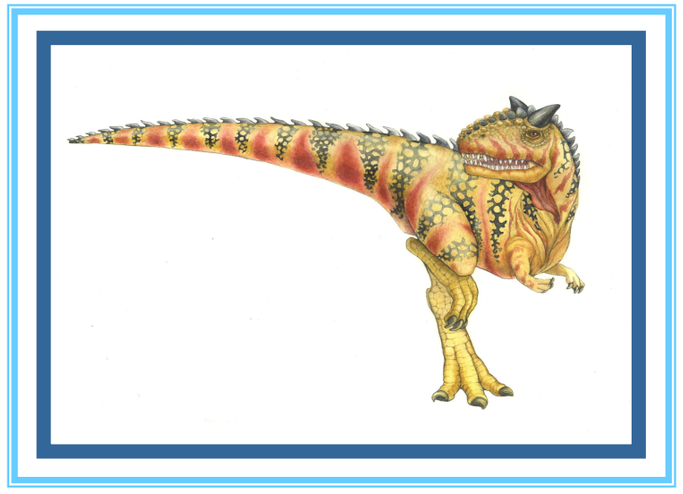
Chapter 4: Multithreaded Programming



Chapter 4: Multithreaded Programming

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

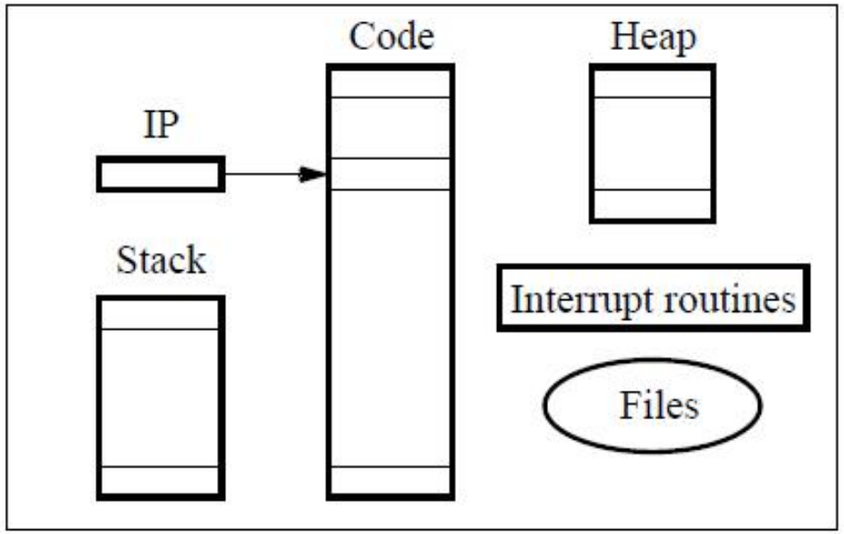
- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

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

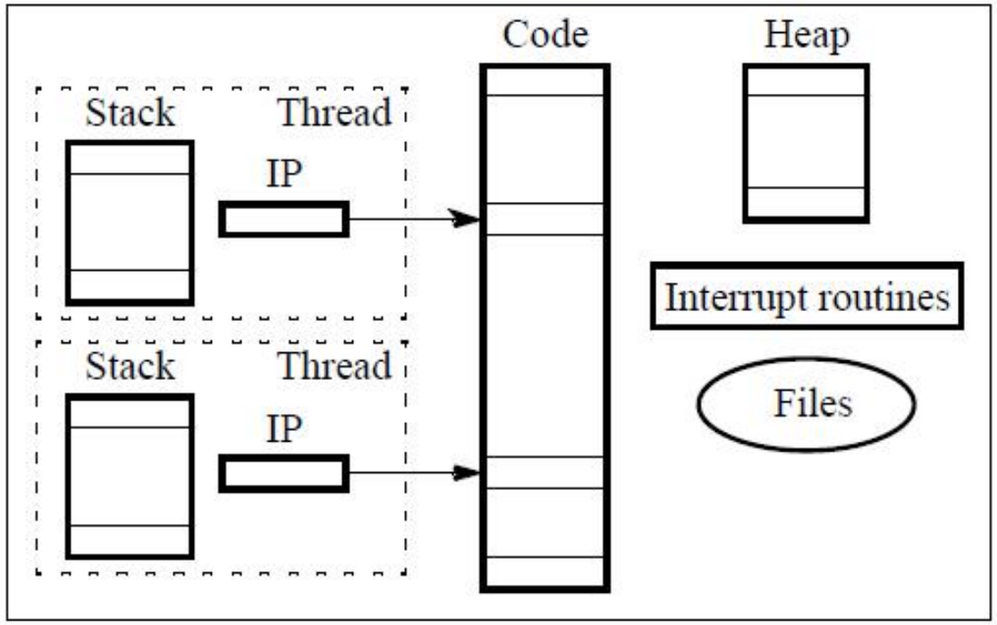
Process vs. Thread

"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.



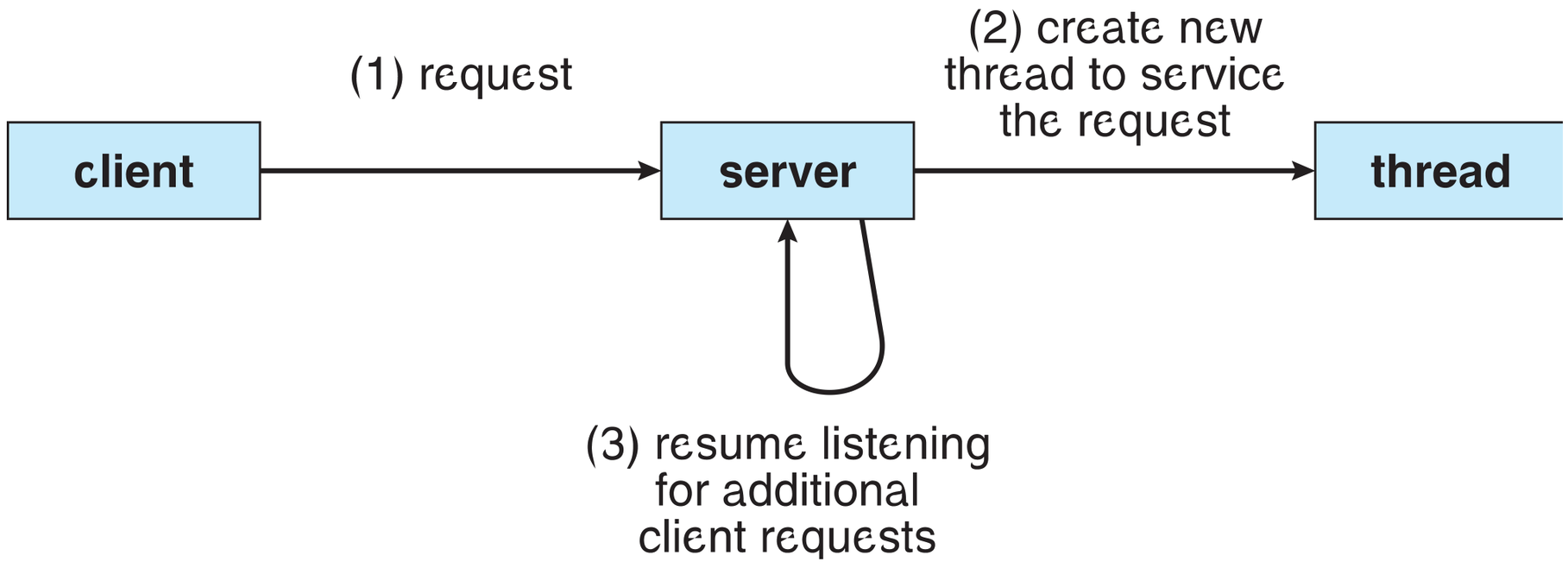
(a) Process

Threads - *shares* the same memory space and global variables between routines.



(b) Threads

Multithreaded Server Architecture



Benefits

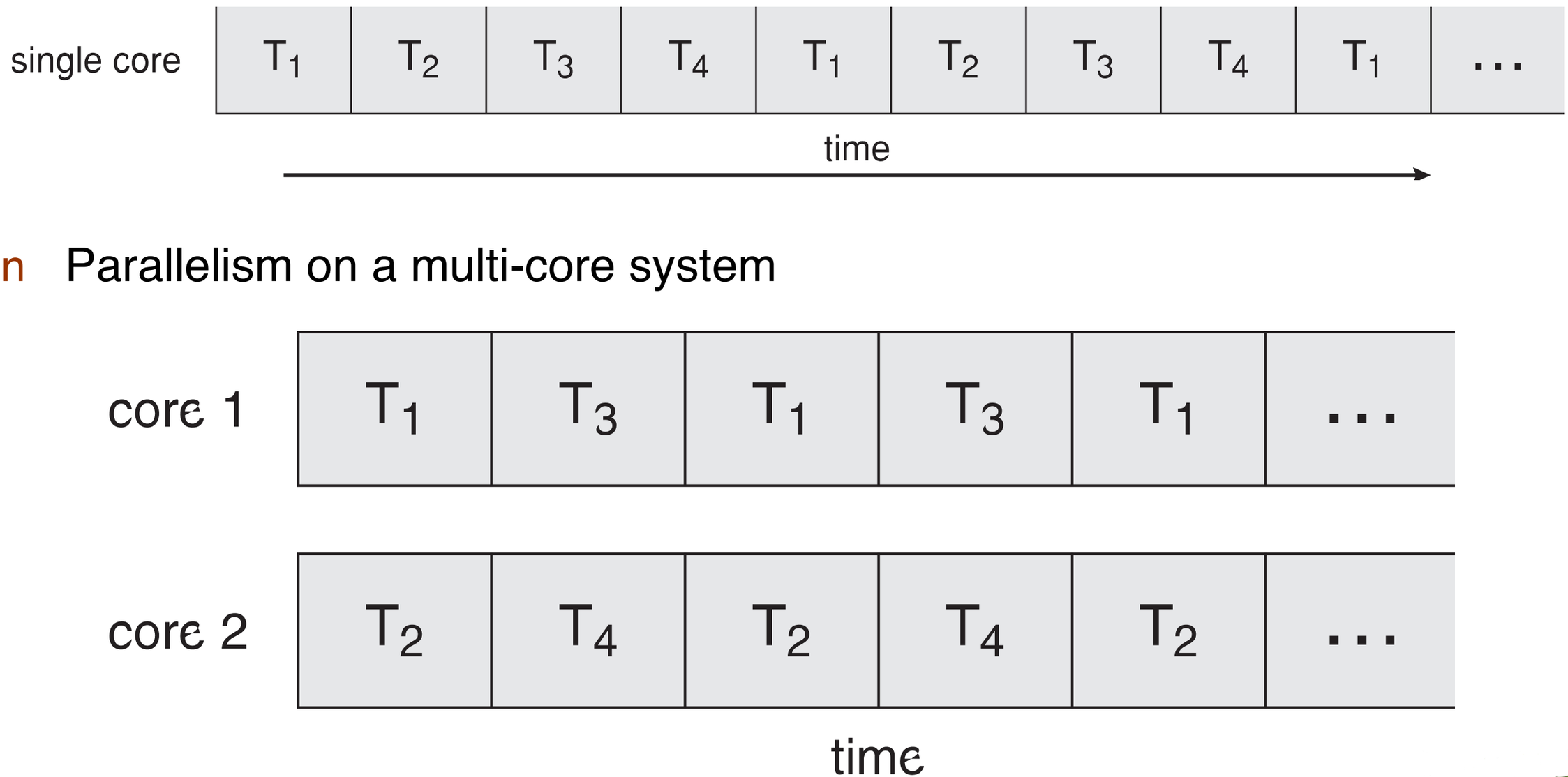
- Responsiveness – May allow continued execution if part of process is blocked, especially important for user interfaces
- Resource Sharing – Threads share resources of process, easier than shared memory or message passing
- Economy – Cheaper than process creation, thread switching lower overhead than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

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
- Types of parallelism
  - Data parallelism – Distributs subsets of the same data across multiple cores, same operation on each
  - Task parallelism – Distribute 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

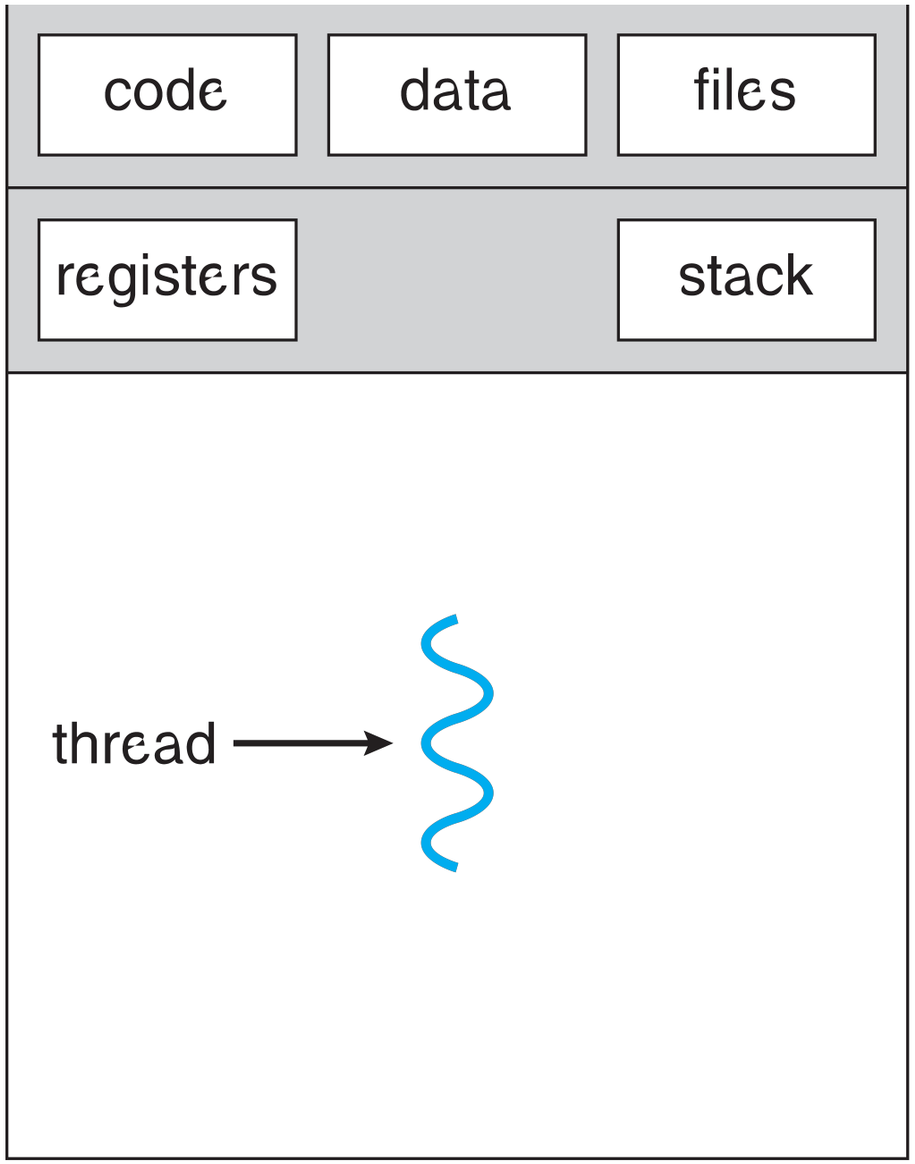
Concurrency vs. Parallelism

- Concurrent execution on single-core system

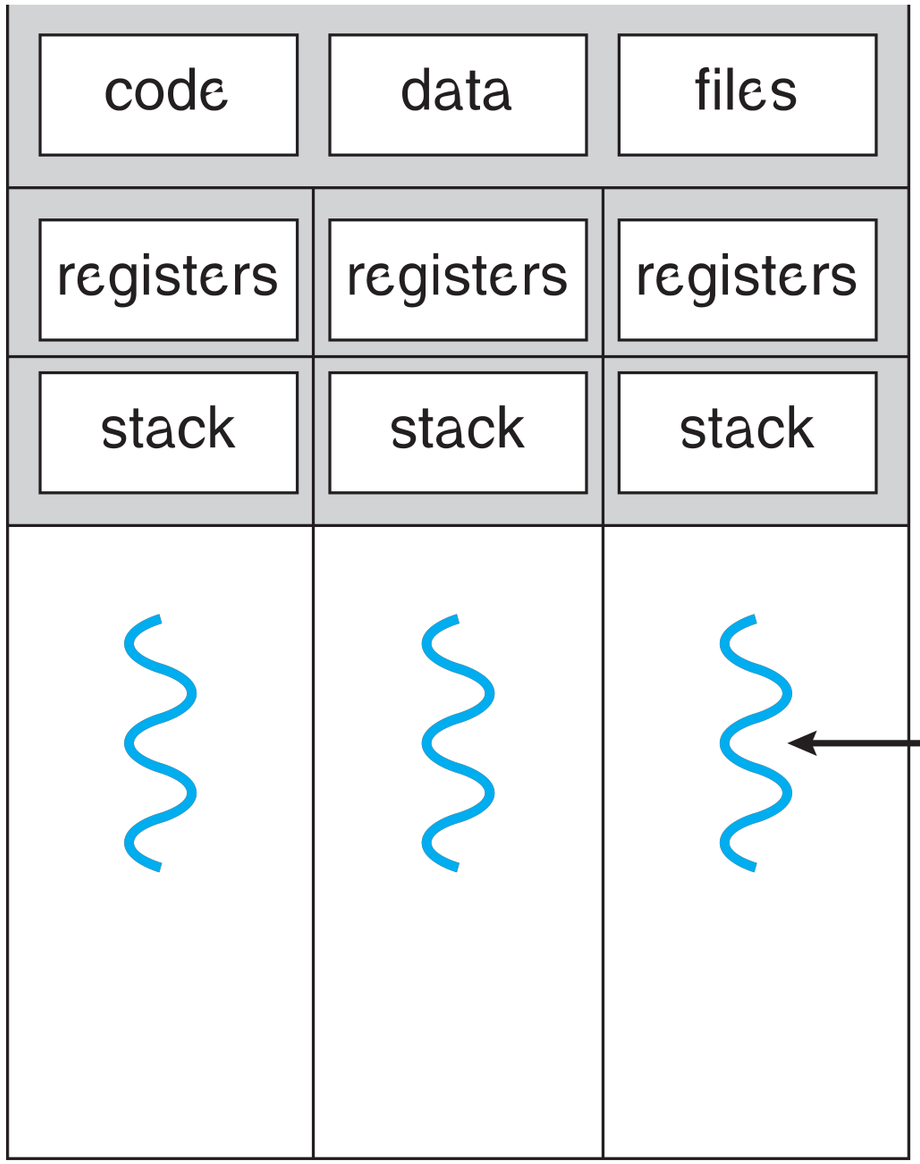


- Parallelism on a multi-core system

Single and Multithreaded Processes



single-threaded process



multithreaded process

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

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

- 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?

User Threads and Kernel Threads

- User threads - Management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

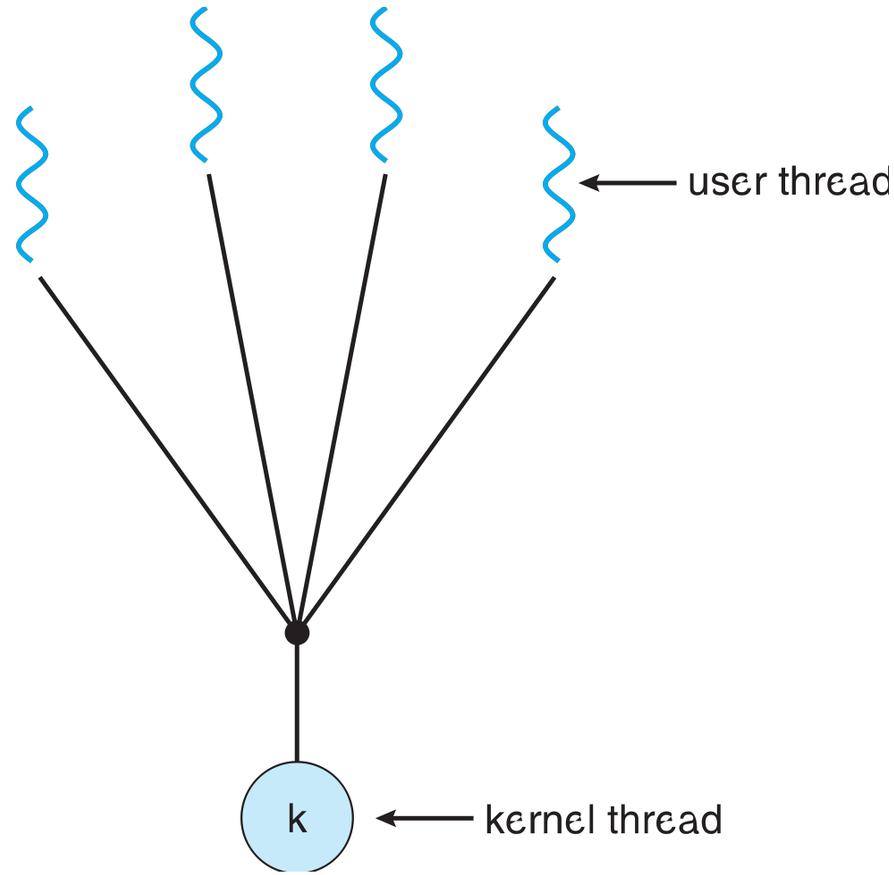
- Kernel threads - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

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

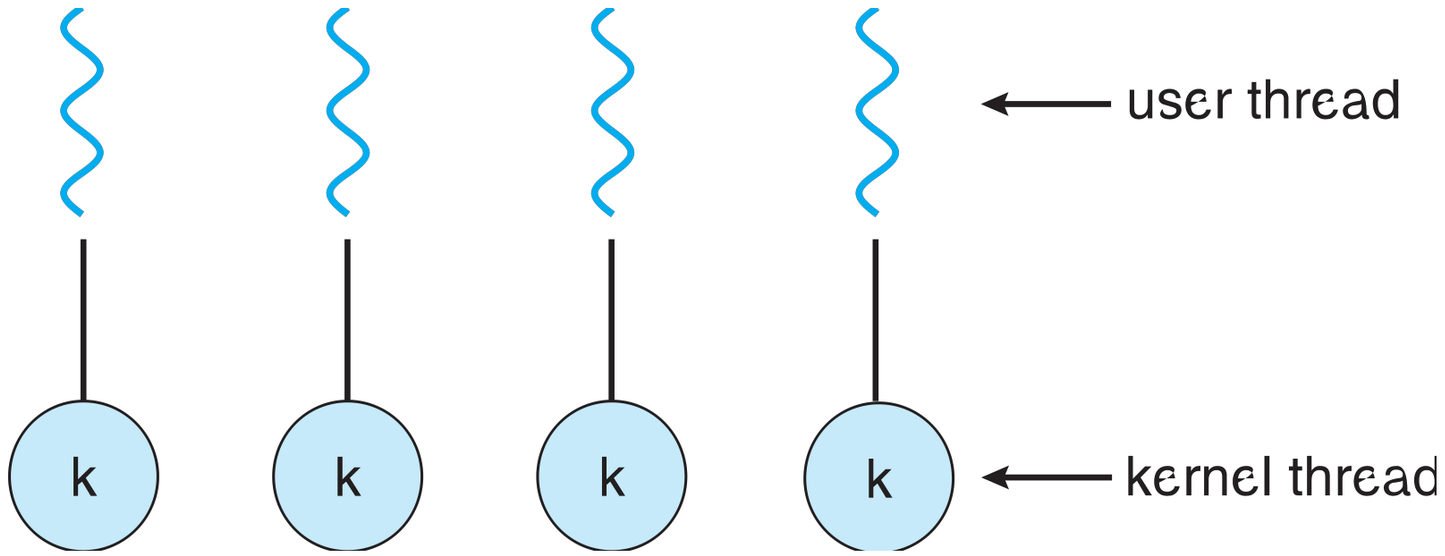
Many-to-One

- 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



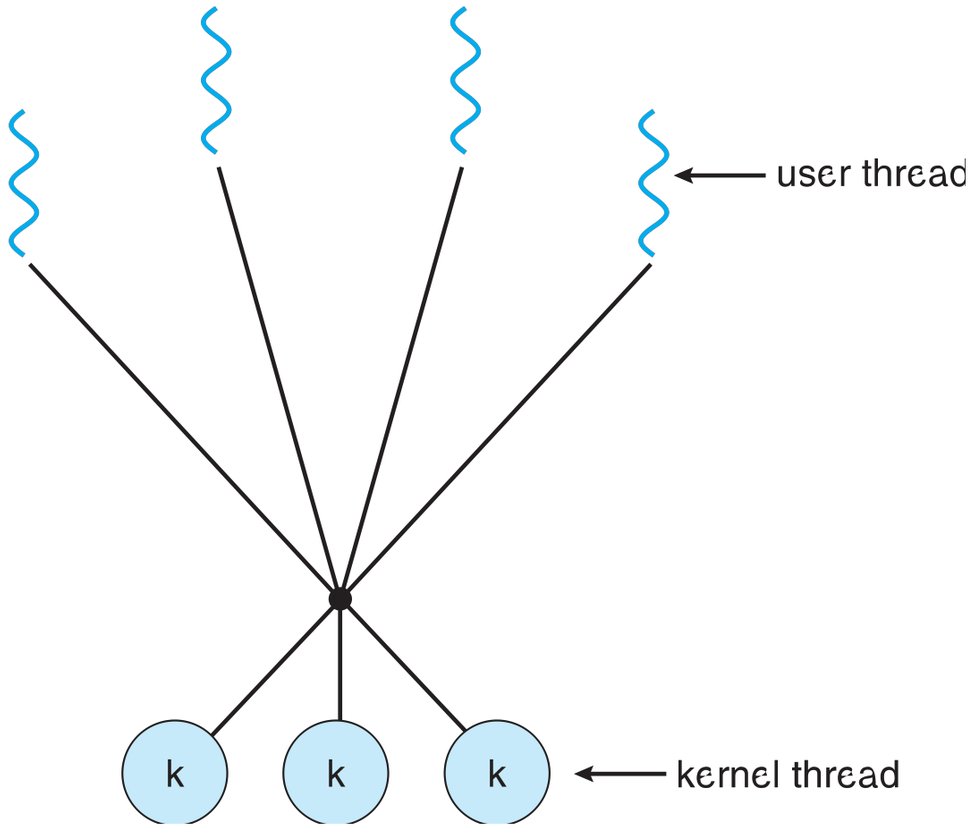
One-to-One

- 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 NT/XP/2000
  - Linux
  - Solaris 9 and later



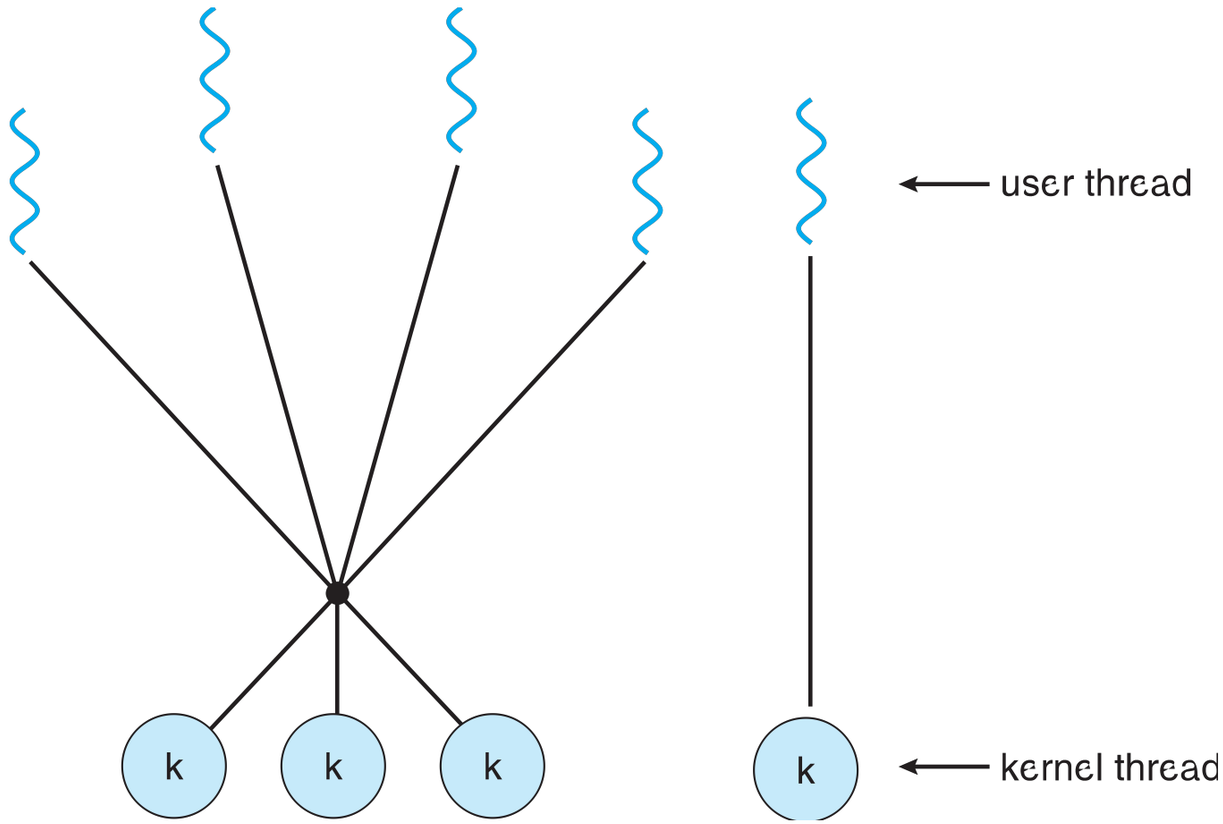
Many-to-Many Model

- 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 NT/2000 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



Thread Libraries

- Thread library provides programmer with API for creating and managing threads

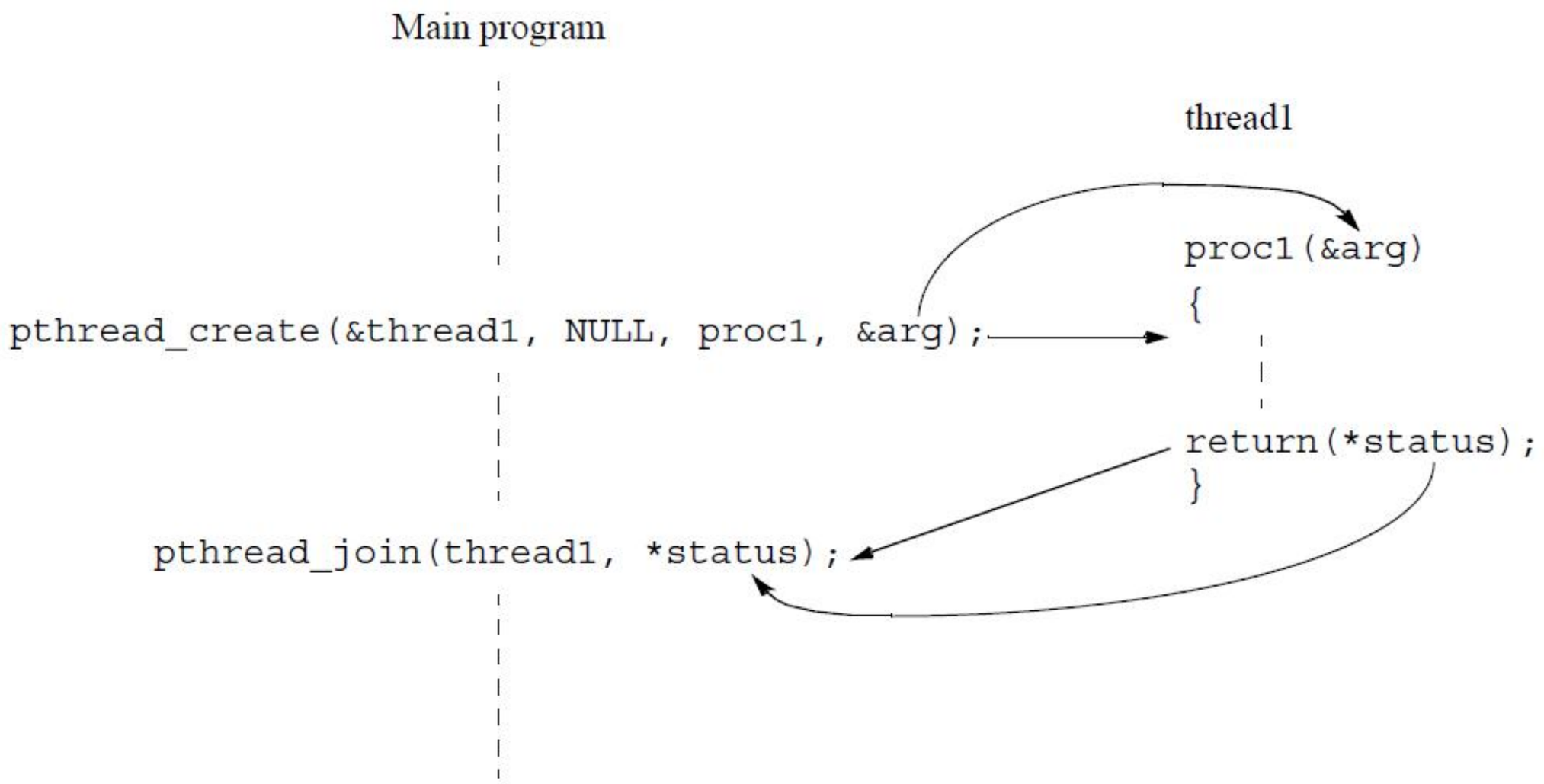
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads (1)

- 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 (2)

Executing a Pthread Thread



Pthreads (3)

Pthread Barrier

The routine `pthread_join()` waits for one specific thread to terminate.

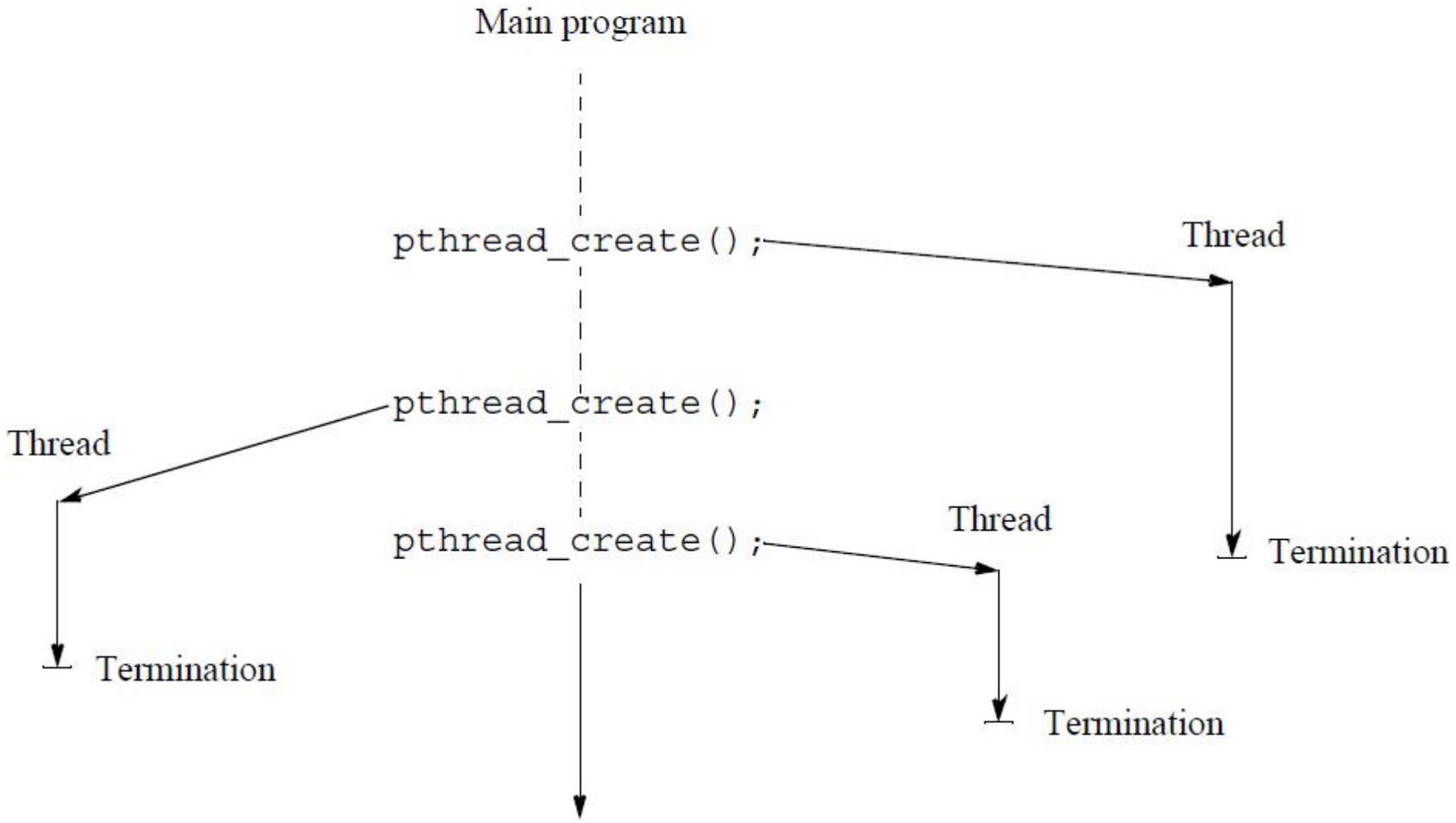
To create a barrier waiting for all threads, `pthread_join()` could be repeated:

```
.
for (i = 0; i < n; i++)
      pthread_create(&thread[i], NULL, (void *) slave, (void *) &arg);
.
for (i = 0; i < n; i++)
      pthread_join(thread[i], NULL);
.
```

Pthreads (4)

Detached Threads

It may be that thread may not be bothered when a thread it creates terminates and in that case a join not be needed. Threads that are not joined are called *detached threads*.



Pthreads (5)

Statement Execution Order

On a multiprocessor system, instructions of individual processes/threads might be interleaved in time.

Example

Process 1	Process 2
Instruction 1.1	Instruction 2.1
Instruction 1.2	Instruction 2.2
Instruction 1.3	Instruction 2.3

there are several possible orderings, including

Instruction 1.1
Instruction 1.2
Instruction 2.1
Instruction 1.3
Instruction 2.2
Instruction 2.3

Pthreads (6)

Thread-Safe Routines

System calls or library routines are called *thread safe* if they can be called from multiple threads simultaneously and always produce correct results.

Example

Standard I/O thread safe (prints messages without interleaving the characters).

Routines that access shared/static data may require special care to be made thread safe.

Example

System routines that return time may not be thread safe.

The thread-safety aspect of any routine can be avoided by forcing only one thread to execute the routine at a time. This could be achieved by simply enclosing the routine in a critical section but this is very inefficient.

Pthreads (7)

Accessing Shared Data

Consider two processes each of which is to add one to a shared data item, `x`. Necessary for the contents of the `x` location to be read, `x + 1` computed, and the result written back to the location. With two processes doing this at approximately the same time, we have

Time ↓

Instruction	Process 1	Process 2
`x = x + 1;`	`read x`	`read x`
	`compute x + 1`	`compute x + 1`
	`write to x`	`write to x`

Pthreads (8)

Critical Section

A mechanism for ensuring that only one process accesses a particular resource at a time is to establish sections of code involving the resource as so-called *critical sections* and arrange that only one such critical section is executed at a time.

The first process to reach a critical section for a particular resource enters and executes the critical section.

The process prevents all other processes from their critical sections for the same resource.

Once the process has finished its critical section, another process is allowed to enter a critical section for the same resource.
This mechanism is known as *mutual exclusion*.

Pthreads (9)

Locks

The simplest mechanism for ensuring mutual exclusion of critical sections.

A lock is a 1-bit variable that is a 1 to indicate that a process has entered the critical section and a 0 to indicate that no process is in the critical section.

The lock operates much like that of a door lock.

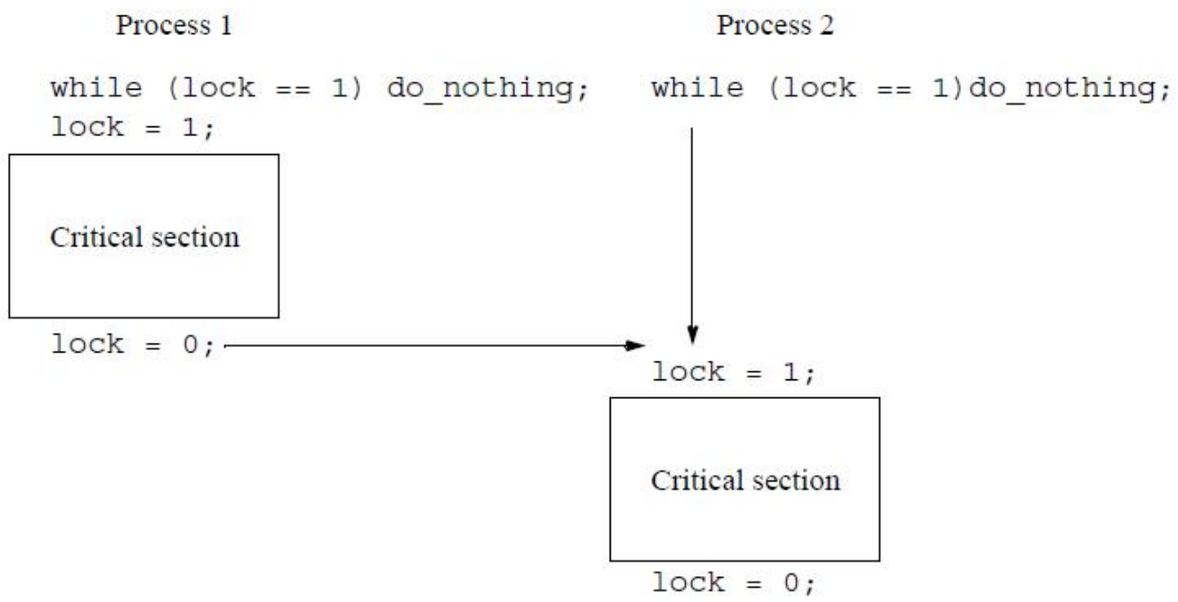
A process coming to the "door" of a critical section and finding it open may enter the critical section, locking the door behind it to prevent other processes from entering. Once the process has finished the critical section, it unlocks the door and leaves.

Pthreads (10)

Spin Lock

Example

```
while (lock == 1) do_nothing;        /* no operation in while loop */
lock = 1;                            /* enter critical section */
                .
        critical section
                .
lock = 0;                            /* leave critical section */
```



Pthreads (11)

Pthread Lock Routines

Locks are implemented in Pthreads with *mutually exclusive lock* variables, or "mutex" variables.

To use mutex, it must be declared as of type `pthread_mutex_t` and initialized:

```
pthread_mutex_t mutex1;
        .
        .
pthread_mutex_init(&mutex1, NULL);
```

NULL specifies a default attribute for the mutex.

A mutex can be destroyed with `pthread_mutex_destroy()`.

Pthreads (12)

A critical section can then be protected using `pthread_mutex_lock()` and `pthread_mutex_unlock()`:

```
pthread_mutex_lock(&mutex1);
        .
    critical section
        .
pthread_mutex_unlock(&mutex1);
```

If a thread reaches a mutex lock and finds it locked, it will wait for the lock to open.

If more than one thread is waiting for the lock to open when it opens, the system will select one thread to be allowed to proceed.

Only the thread that locks a mutex can unlock it.

Pthreads (13)

Condition Variables

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

With locks, the global variable would need to be examined at frequent intervals ("polled") within a critical section.

This is a very time-consuming and unproductive exercise. The problem can be overcome by introducing so-called *condition variables*.

Pthreads (14)

Condition Variable Operations

Three operations are defined for a condition variable:

`Wait(cond_var)` — wait for a condition to occur

`Signal(cond_var)` — signal that the condition has occurred

`Status(cond_var)` — return the number of processes waiting for the condition to occur

The wait operation will also release a lock or semaphore and can be used to allow another process to alter the condition.

When the process calling `wait()` is finally allowed to proceed, the lock or semaphore is again set.

Pthreads (15)

Example

Consider one or more processes (or threads) designed to take action when a counter, x, is zero. Another process or thread is responsible for decrementing the counter. The routines could be of the form

```
action()                          counter()
{                                 {
    .                                 .
  lock();                           lock();
  while (x != 0)                    x--;
    wait(s);  <-------------------  if (x == 0) signal(s);
  unlock();                         unlock();
  take_action();                      .
    .                                 .
}                                 }
```

Pthreads (16)

Pthread Condition Variables

Associated with a specific mutex. Given the declarations and initializations:

```
pthread_cond_t cond1;
pthread_mutex_t mutex1;
pthread_cond_init(&cond1, NULL);
pthread_mutex_init(&mutex1, NULL);
```

the Pthreads arrangement for signal and wait is as follows:

```
action()
{
        .
        .
pthread_mutex_lock(&mutex1);
while (c <> 0)
  pthread_cond_wait(cond1,mutex1);
pthread_mutex_unlock(&mutex1);
take_action();
        .
        .
}
```

```
counter()
  {
      .
      .
pthread_mutex_lock(&mutex1);
c--;
if (c == 0) pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);
   .
   .
   .
}
```

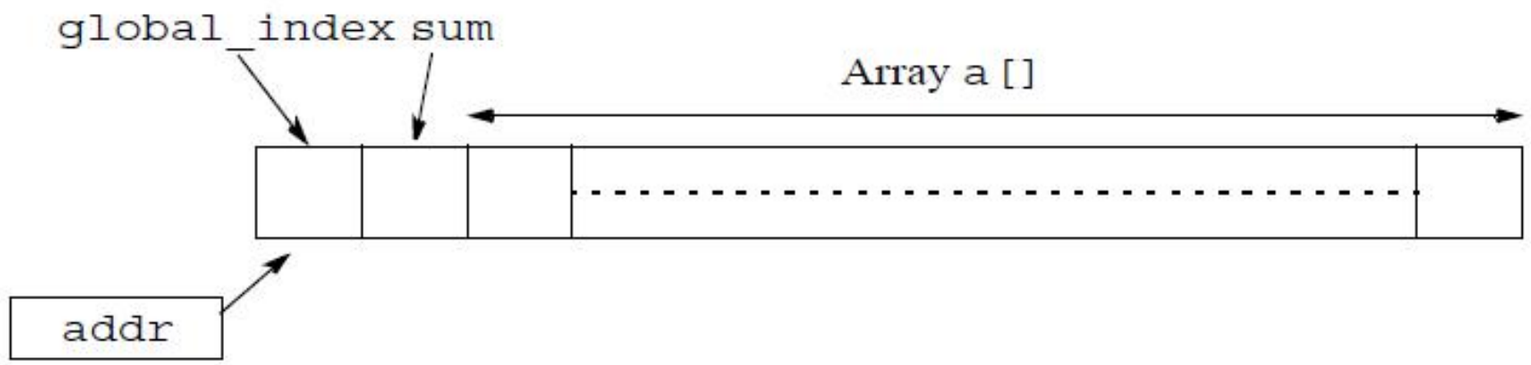
Signals are *not* remembered - threads must already be waiting for a signal to receive it.

Pthreads Example (1)

Program Examples

To sum the elements of an array, `a[1000]`:

```
int sum, a[1000];
        sum = 0;
        for (i = 0; i < 1000; i++)
                sum = sum + a[i];
```



Pthreads Example (2)

```
#include <stdio.h>
#include <pthread.h>
#define array_size 1000
#define no_threads 10
                                    /* shared data */
int a[array_size];                  /* array of numbers to sum */
int global_index = 0;               /* global index */
int sum = 0;                        /* final result, also used by slaves */
pthread_mutex_t mutex1;             /* mutually exclusive lock variable */
void *slave(void *ignored)          /* Slave threads */
{
int local_index, partial_sum = 0;
do {
   pthread_mutex_lock(&mutex1);/* get next index into the array */
      local_index = global_index;/* read current index & save locally*/
      global_index++;           /* increment global index */
   pthread_mutex_unlock(&mutex1);

   if (local_index < array_size) partial_sum += *(a + local_index);
} while (local_index < array_size);

pthread_mutex_lock(&mutex1);  /* add partial sum to global sum */
   sum += partial_sum;
pthread_mutex_unlock(&mutex1);

return ();                          /* Thread exits */
}
```

Pthreads Example (3)

```
  main () {
int i;
pthread_t thread[10];                    /* threads */
pthread_mutex_init(&mutex1,NULL);        /* initialize mutex */

for (i = 0; i < array_size; i++)         /* initialize a[] */
  a[i] = i+1;

for (i = 0; i < no_threads; i++)         /* create threads */
  if (pthread_create(&thread[i], NULL, slave, NULL) != 0)
    perror("Pthread_create fails");

for (i = 0; i < no_threads; i++)         /* join threads */
  if (pthread_join(thread[i], NULL) != 0)
    perror("Pthread_join fails");
printf("The sum of 1 to %i is %d\n", array_size, sum);
}                                        /* end of main */
```

SAMPLE OUTPUT

```
The sum of 1 to 1000 is 500500
```

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
- 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

- 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 by one of two signal handlers:
     1. default
     2. user-defined
- 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
- 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 (1)

- 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 (2)

- 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

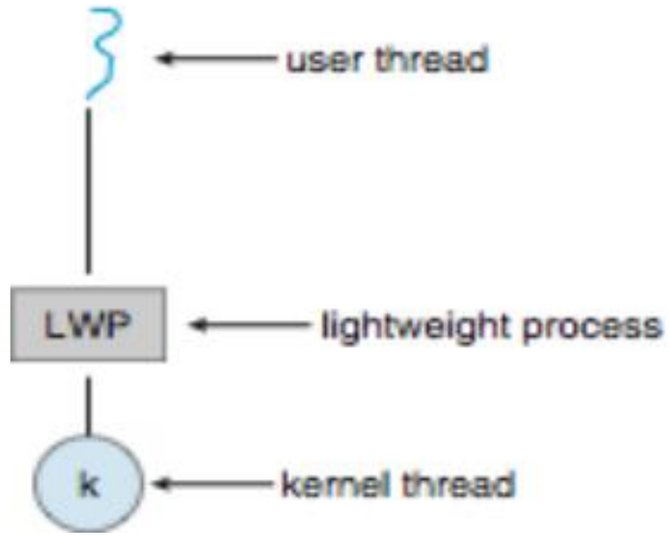
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

- 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



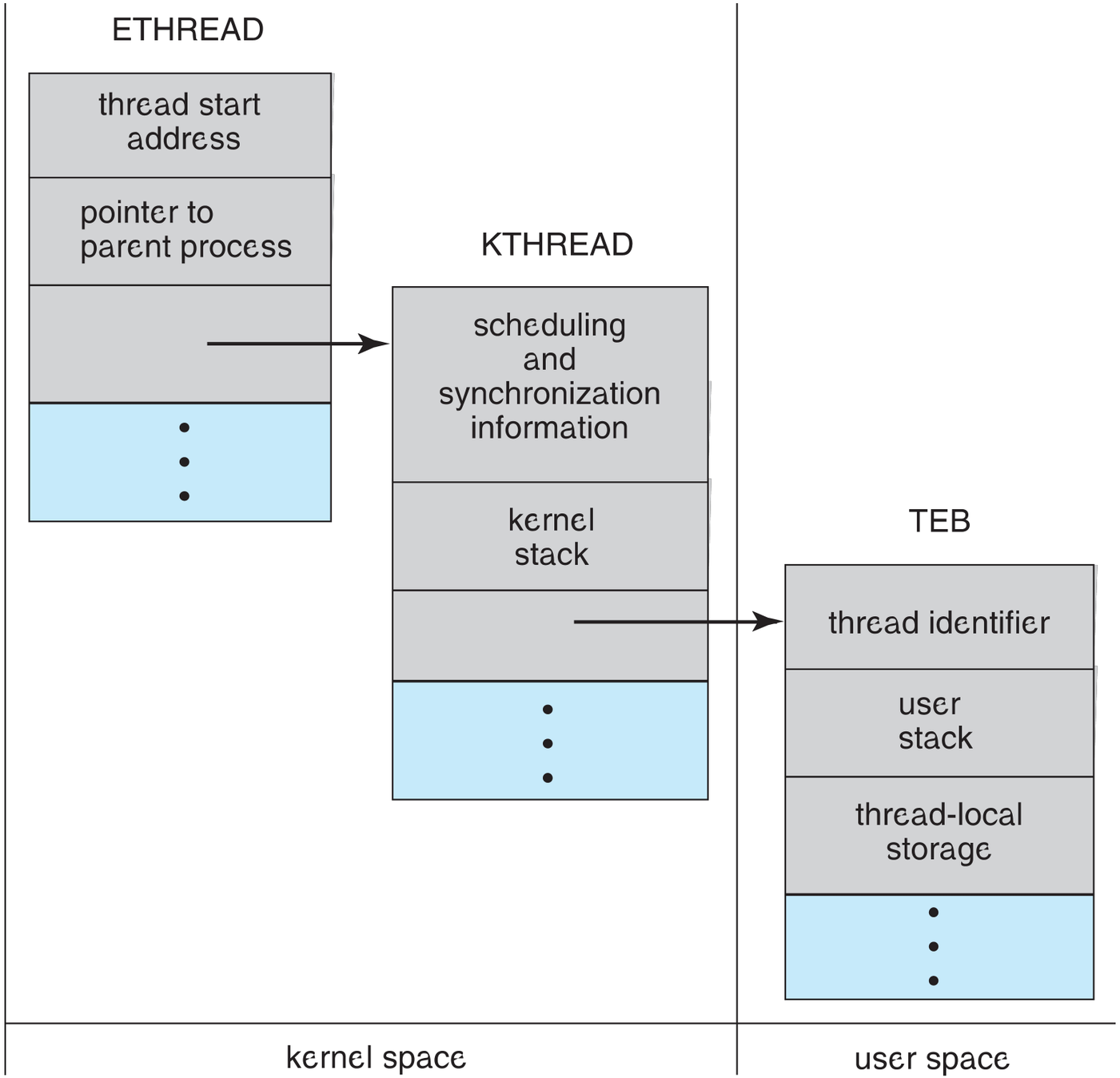
Operating System Examples

- Windows XP Threads
- Linux Thread

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
- 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 XP Threads Data Structures



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)