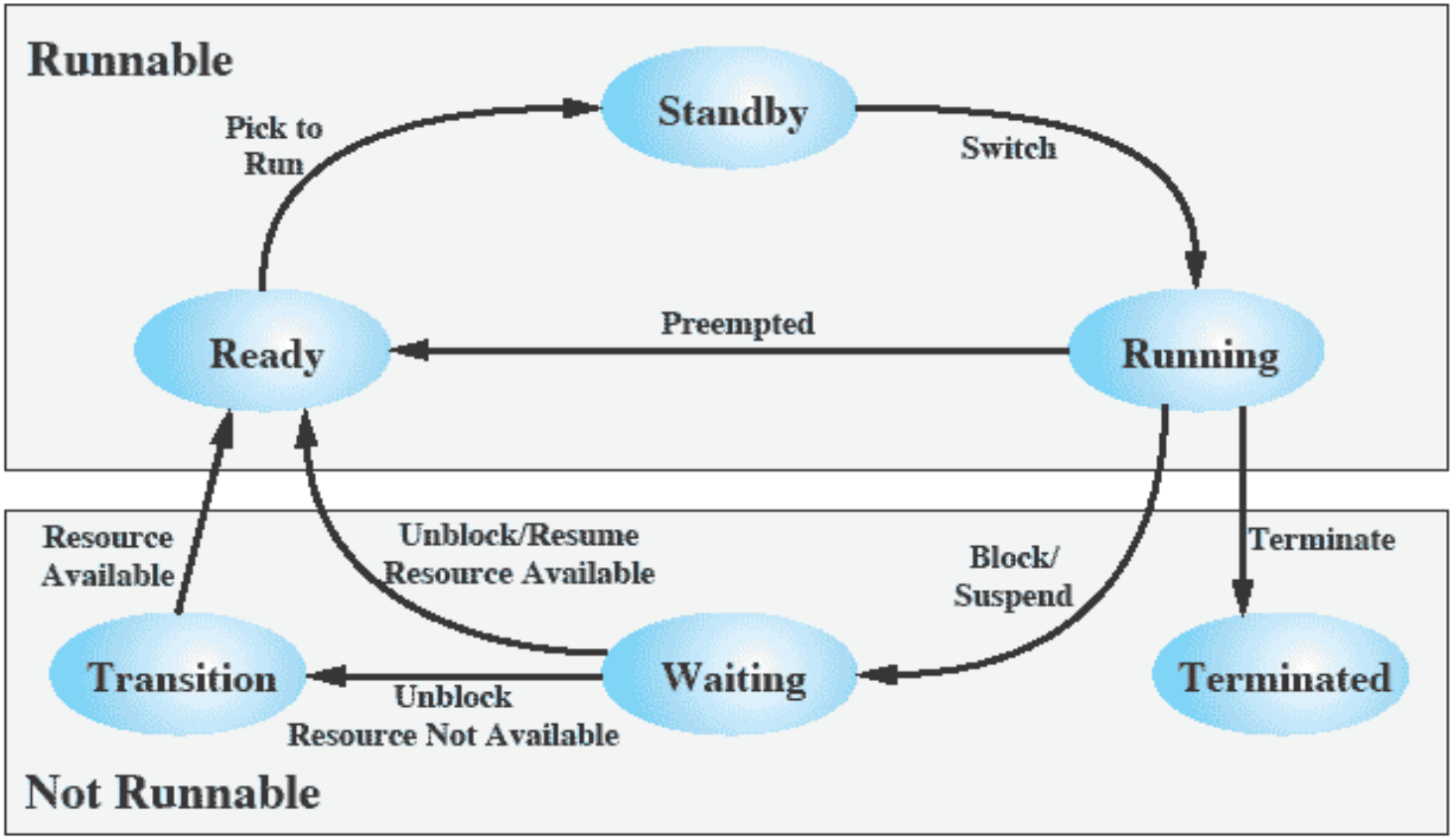
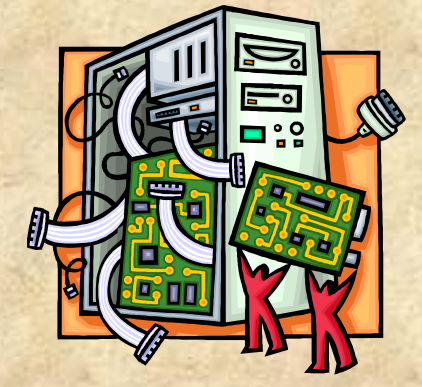


Figure 4.12 Windows Thread States

Symmetric Multiprocessing Support (SMP)

Threads of any process can run on any processor

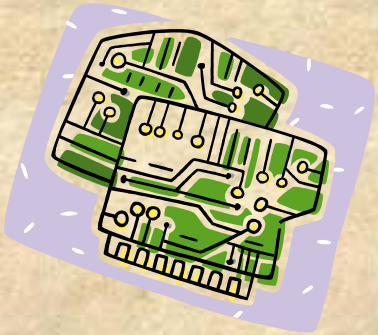


Soft Affinity

- 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

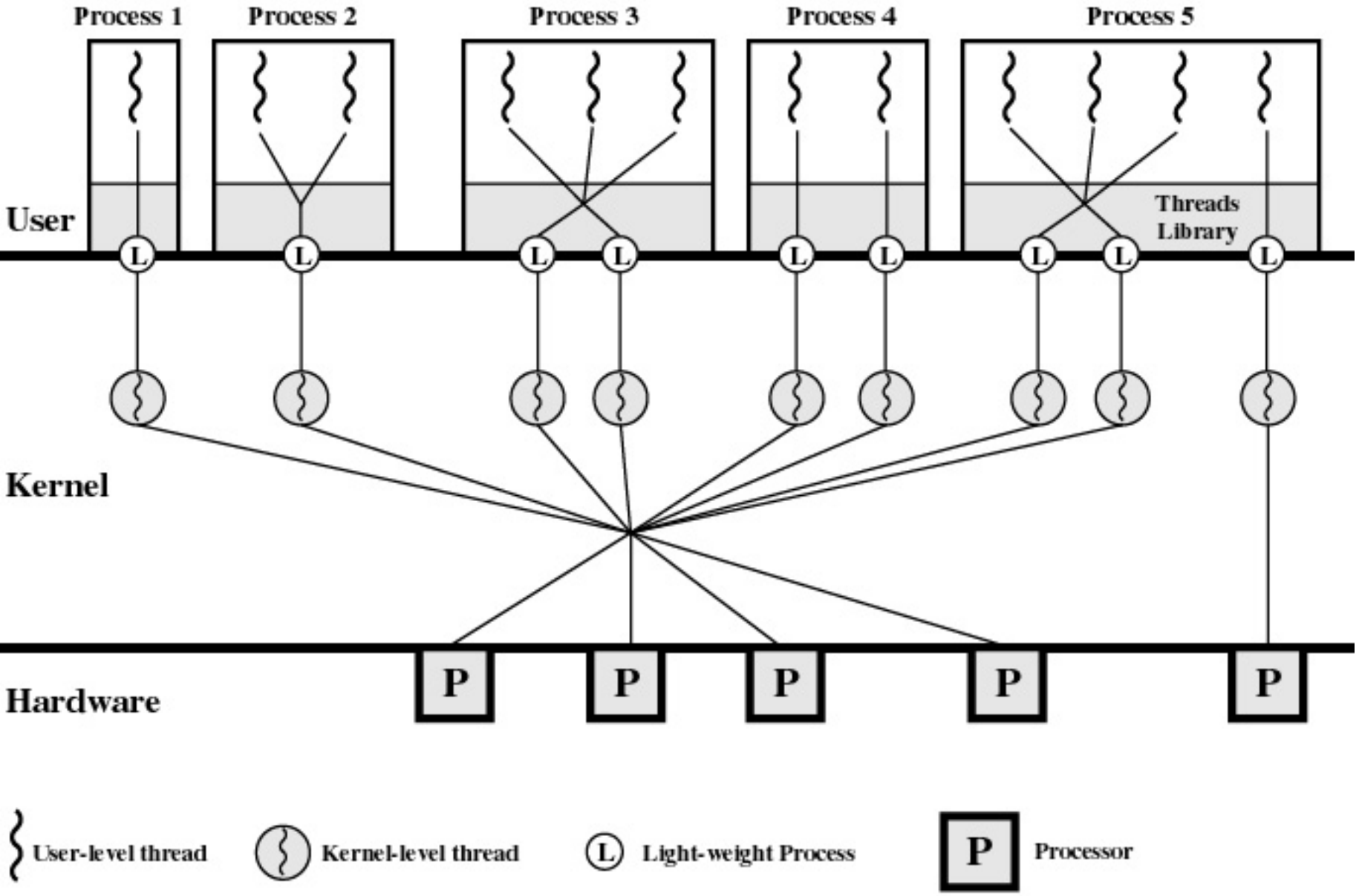


Figure 4.15 Solaris Multithreaded Architecture Example

Processes and Threads in Solaris

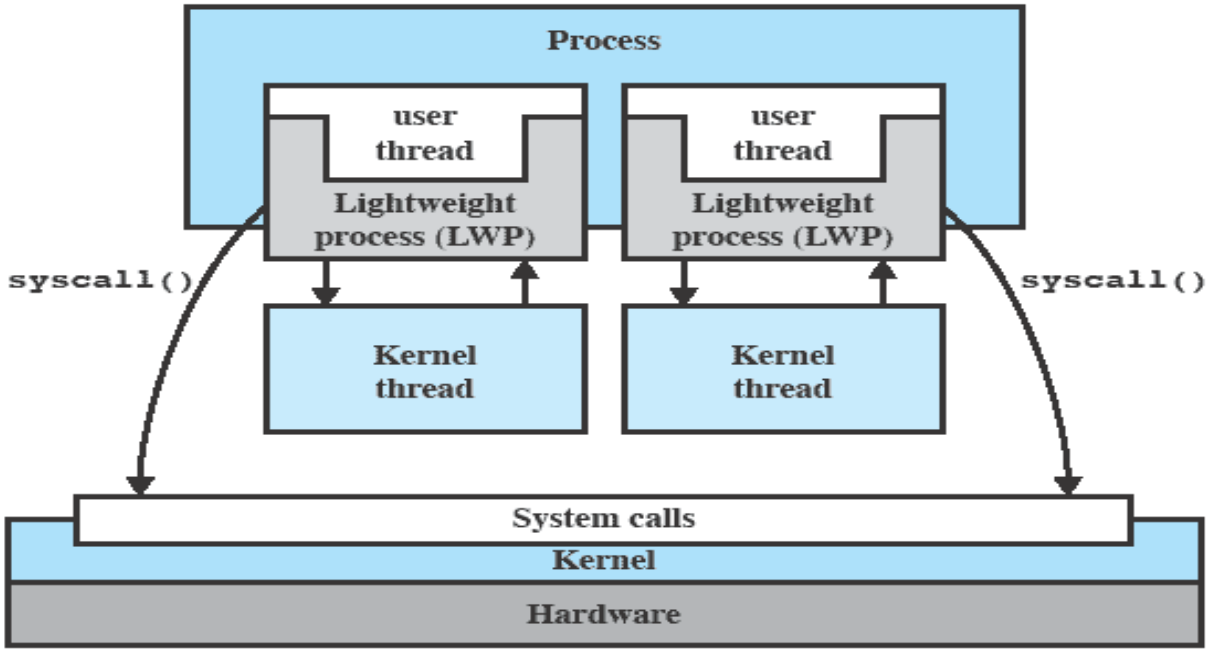


Figure 4.13 Processes and Threads in Solaris [MCDO07]

Traditional Unix vs Solaris

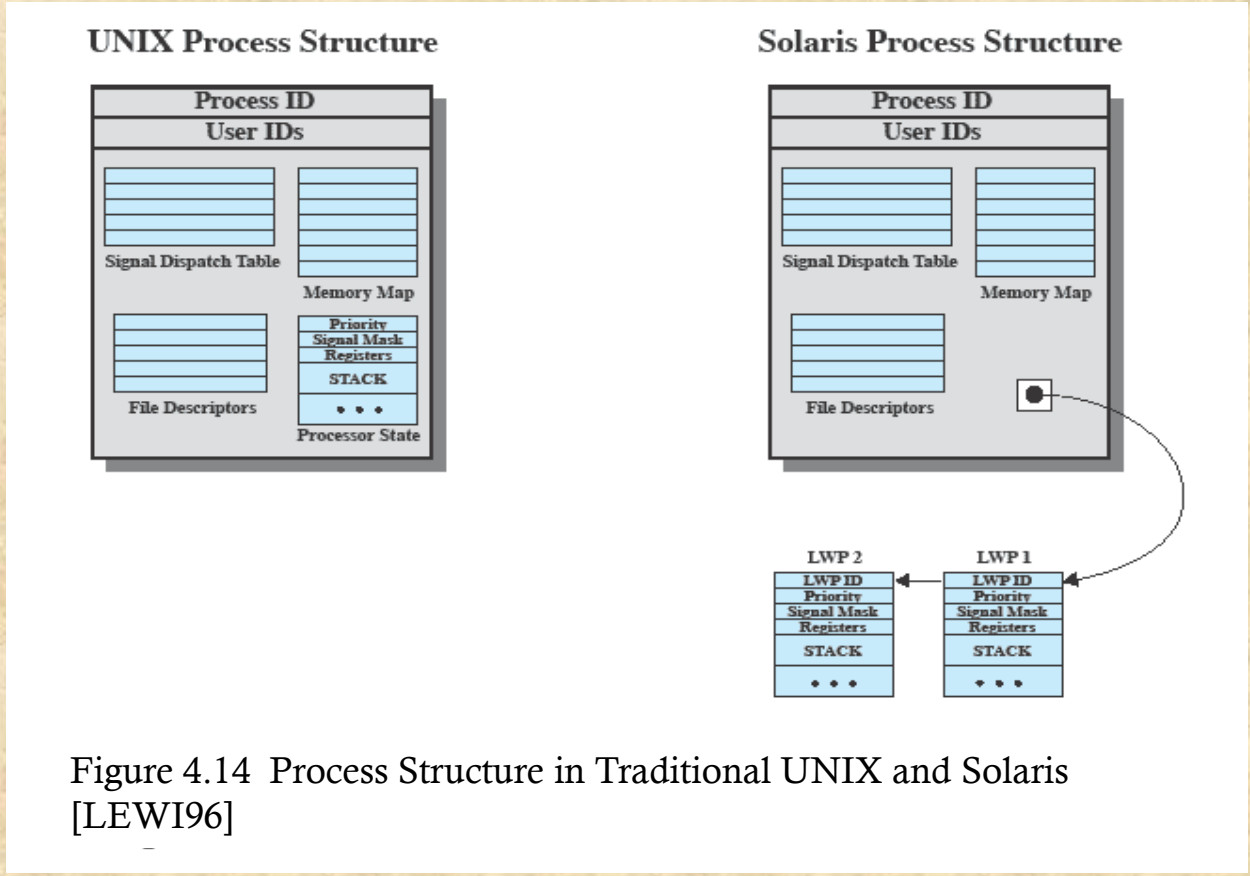


Figure 4.14 Process Structure in Traditional UNIX and Solaris [LEWI96]

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

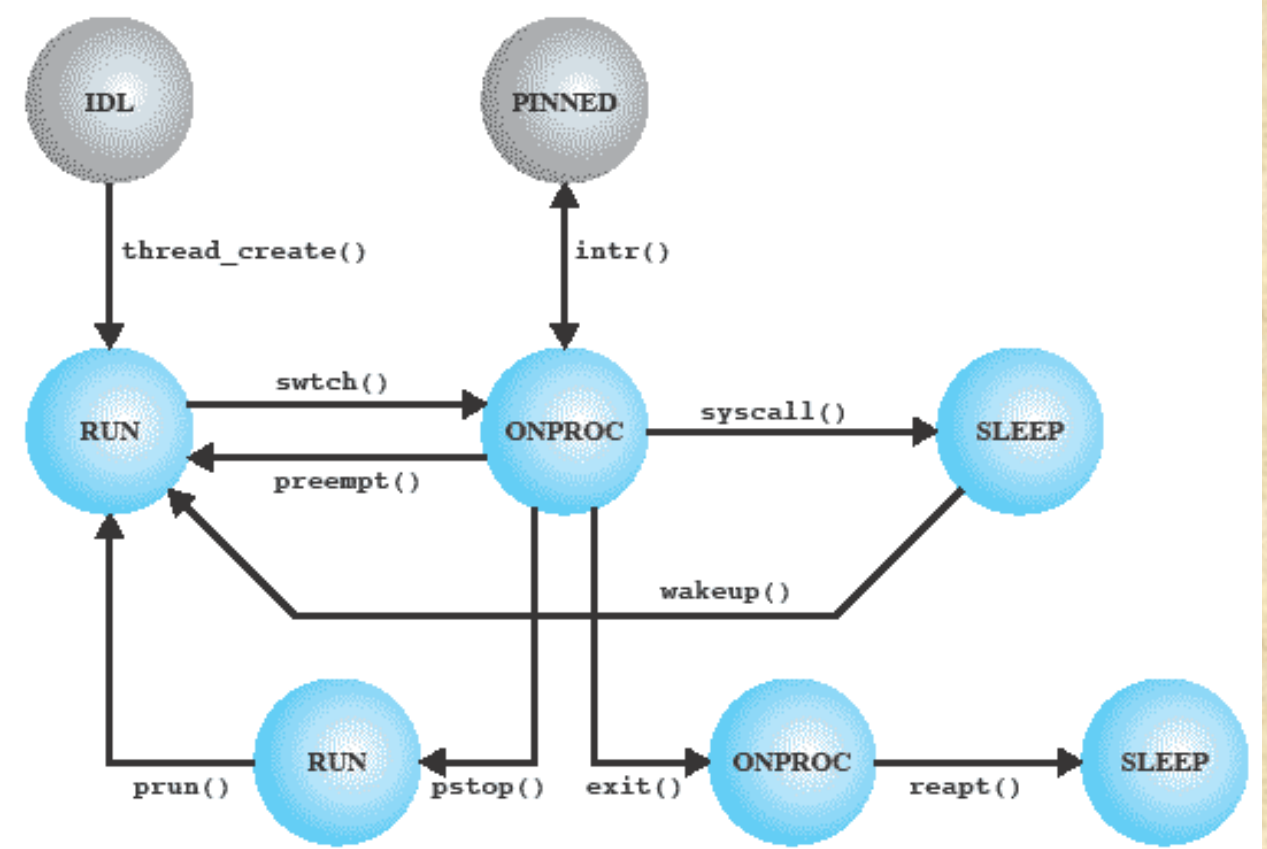


Figure 4.15 Solaris Thread States

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

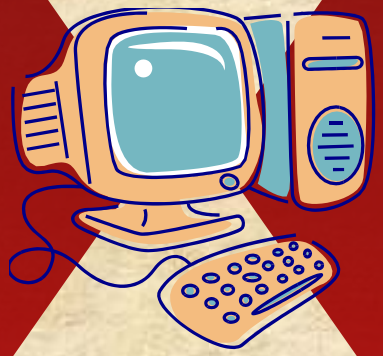
- synchronized by preventing their handling for a period of time

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

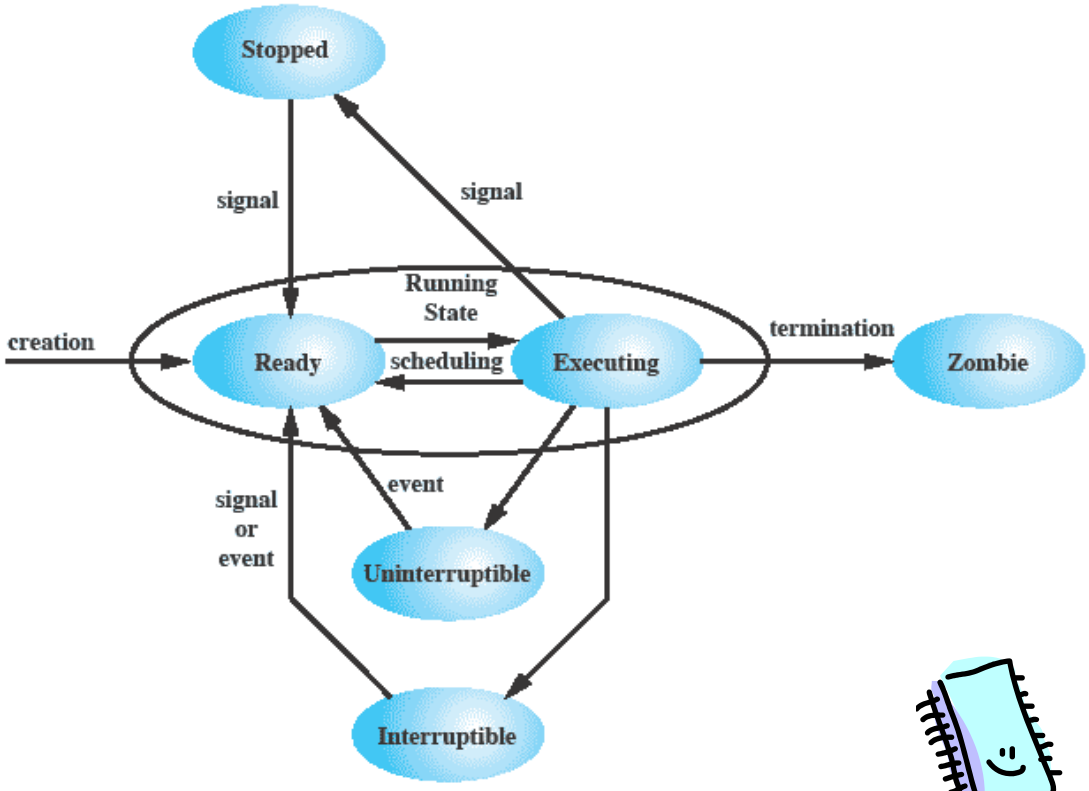


Figure 4.16 Linux Process/Thread Model

Linux Threads

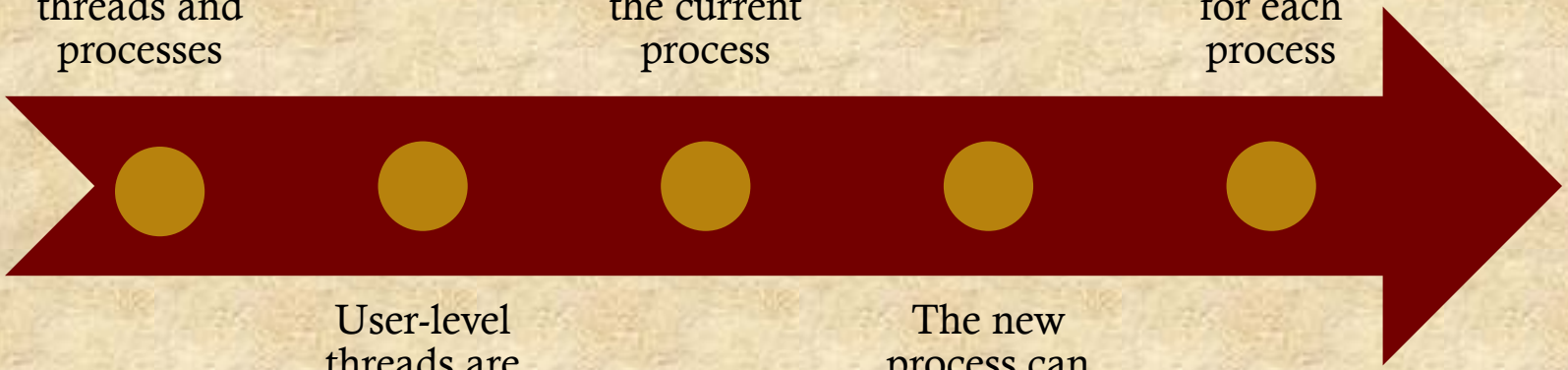
Linux does not recognize a distinction between threads and processes

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 Namespaces

- A namespace enables a process to have a different view of the system than other processes that have other associated namespaces
- One of the overall goals is to support the implementation of control groups, cgroups), a tool for lightweight virtualization that provides a process or group of processes with the illusion that they are the only processes on the system
- There are currently six namespaces in Linux
  - mnt
  - pid
  - net
  - ipc
  - uts
  - user

Linux Clone () Flags

CLONE_CLEARID	Clear the task ID.
CLONE_DETACHED	The parent does not want a SIGCHLD signal s ent 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 schedu led for execution until the child invokes the *execve()* system call.
CLONE_VM	Shares the address space (memory descriptor and all page tables).

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 and even by other applications
- Four types of components:
  - Activities
  - Services
  - Content providers
  - Broadcast receivers

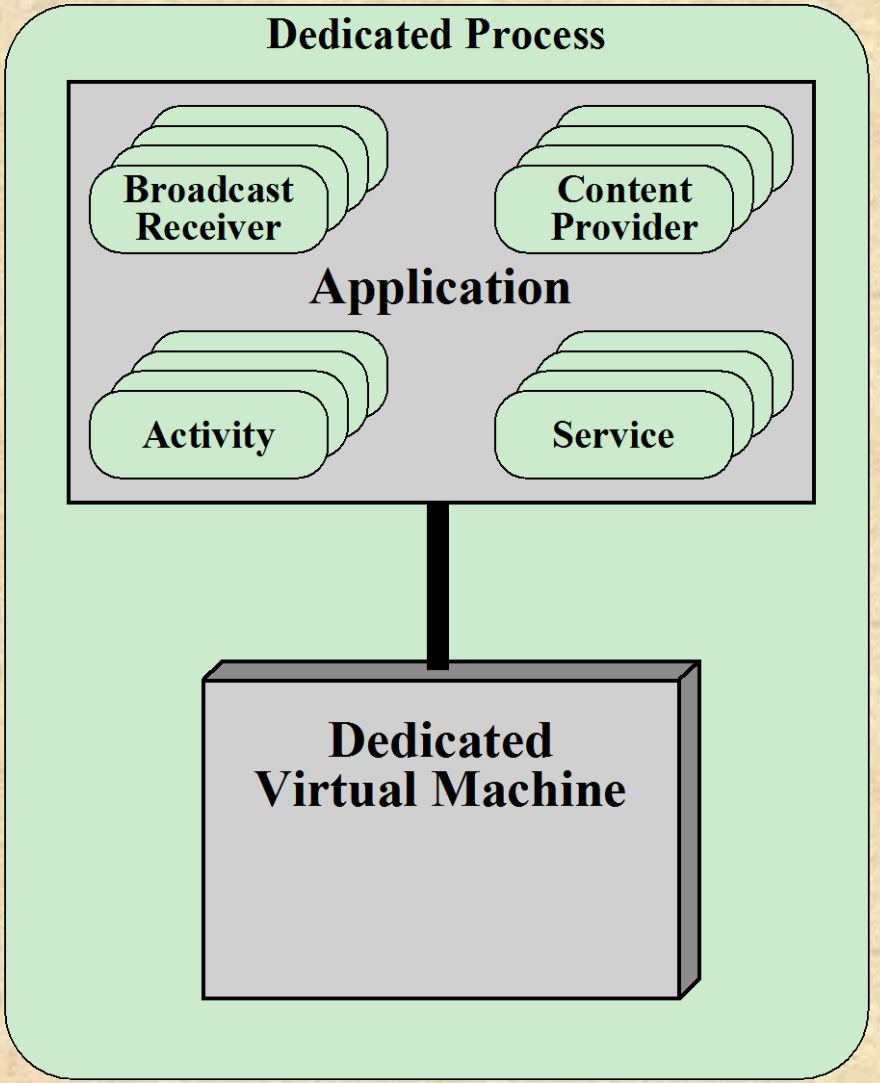


Figure 4.16 Android Application

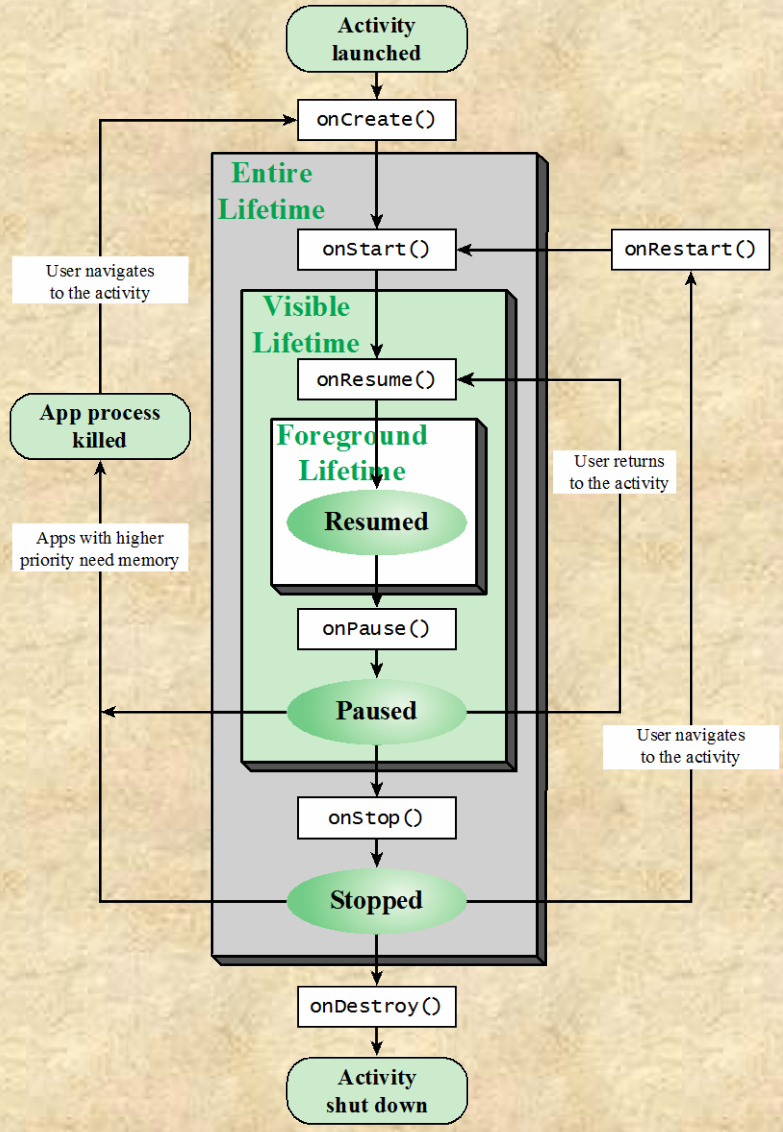
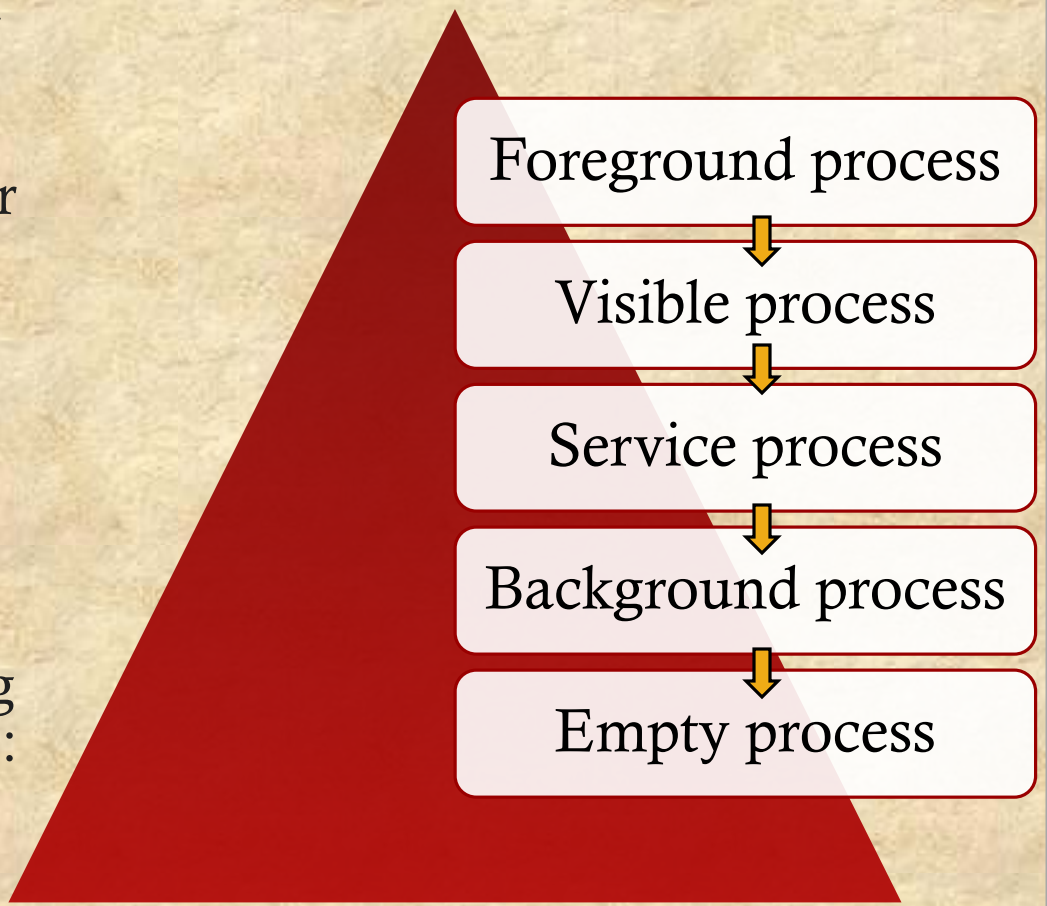


Figure 4.17 Activity State Transition Diagram

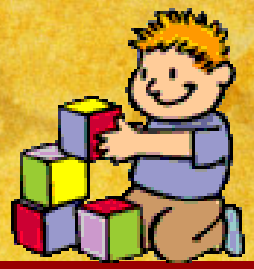
Processes and Threads

- 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:



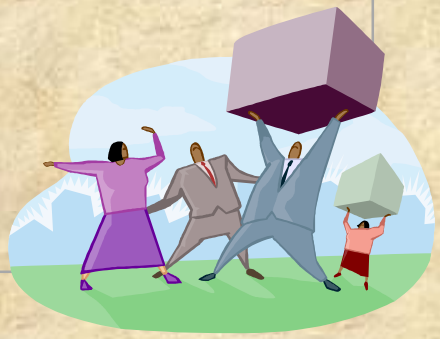
Mac OS X Grand Central Dispatch (GCD)

- 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



Block

- A simple extension to a language
- A block defines a self-contained unit of work
- Enables the programmer to encapsulate complex functions
- 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

- 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

Summary

- Processes and threads
  - Multithreading
  - Thread functionality
- Types of threads
  - User level and kernel level threads
- Multicore and multithreading
- Windows 8 process and thread management
  - Changes in Windows 8
  - 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