

Processes and Threads

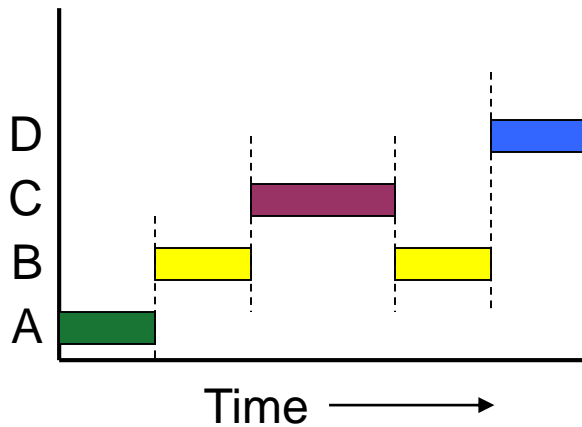
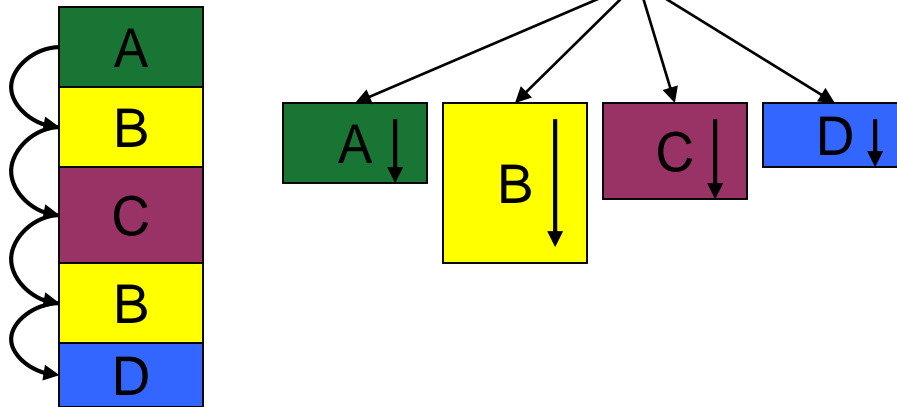
What is a process?

- Another view of process
 - Code, data, and stack
 - Usually (but not always) has its own address space
 - Program state
 - CPU registers
 - Program counter (current location in the code)
 - Stack pointer
- Only one process can be running in the CPU at any given time!

The process model

Single PC
(CPU's point of view)

Multiple PCs
(process point of view)



- Multiprogramming of four programs
- Conceptual model
 - 4 independent processes
 - Processes run sequentially
- Only one program active at any instant!
 - That instant can be very short...

When is a process created?

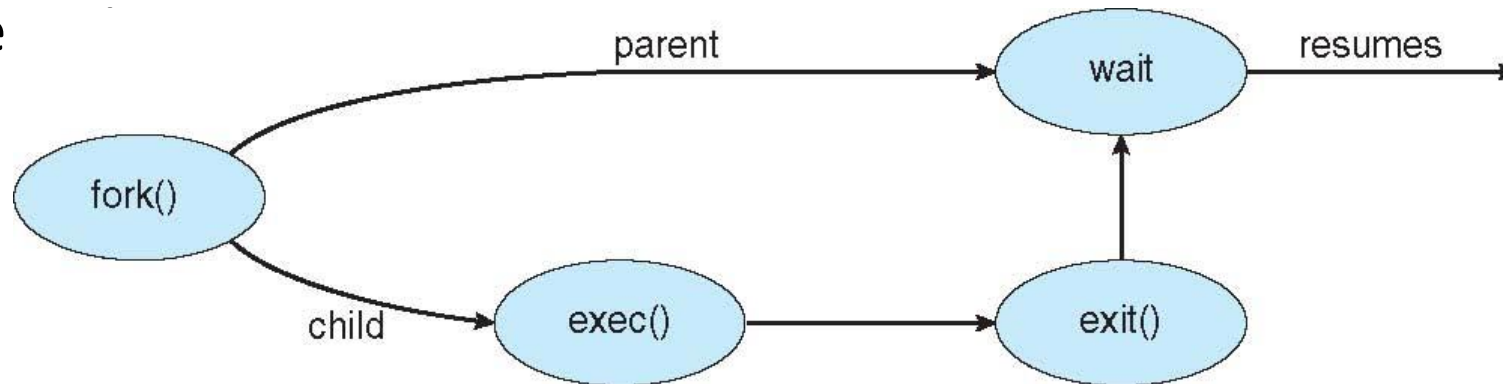
- Processes can be created in two ways
 - System initialization: one or more processes created when the OS starts up
 - Execution of a process creation system call: something explicitly asks for a new process
- System calls can come from
 - User request to create a new process (system call executed from user shell)
 - Already running processes
 - User programs
 - System daemons

Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier (pid)**
- Resource sharing options
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- Execution options
 - Parent and children execute concurrently
 - Parent waits until children terminate

Process Creation (Cont.)

- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- UNIX examples
 - **fork()** system call creates new process
 - **exec()** system call used after a **fork()** to replace the process' memory space



When do processes end?

- Conditions that terminate processes can be
 - Voluntary
 - Involuntary
- Voluntary
 - Normal exit
- Involuntary
 - Fatal error (only sort of involuntary)
 - Killed by another process

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **exit()** system call.
 - Returns status data from child to parent (via **wait()**)
 - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **abort()** system call. Some reasons for doing so:
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
 - **cascading termination.** All children, grandchildren, etc. are terminated.
 - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the **wait()** system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status) ;
```

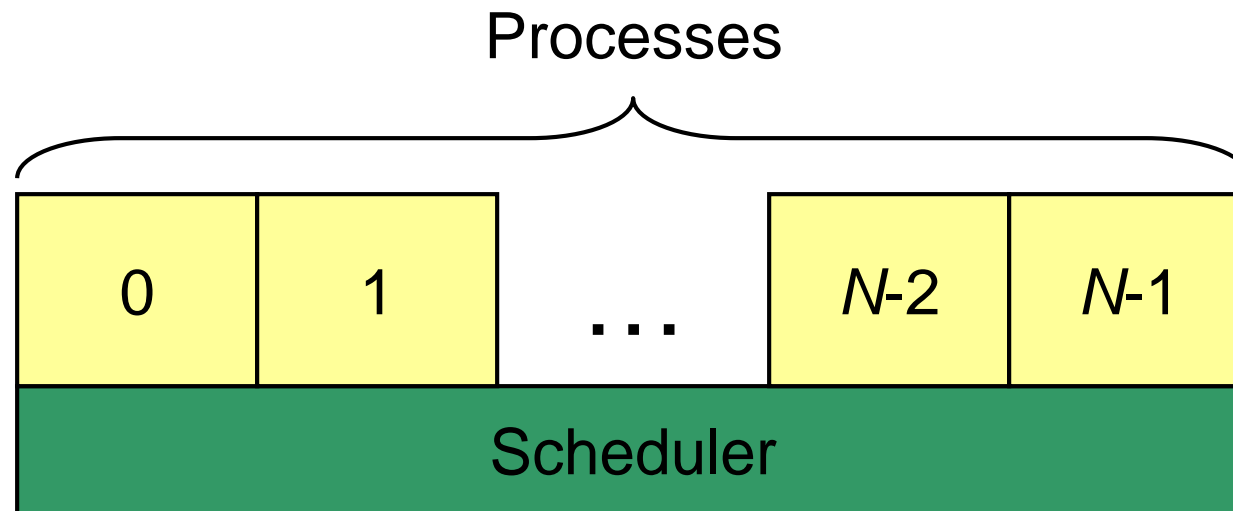
- If no parent waiting (did not invoke **wait()**) process is a **zombie**
- If parent terminated without invoking **wait**, process is an **orphan**

Process hierarchies

- Parent creates a child process
 - Child processes can create their own children
- Forms a hierarchy
 - UNIX calls this a “process group”
 - If a process exits, its children are “inherited” by the exiting process’s parent
- Windows has no concept of process hierarchy
 - All processes are created equal

Processes in the OS

- Two “layers” for processes
- Lowest layer of process-structured OS handles interrupts, scheduling
- Above that layer are sequential processes
 - Processes tracked in the *process table*
 - Each process has a *process table entry*



What's in a process table entry?

May be
stored
on stack {

Process management

Registers
Program counter
CPU status word
Stack pointer
Process state
Priority / scheduling parameters
Process ID
Parent process ID
Signals
Process start time
Total CPU usage

File management

Root directory
Working (current) directory
File descriptors
User ID
Group ID

Memory management

Pointers to text, data, stack
or
Pointer to page table

What happens on a trap/interrupt?

1. Hardware saves program counter (on stack or in a special register)
2. Hardware loads new PC, identifies interrupt
3. Assembly language routine saves registers
4. Assembly language routine sets up stack
5. Assembly language calls C to run service routine
6. Service routine calls scheduler
7. Scheduler selects a process to run next (might be the one interrupted...)
8. Assembly language routine loads PC & registers for the selected process

Thread Concept: Motivation

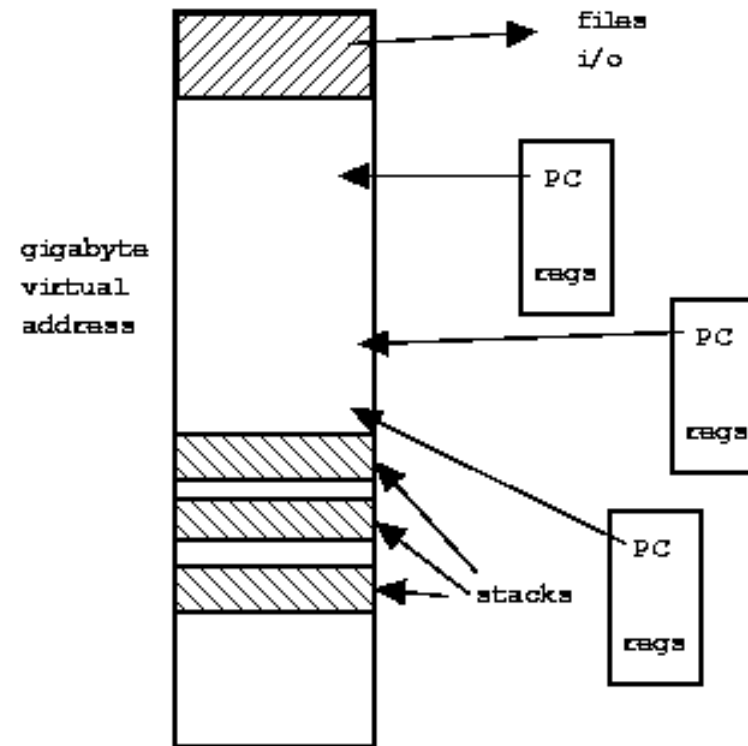
- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Processes and Threads

- Traditional process
 - One thread of control through a large, potentially sparse address space
 - Address space may be shared with other processes (shared mem)
 - Collection of systems resources (files, semaphores)
- Thread (light weight process)
 - A flow of control through an address space
 - Each address space can have multiple concurrent control flows
 - Each thread has access to entire address space
 - Potentially parallel execution, minimal state (low overheads)
 - May need synchronization to control access to shared variables

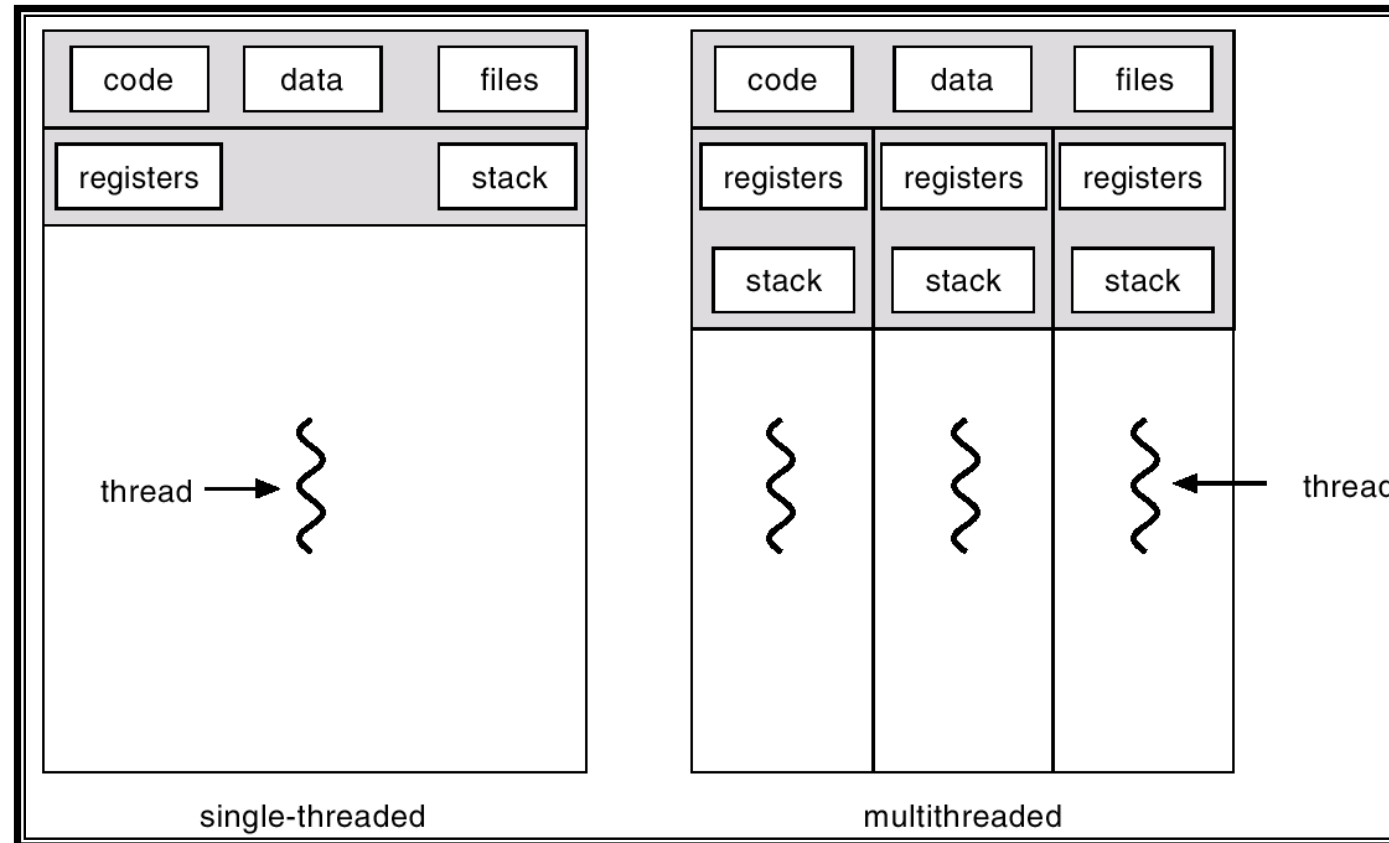
Threads

- Each thread has its own stack, PC, registers
 - Share address space, files,...



THREADS

Single and Multithreaded Processes



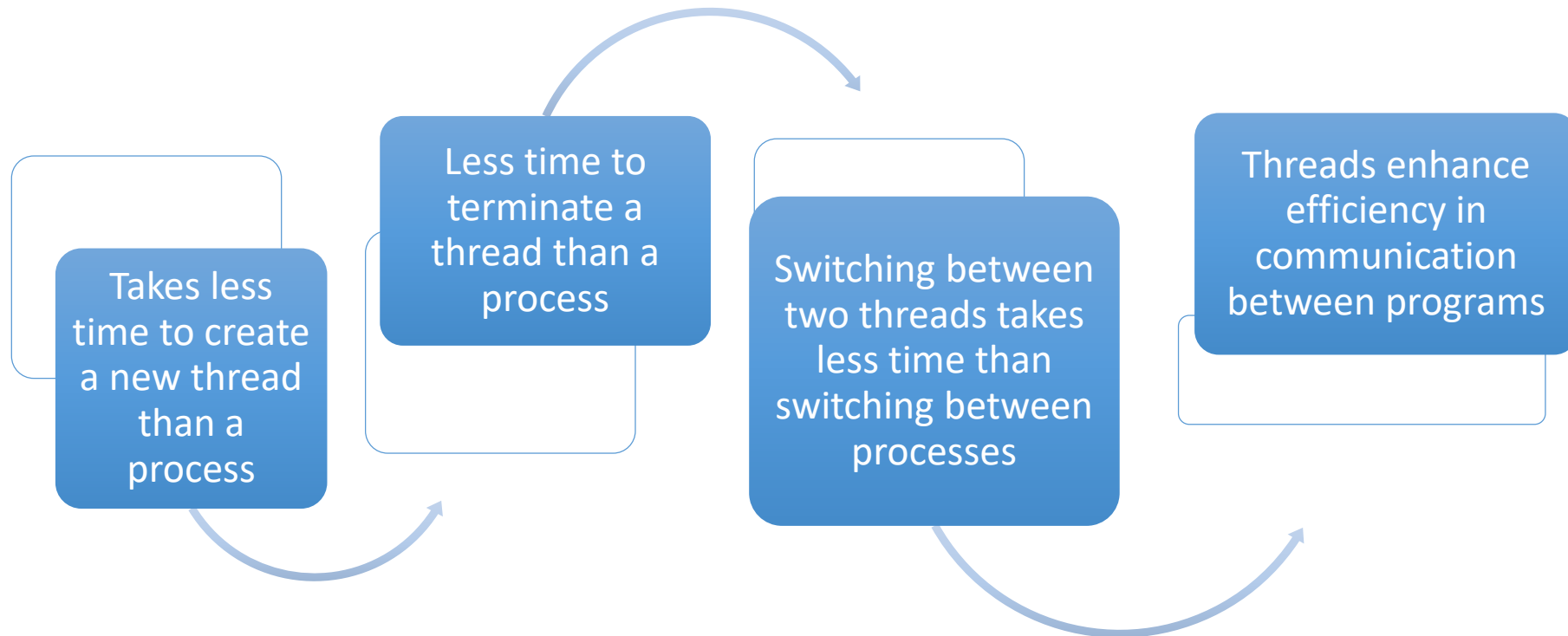
Why use Threads?

- Large multiprocessors need many computing entities (one per CPU)
- Switching between processes incurs high overhead
- With threads, an application can avoid per-process overheads
 - Thread creation, deletion, switching cheaper than processes
- Threads have full access to address space (easy sharing)
- Threads can execute in parallel on multiprocessors

Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

Benefits of Threads

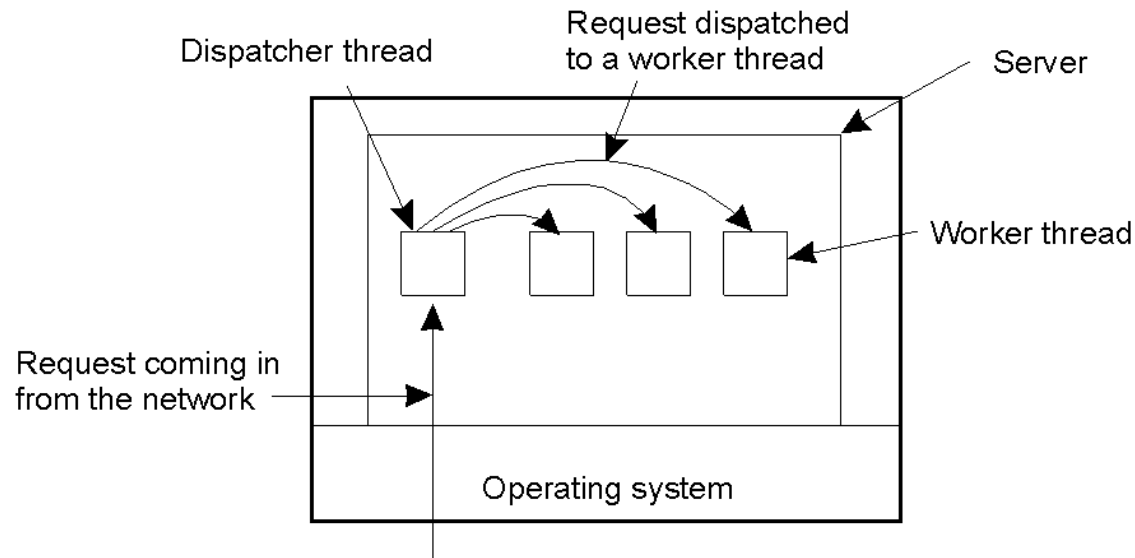


Multi-threaded Clients Example : Web Browsers

- Browsers such as IE are multi-threaded
- Such browsers can display data before entire document is downloaded: performs multiple simultaneous tasks
 - Fetch main HTML page, activate separate threads for other parts
 - Each thread sets up a separate connection with the server
 - Uses blocking calls
 - Each part (gif image) fetched separately and in parallel
 - Advantage: connections can be setup to different sources
 - Ad server, image server, web server...

Multi-threaded Server Example

- Apache web server: pool of pre-spawned worker threads
 - Dispatcher thread waits for requests
 - For each request, choose an idle worker thread
 - Worker thread uses blocking system calls to service web request



Thread Management

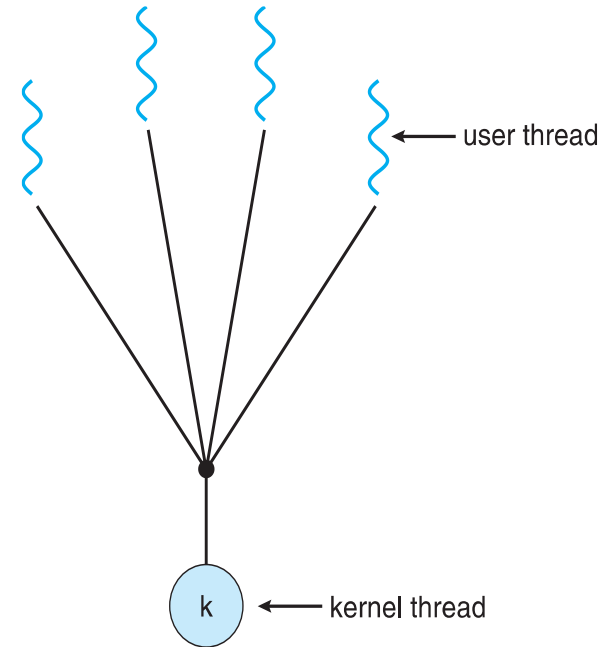
- Creation and deletion of threads
 - Static versus dynamic
- Critical sections
 - Synchronization primitives: blocking, spin-lock (busy-wait)
 - Condition variables
- Global thread variables
- Kernel versus user-level threads

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - **Solaris Green Threads**
 - **GNU Portable Threads**



One-to-One

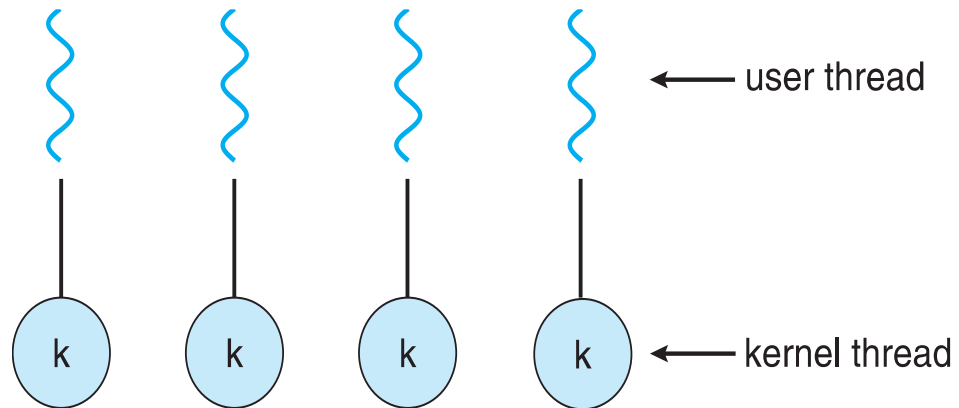
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

- Examples

- Windows 95/98/NT/2000

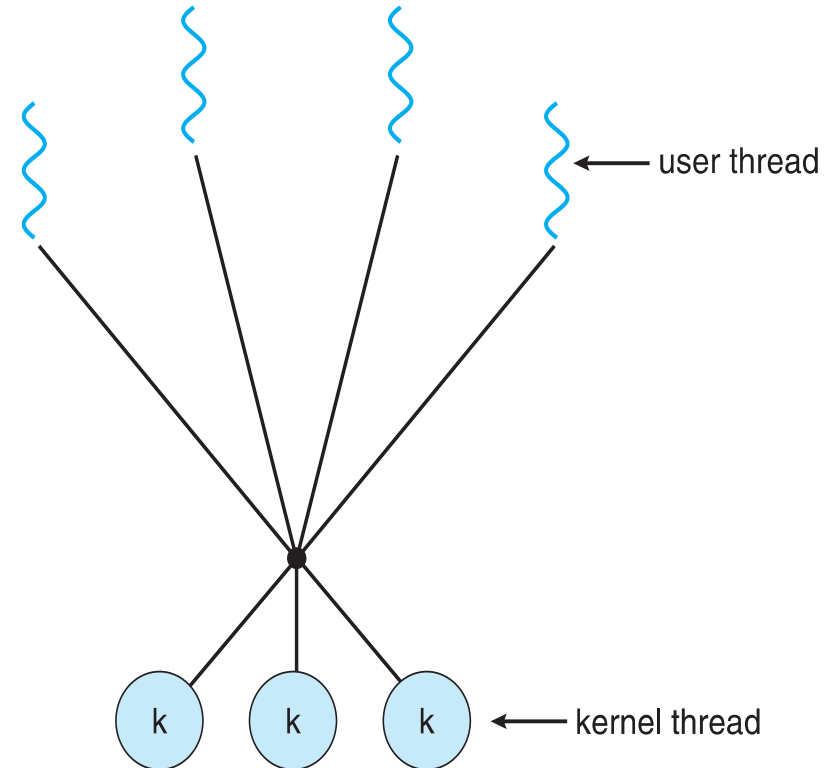
- Linux

Solaris 9 and later



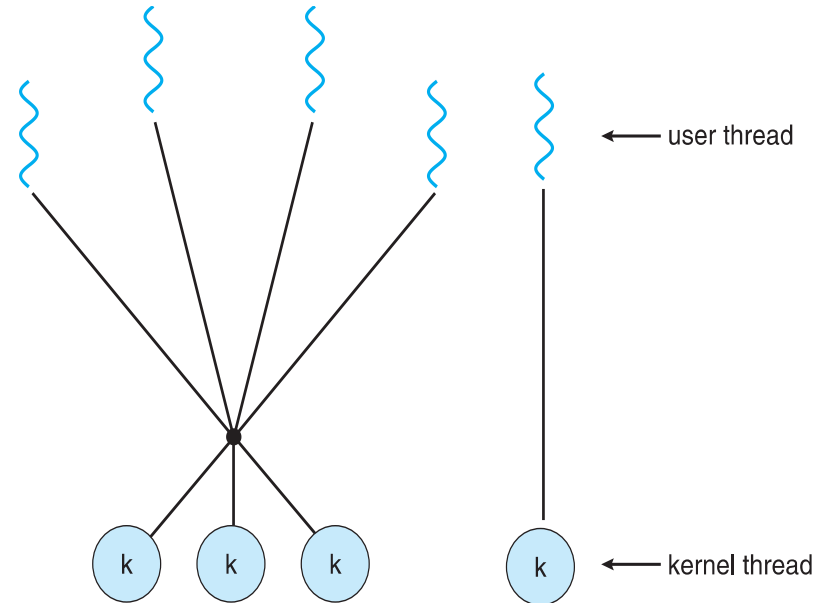
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier



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

Threads vs. Processes

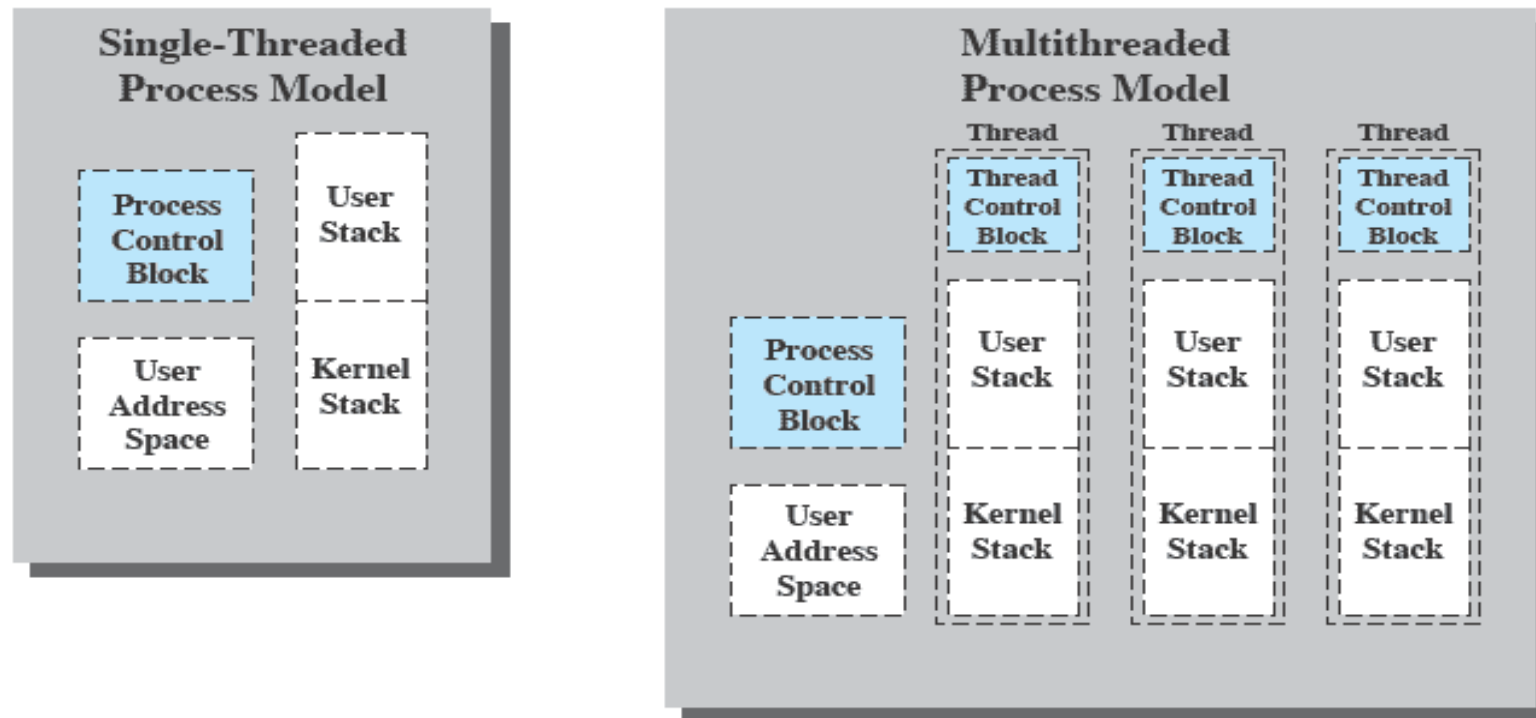


Figure 4.2 Single Threaded and Multithreaded Process Models

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

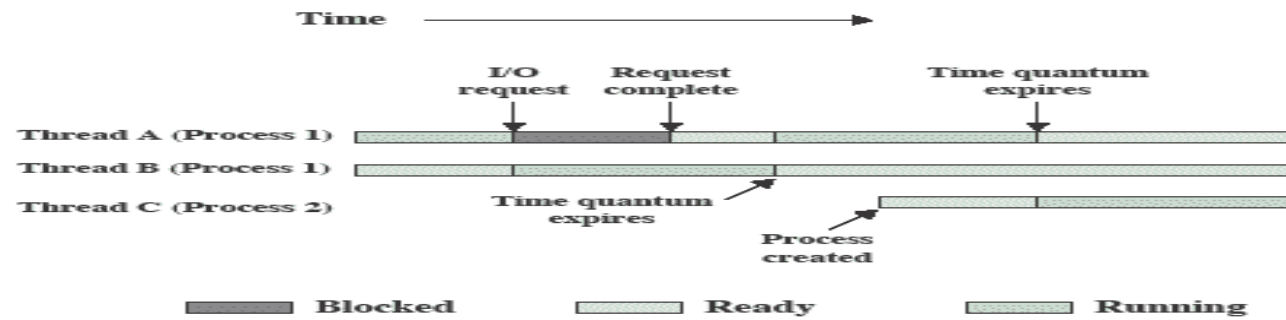


Figure 4.4 Multithreading Example on a Uniprocessor

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

User-level versus kernel threads

- *Key issues:*
- Cost of thread management
 - More efficient in user space
- Ease of scheduling
- Flexibility: many parallel programming models and schedulers
- Process blocking – a potential problem

User-level Threads

- Threads managed by a threads library
 - Kernel is unaware of presence of threads
- Advantages:
 - No kernel modifications needed to support threads
 - Efficient: creation/deletion/switches don't need system calls
 - Flexibility in scheduling: library can use different scheduling algorithms, can be application dependent
- Disadvantages
 - Need to avoid blocking system calls [all threads block]
 - Threads compete for one another
 - Does not take advantage of multiprocessors [no real parallelism]

User-level threads

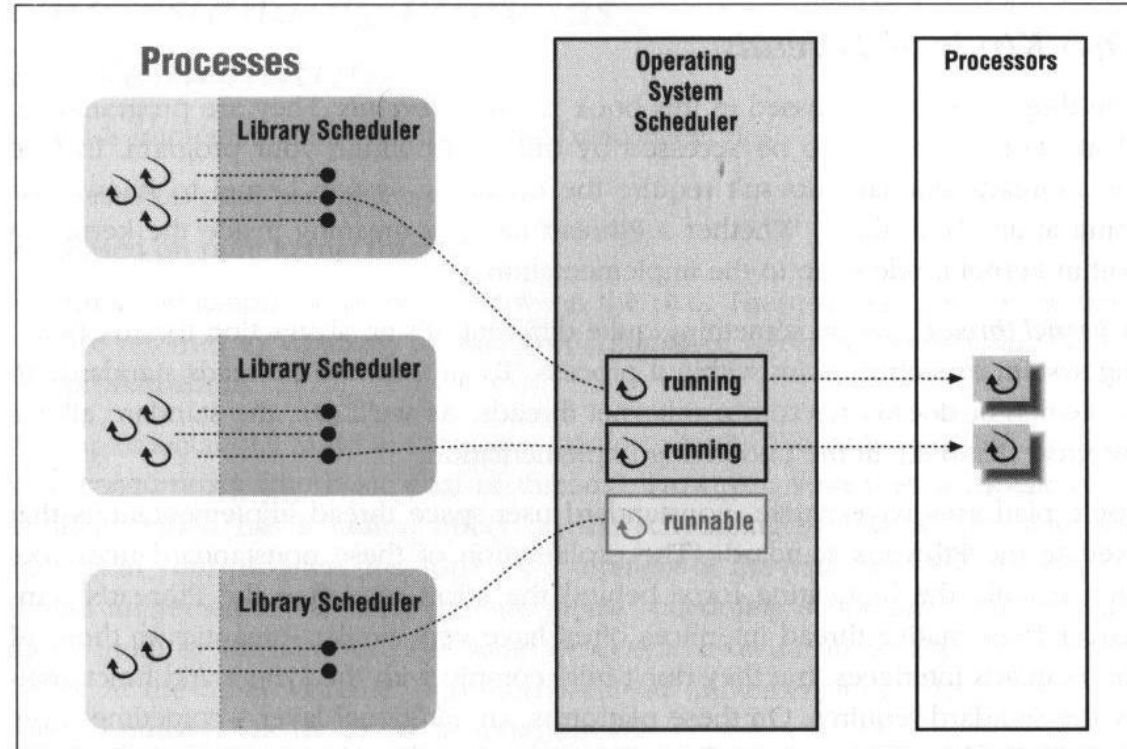


Figure 6-1: User-space thread implementations

Kernel-level threads

- Kernel aware of the presence of threads
 - Better scheduling decisions, more expensive
 - Better for multiprocessors, more overheads for uniprocessors

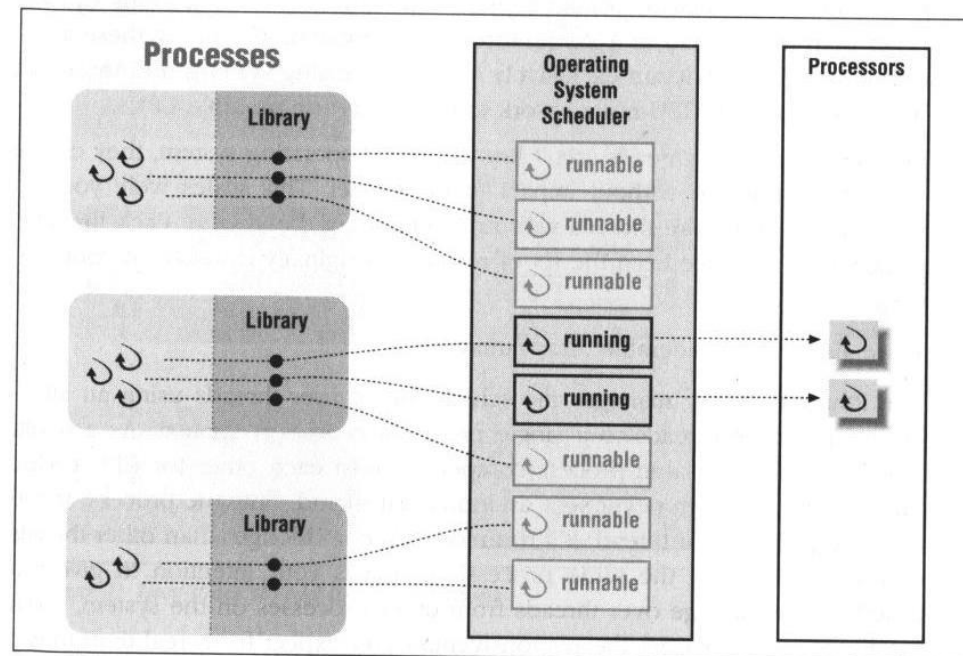


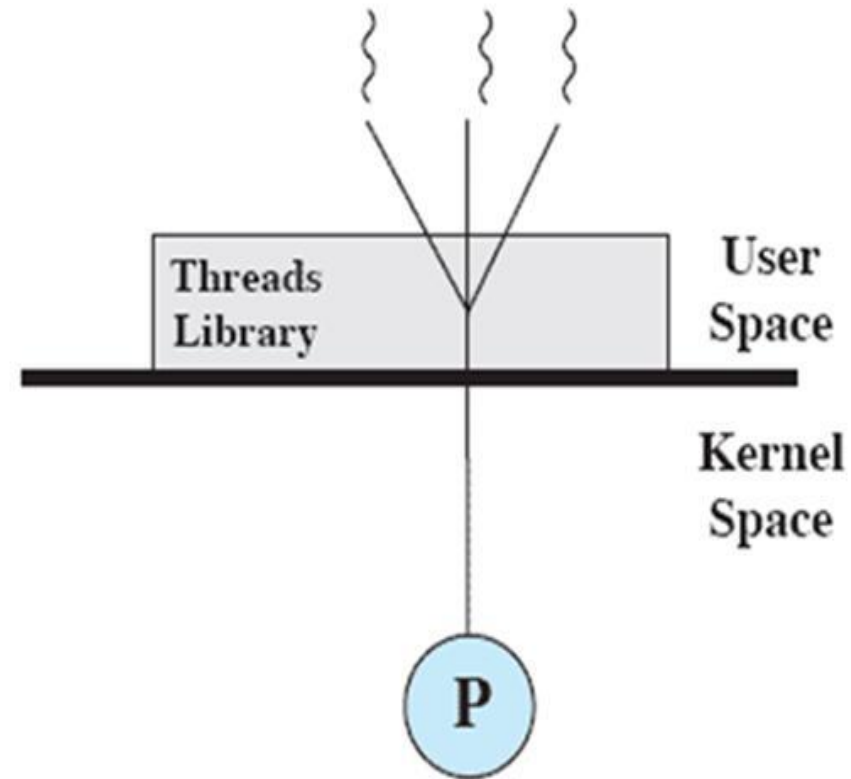
Figure 6-2: Kernel thread-based implementations

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
 - Windows
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X

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

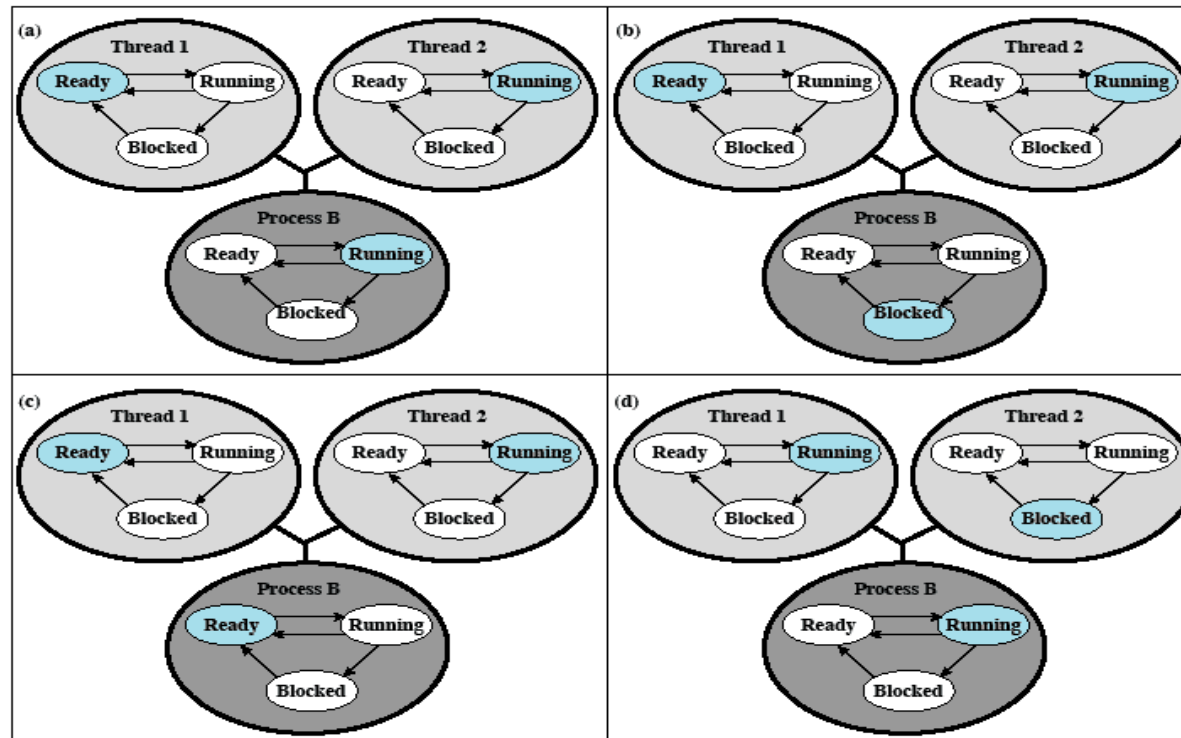
Relationships Between ULT States and Process States

Possible
transitions
from 4.6a:

4.6a→4.6b

4.6a→4.6c

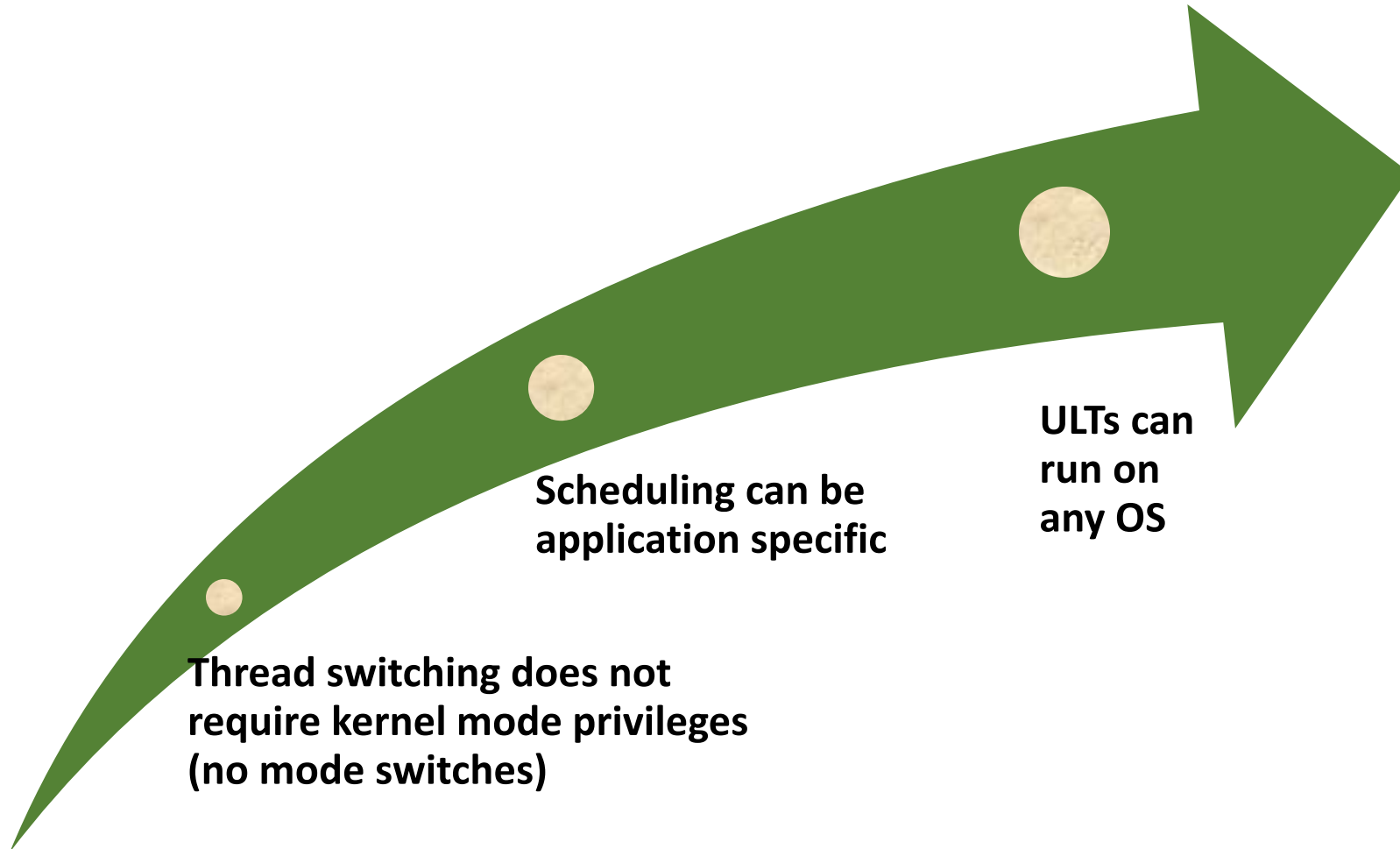
4.6a→4.6d



Colored state
is current state

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

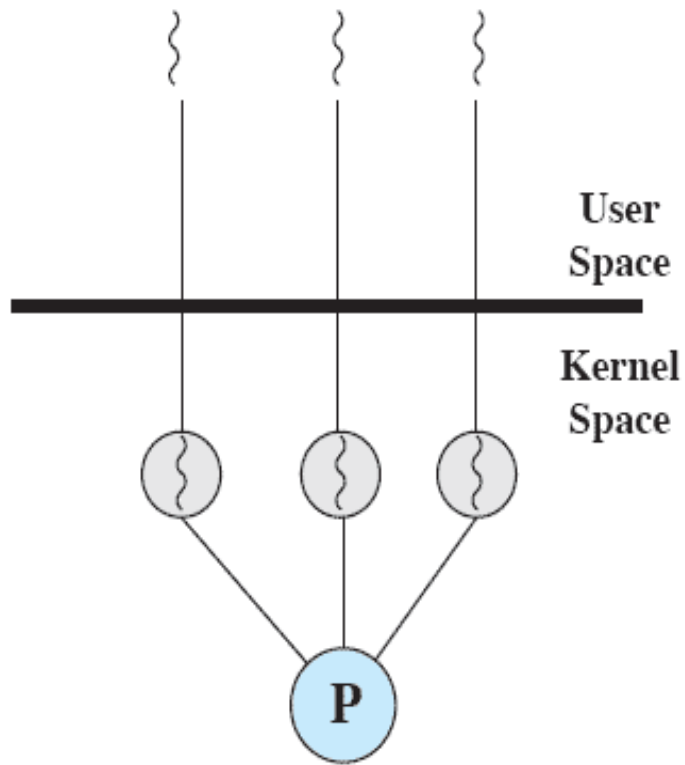
Advantages of ULTs



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

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

- ◆ Thread management is done by the kernel (could call them *KMT*)
 - ◆ 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

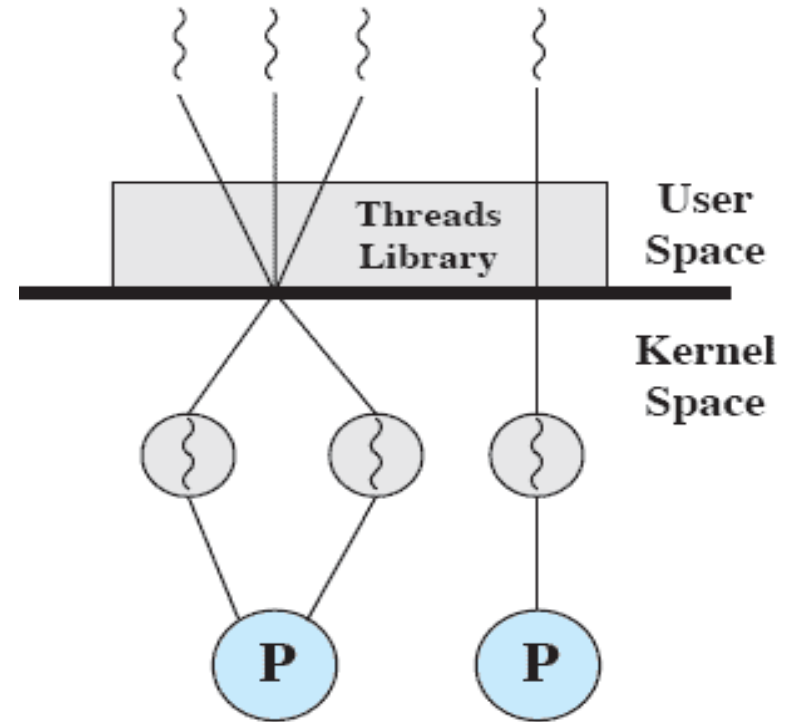
 The transfer of control from one thread to another within the same process requires a mode switch to the

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 (μ s)

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

Light-weight Processes

- Several LWPs per heavy-weight process
- User-level threads package
 - Create/destroy threads and synchronization primitives
- Multithreaded applications – create multiple threads, assign threads to LWPs (one-one, many-one, many-many)
- Each LWP, when scheduled, searches for a runnable thread *[two-level scheduling]*
 - Shared thread table: no kernel support needed
- When a LWP thread block on system call, switch to kernel mode and OS context switches to another LWP

LWP Example

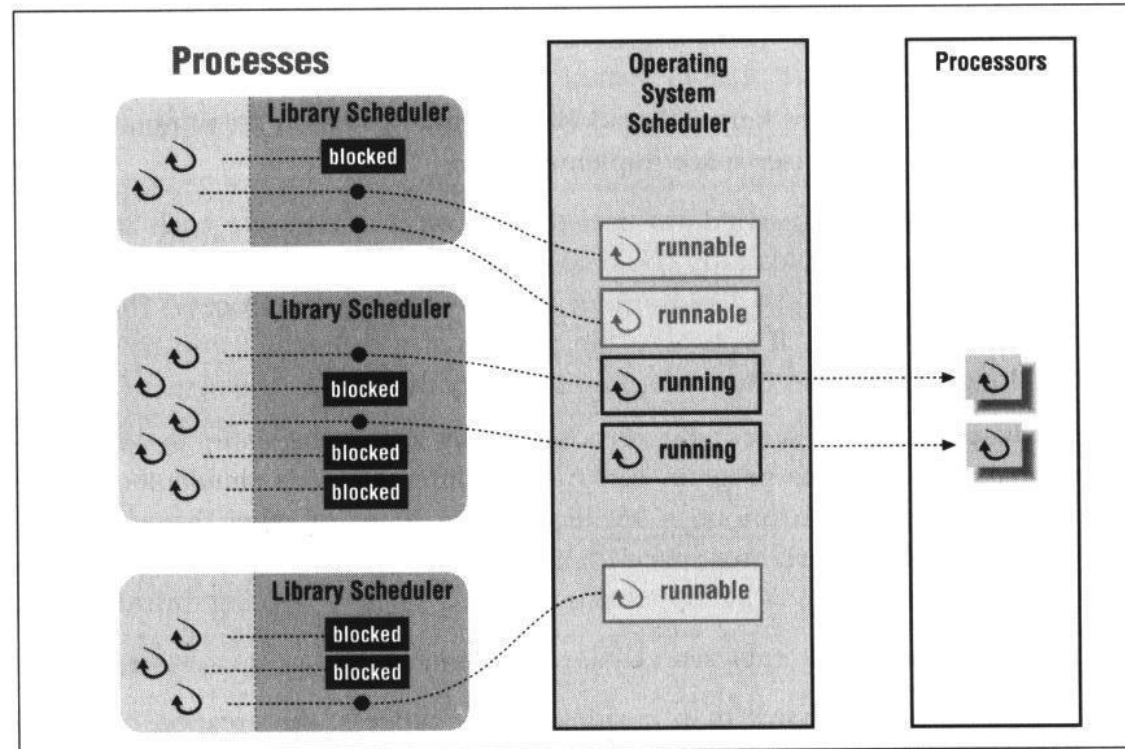


Figure 6-3: Two-level scheduler implementations

THREADS

User Threads

- Thread management done by user-level threads library
- Examples
 - POSIX *Pthreads*
 - Mach *C-threads*
 - Solaris *threads*

- Supported by the Kernel
- Examples
 - Windows 95/98/NT/2000
 - Solaris
 - Tru64 UNIX
 - BeOS
 - Linux

Kernel Threads

THREADS

Semantics of `fork()` and `exec()` system calls

- Does **`fork()`** duplicate only the calling thread or all threads?

Thread cancellation

- Terminating a thread before it has finished
- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

THREADS

Threading Issues

Signal handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A **signal handler** is used to process signals
 - 1.Signal is generated by particular event
 - 2.Signal is delivered to a process
 - 3.Signal is handled
- Options:
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process

Threading Issues

Thread pools

- Create a number of threads in a pool where they await work
- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool

THREADS

Threading Issues

Thread specific data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler activations

- Many:Many models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

THREADS

Various Implementations

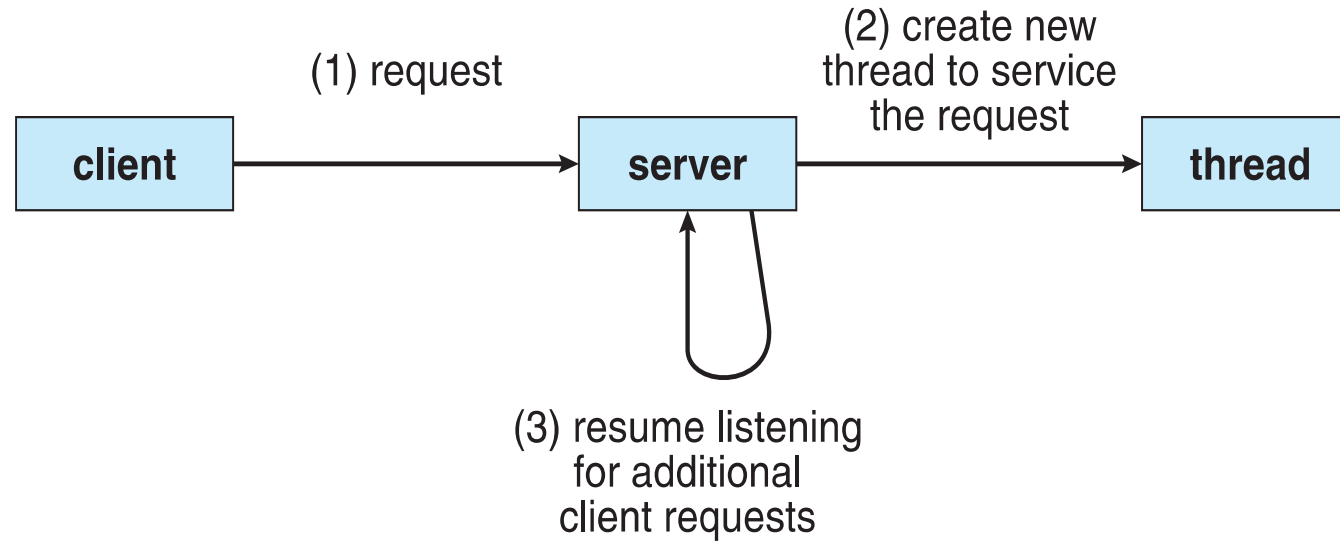
PThreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Windows Threads

- Implements the one-to-one mapping
- Each thread contains
 - A thread id
 - Register set
 - Separate user and kernel stacks
 - Private data storage area
- The register set, stacks, and private storage area are known as the **context** of the threads

Multithreaded Server Architecture



Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

- Types of parallelism
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
 - CPUs have cores as well as ***hardware threads***
 - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches $1 / S$

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

Operating System Examples

- Windows Threads
- Linux Threads
- Solaris Thread
- Mac Threads
- Java Threads

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

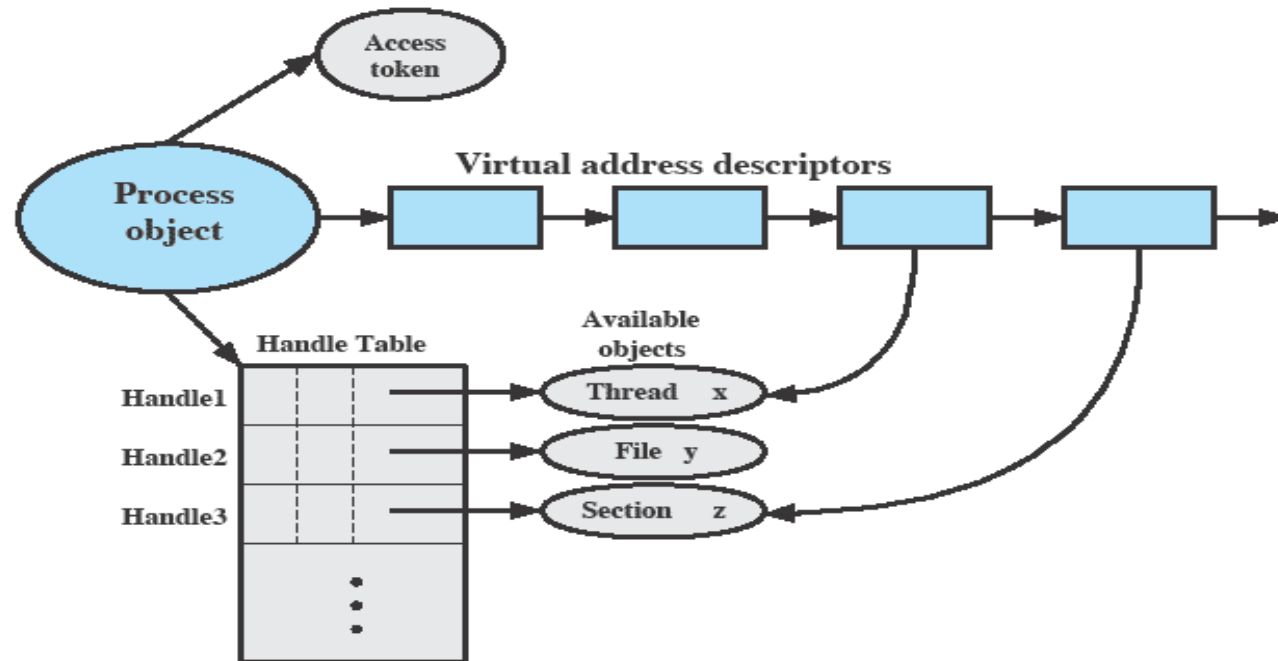


Figure 4.10 A Windows Process and Its Resources

Figure 4.12 A Windows Process and Its Resources

Process and Thread Objects

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

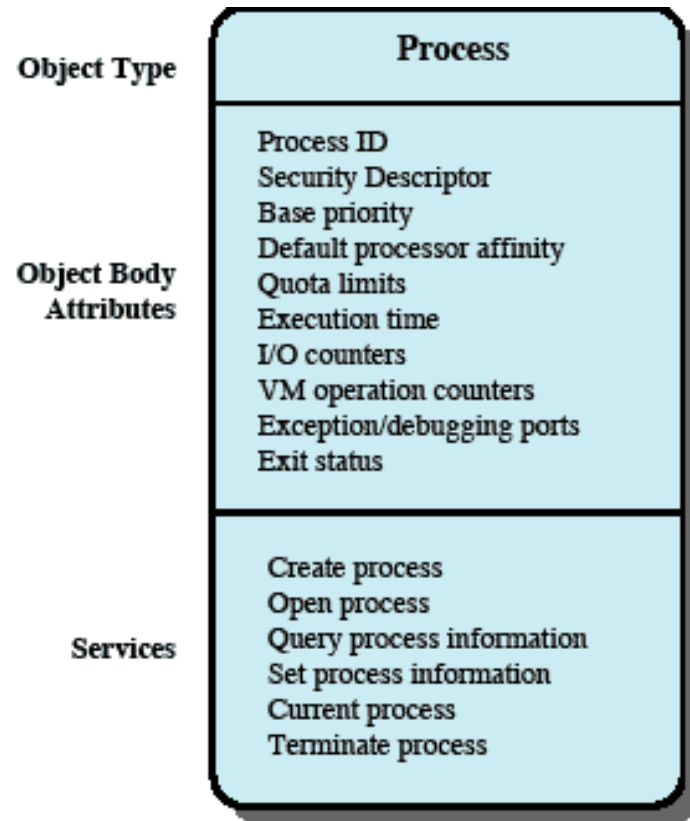
Processes

- an entity corresponding to a user job or application that owns resources

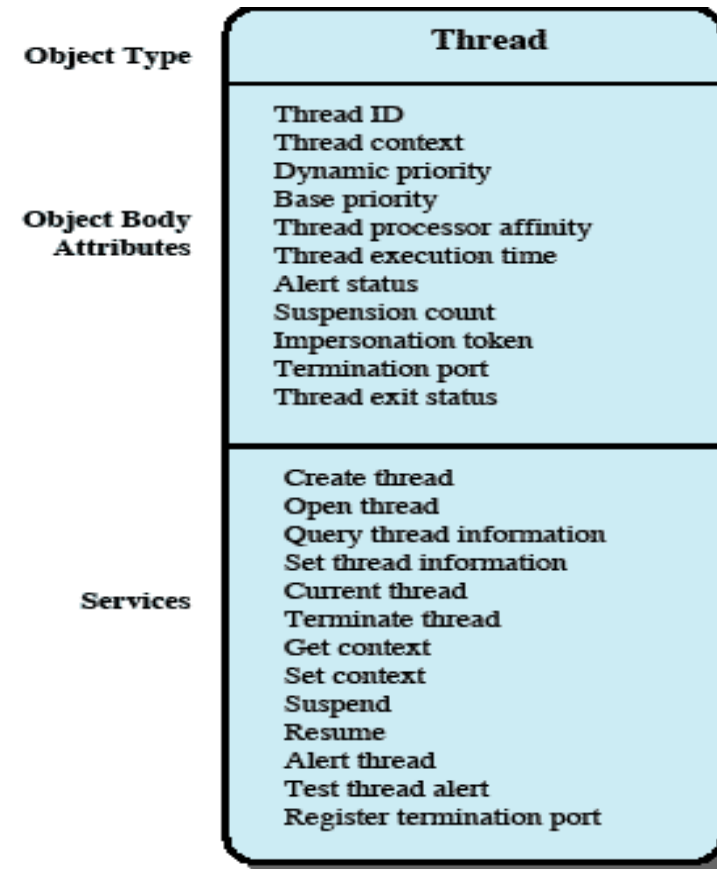
Threads

- 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.

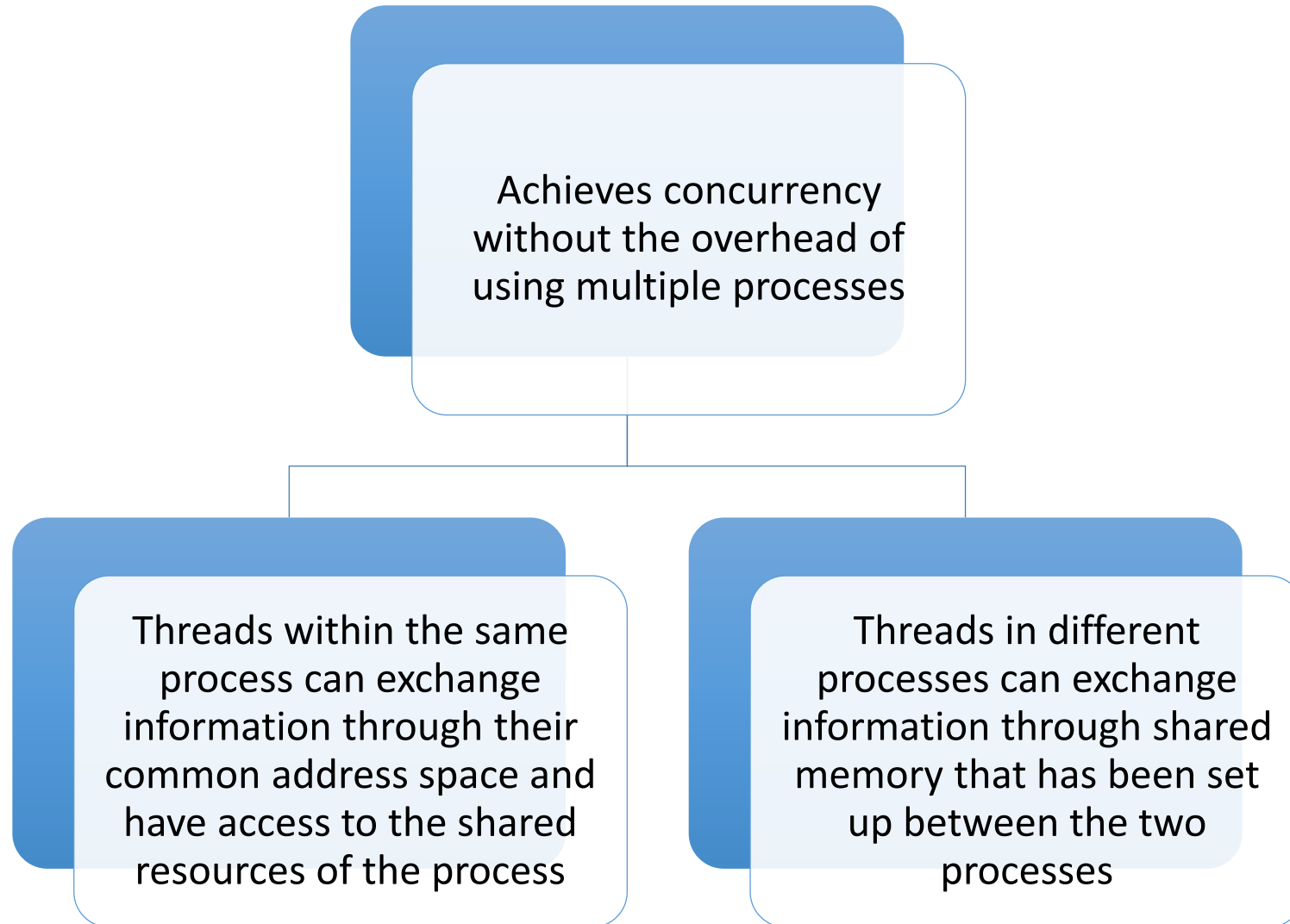
Table 4.3 Windows Process Object Attributes

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 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



Thread States

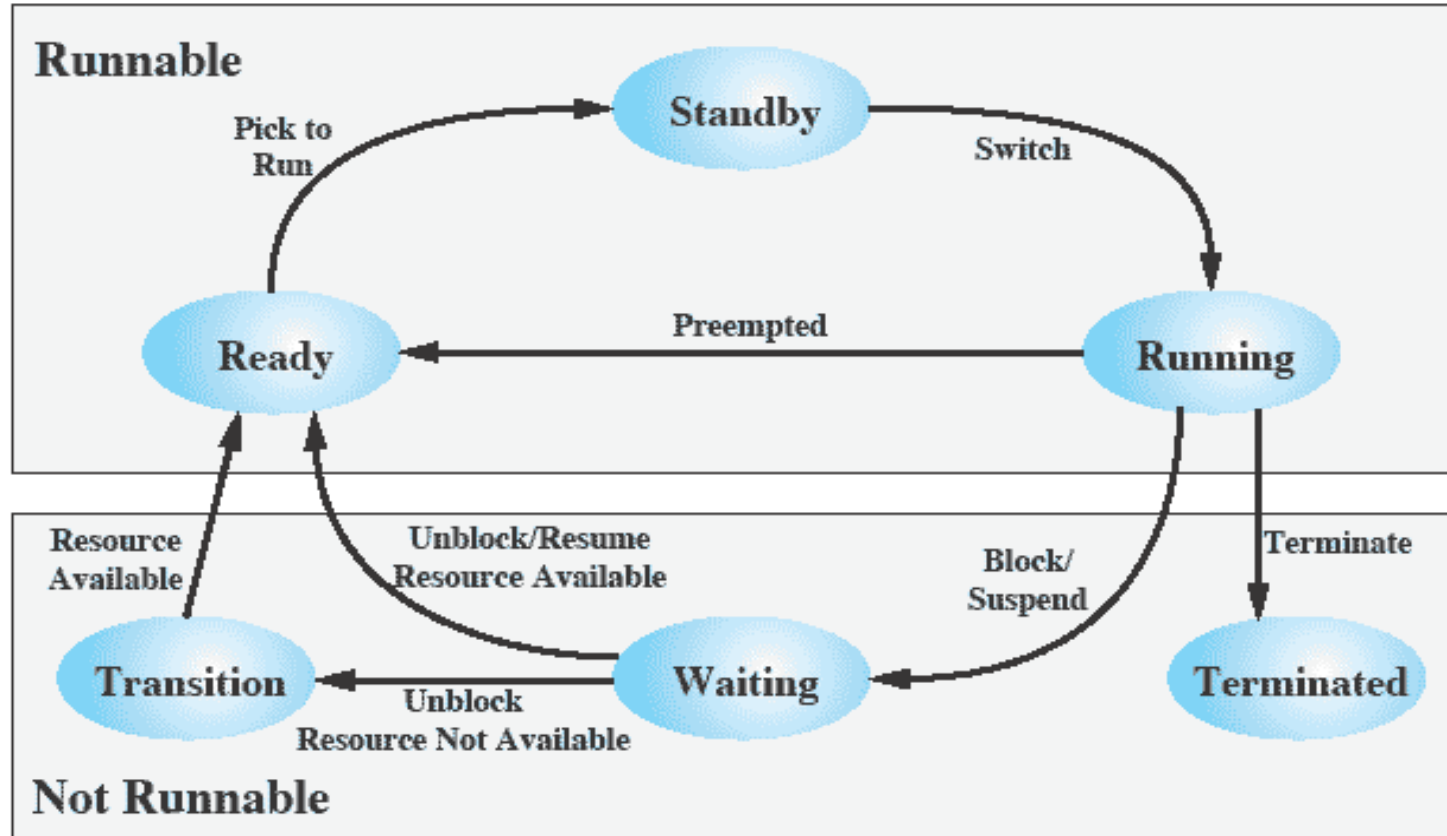
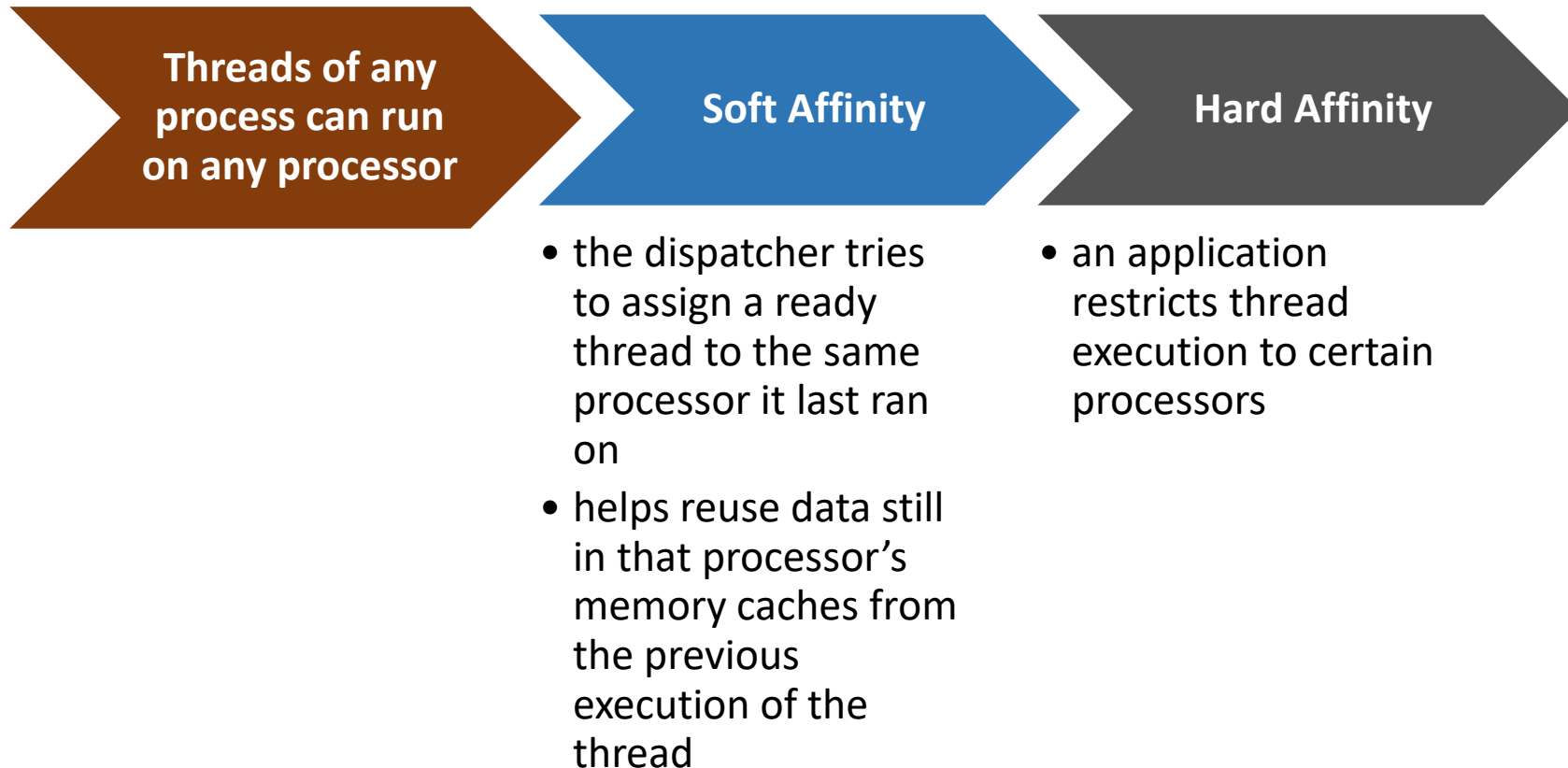


Figure 4.14 Windows Thread States

Symmetric Multiprocessing Support (SMP)



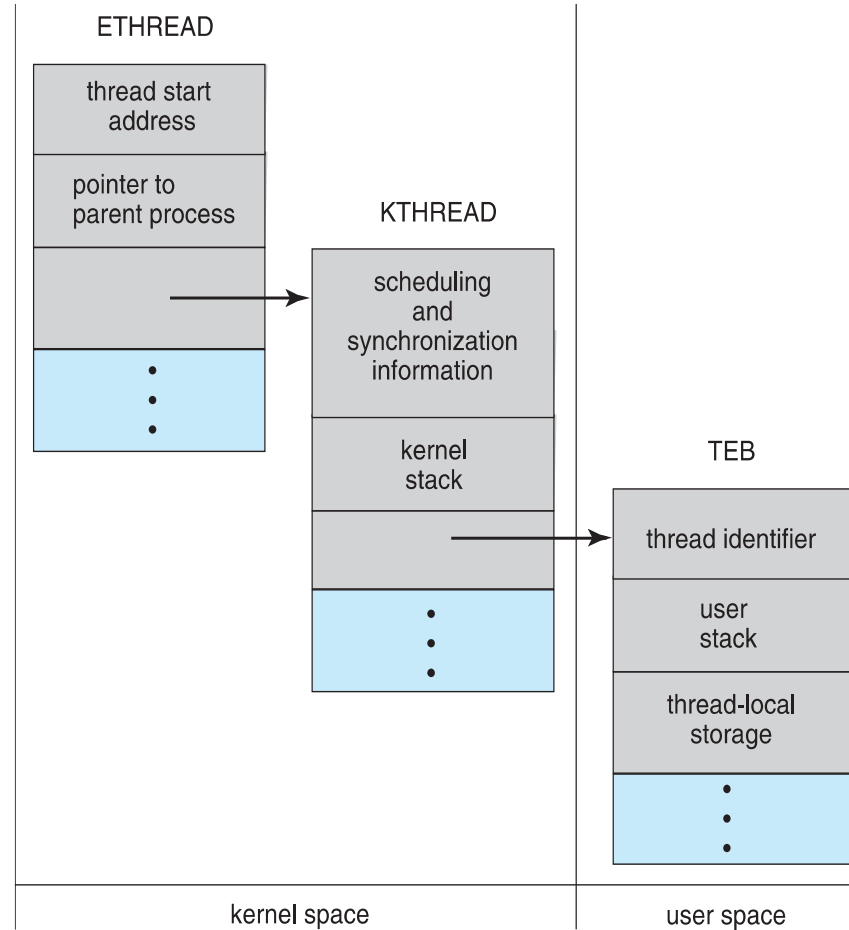
Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
 - A thread id
 - Register set representing state of processor
 - Separate user and kernel stacks for when thread runs in user mode or kernel mode
 - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

Windows Threads (Cont.)

- The primary data structures of a thread include:
 - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
 - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
 - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows Threads Data Structures



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

Processes and Threads in Solaris

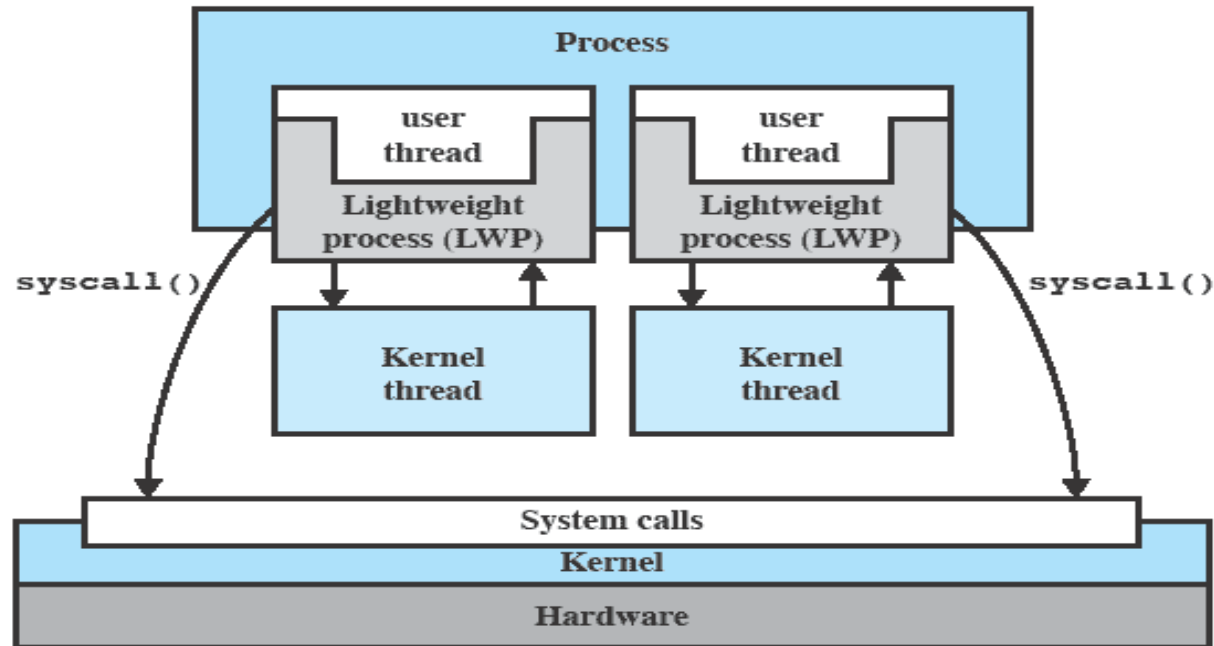
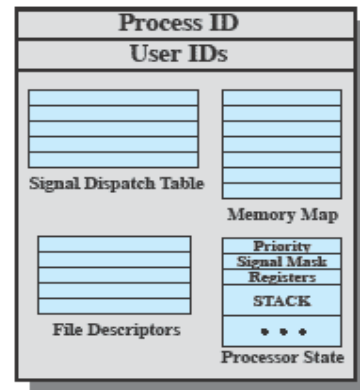


Figure 4.13 Processes and Threads in Solaris [MCDO07]

Traditional Unix vs Solaris

UNIX Process Structure



Solaris Process Structure

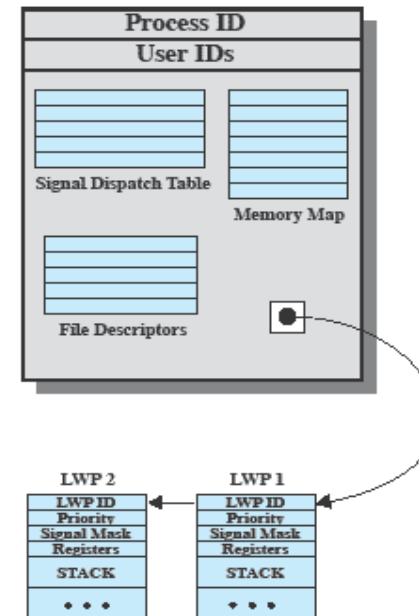


Figure 4.14 Process Structure in Traditional UNIX and Solaris [LEWI96]

Figure 4.16 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

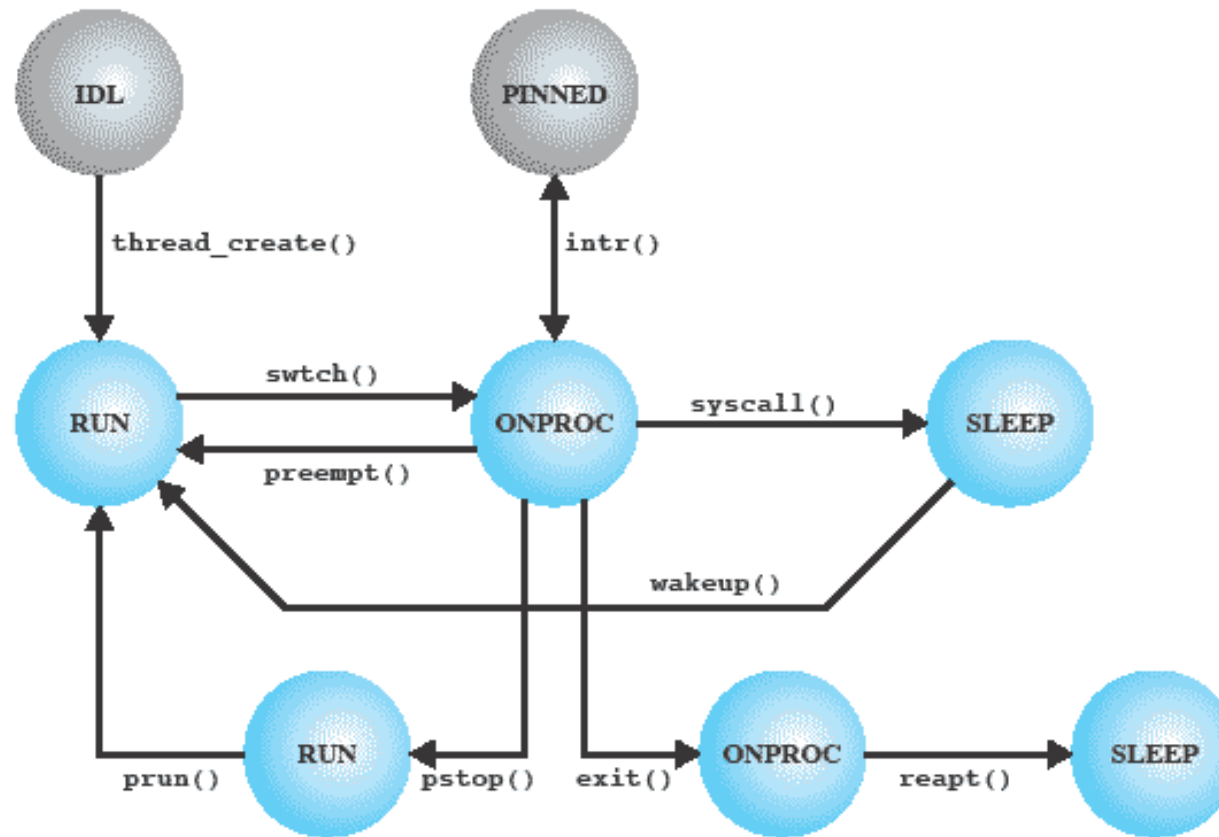


Figure 4.15 Solaris Thread States

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

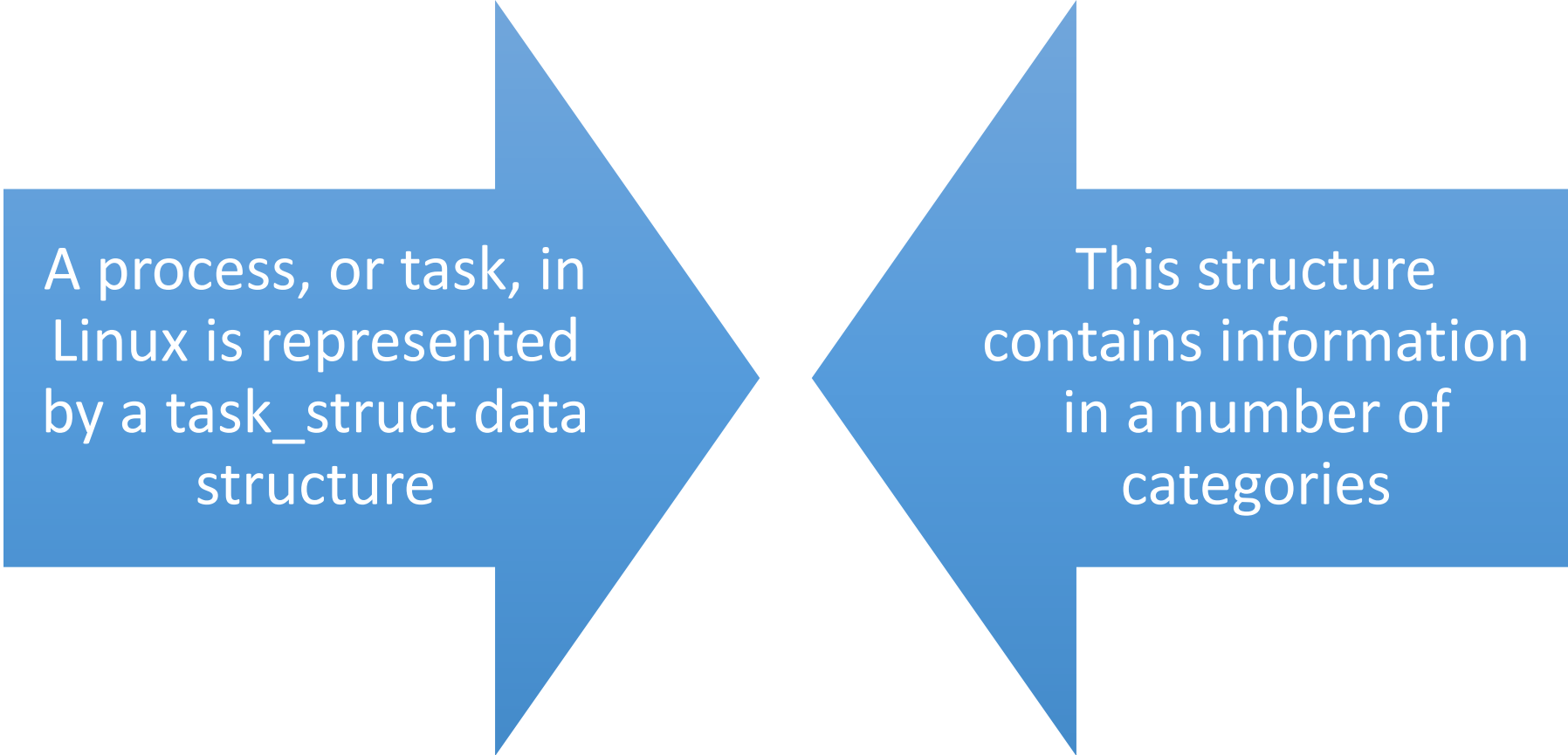
Interrupts

- 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

The diagram consists of two large blue arrows pointing towards each other, meeting at a central point. The left arrow points right and contains the text 'A process, or task, in Linux is represented by a task_struct data structure'. The right arrow points left and contains the text 'This structure contains information in a number of categories'.

This structure contains information in a number of categories

Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
 - Flags control behavior

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

- **`struct task_struct`** points to process data structures (shared or unique)

Linux Process/Thread Model

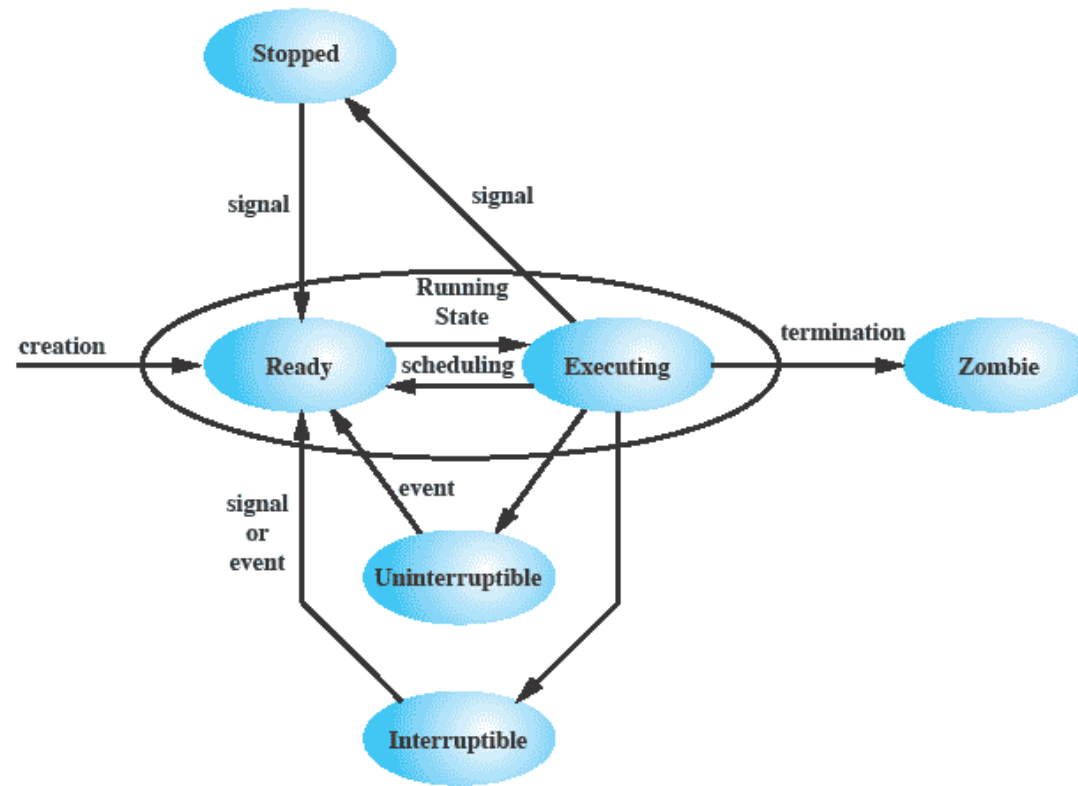
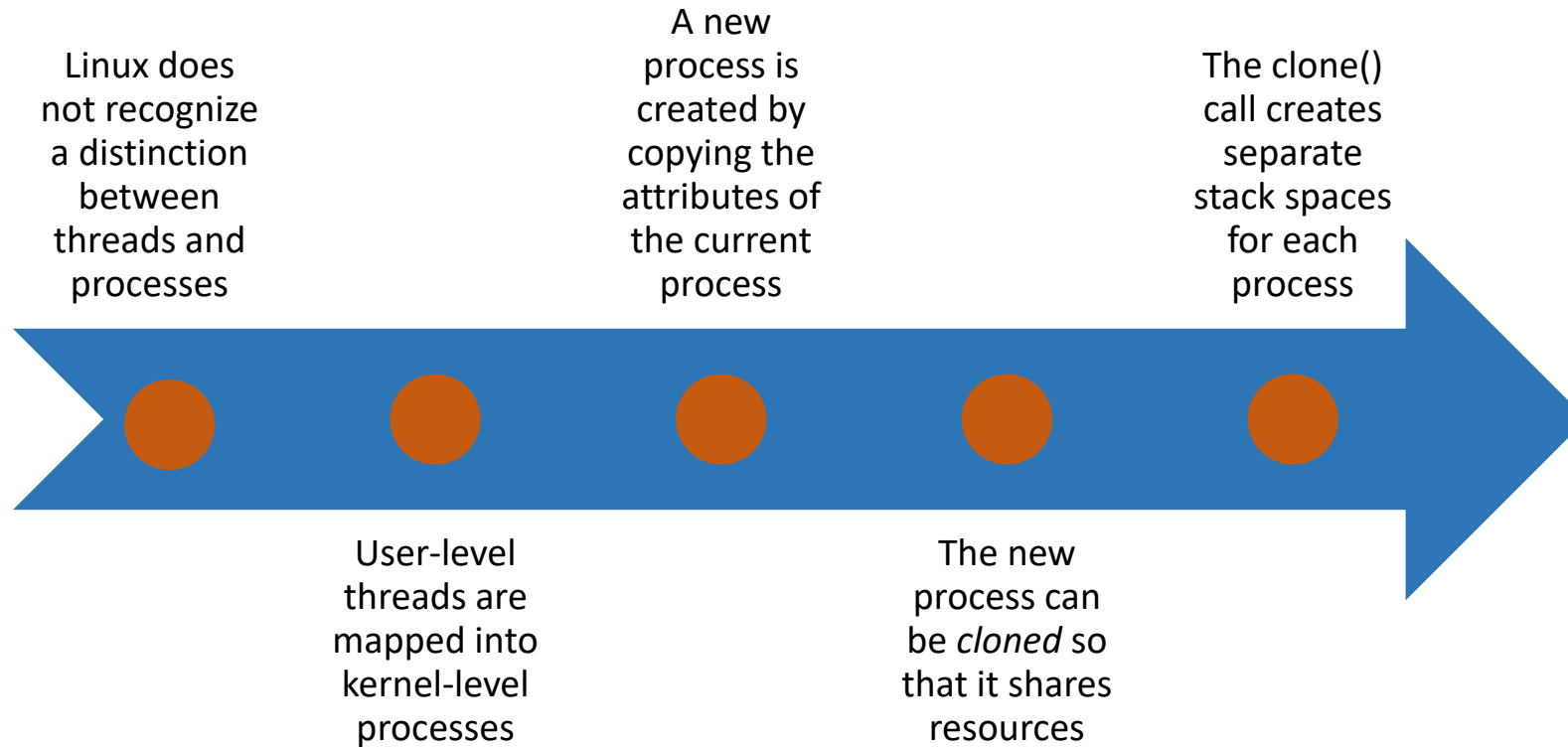


Figure 4.16 Linux Process/Thread Model

Linux Threads



Linux Clone () Flags

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

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

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr, "usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }
}
```

Pthreads Example (Cont.)

```
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }
}
```

Windows Multithreaded C Program (Cont.)

```
/* create the thread */
ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}
}
```

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

```
public interface Runnable
{
    public abstract void run();
}
```

- Extending Thread class
- Implementing the Runnable interface

Java Multithreaded Program

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```


Java Multithreaded Program (Cont.)

```
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>"); }
}
```

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
 - Thread Pools
 - OpenMP
 - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), **java.util.concurrent** package

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool
 - Separating task to be performed from mechanics of creating task allows different strategies for running task
 - i.e.Tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for  
for(i=0;i<N;i++) {  
    c[i] = a[i] + b[i];  
}
```

Run for loop in parallel

```
#include <omp.h>  
#include <stdio.h>  
  
int main(int argc, char *argv[])  
{  
    /* sequential code */  
  
    #pragma omp parallel  
    {  
        printf("I am a parallel region.");  
    }  
  
    /* sequential code */  
  
    return 0;  
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “`^ { }`” - `^ { printf("I am a block"); }`
- Blocks placed in dispatch queue
 - Assigned to available thread in thread pool when removed from queue

Grand Central Dispatch

- Two types of dispatch queues:
 - serial – blocks removed in FIFO order, queue is per process, called **main queue**
 - Programmers can create additional serial queues within program
 - concurrent – removed in FIFO order but several may be removed at a time
 - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue  
    (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);  
  
dispatch_async(queue, ^{ printf("I am a block."); });
```

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
 - Synchronous and asynchronous
- Thread cancellation of target thread
 - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of `fork()` and `exec()`

- Does **`fork()`** duplicate only the calling thread or all threads?
 - Some UNIXes have two versions of `fork`
- **`exec()`** usually works as normal – replace the running process including all threads

Signal Handling

- n **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- n A **signal handler** is used to process signals
 1. Signal is generated by particular event
 2. Signal is delivered to a process
 3. Signal is handled by one of two signal handlers:
 1. default
 2. user-defined
- n Every signal has **default handler** that kernel runs when handling signal
 - | **User-defined signal handler** can override default
 - | For single-threaded, signal delivered to process

Signal Handling (Cont.)

- n Where should a signal be delivered for multi-threaded?
 - | Deliver the signal to the thread to which the signal applies
 - | Deliver the signal to every thread in the process
 - | Deliver the signal to certain threads in the process
 - | Assign a specific thread to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
 - Cancellation only occurs when thread reaches **cancellation point**
 - I.e. `pthread_testcancel()`
 - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across function invocations
- Similar to **static** data
 - TLS is unique to each thread

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process (LWP)**
 - Appears to be a virtual processor on which process can schedule user thread to run
 - Each LWP attached to kernel thread
 - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads

