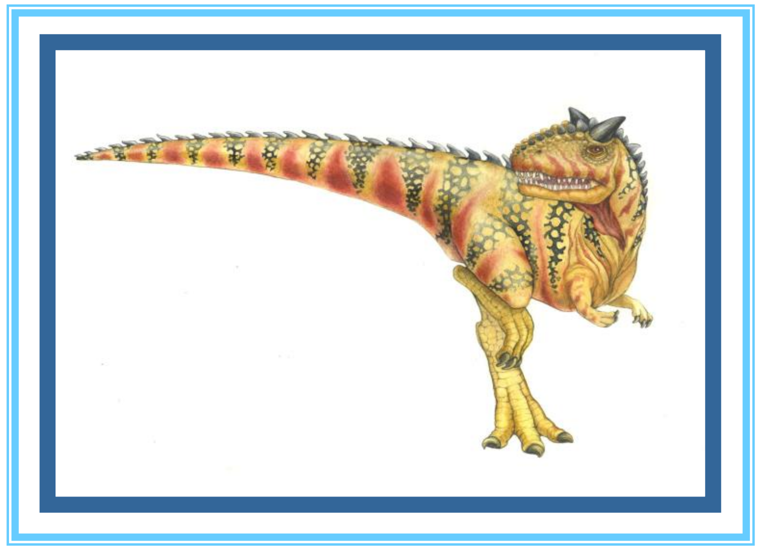
Chapter 4: Threads & Concurrency

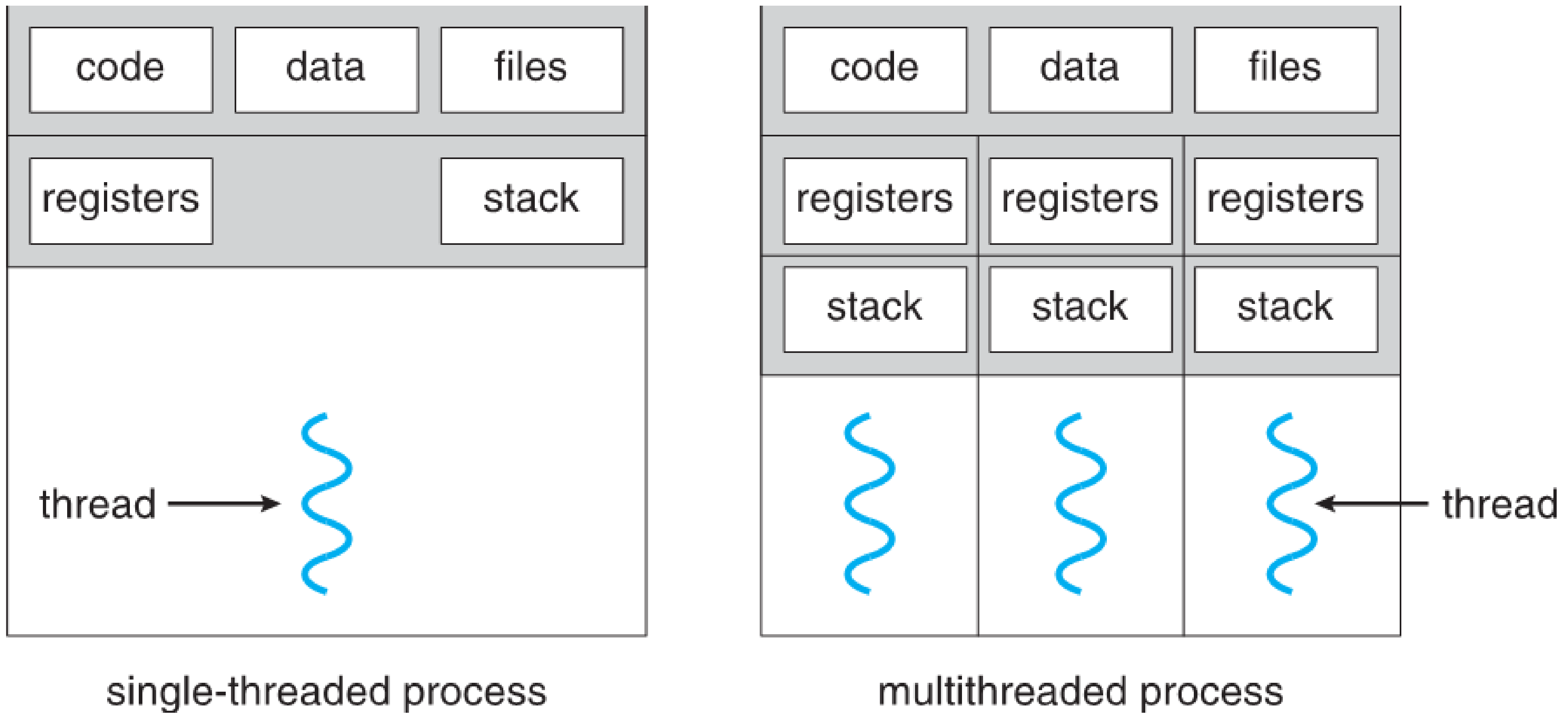


Chapter 4: Multithreaded Programming

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

Thread Overview

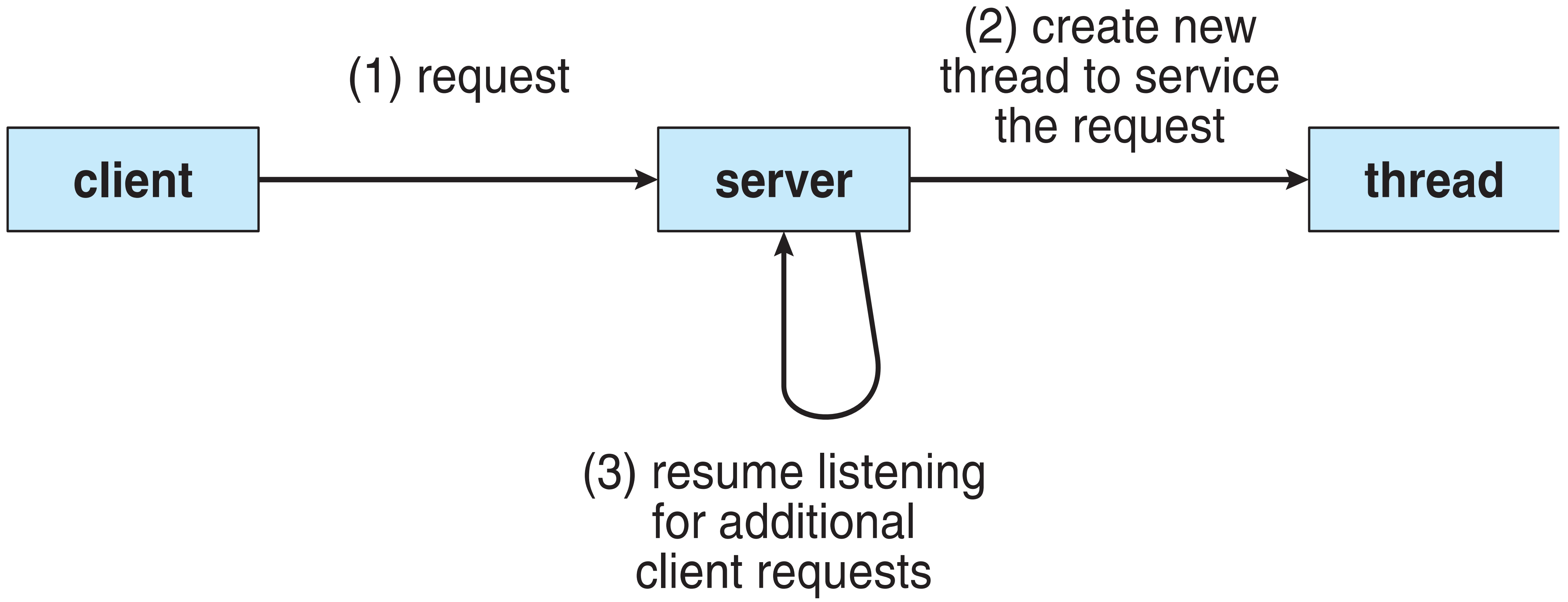
- Traditional (heavyweight) processes have a single thread of control.
- Multi-threaded applications have multiple threads within a single process, each having their own program counter, stack and set of registers, but sharing common code, data, and certain structures such as open files.



Motivation

- Most modern applications are multithreaded
  - Threads run within application
  - Can simplify code, increase efficiency
  - 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
  - Allocating memory and resource for process creation is costly
- Kernels are generally multithreaded,
  - each thread performs a specific task, such as managing devices, managing memory, or interrupt handling

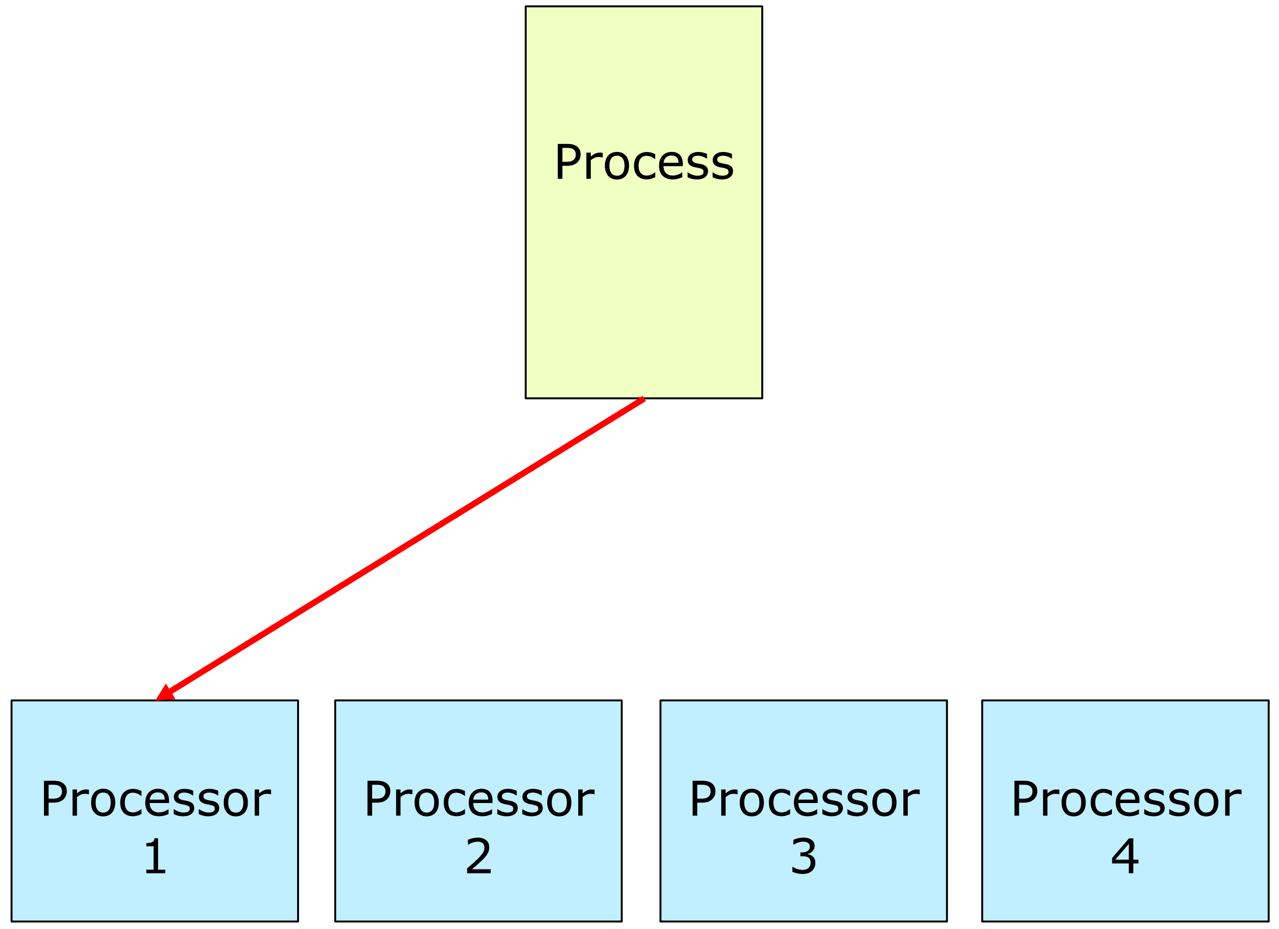
Multithreaded Server Architecture


(1) request
client
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

Consider this example

```
#include <stdio.h>

unsigned long addall() {
   int i=0;
   unsigned long sum = 0;

   while( i< 10000000) {
       sum += i;
       i++;
   }
   return sum;
 }

int main()
{
      unsigned long sum;
      sum = addall();
      printf("%lu\n", sum);
}
```

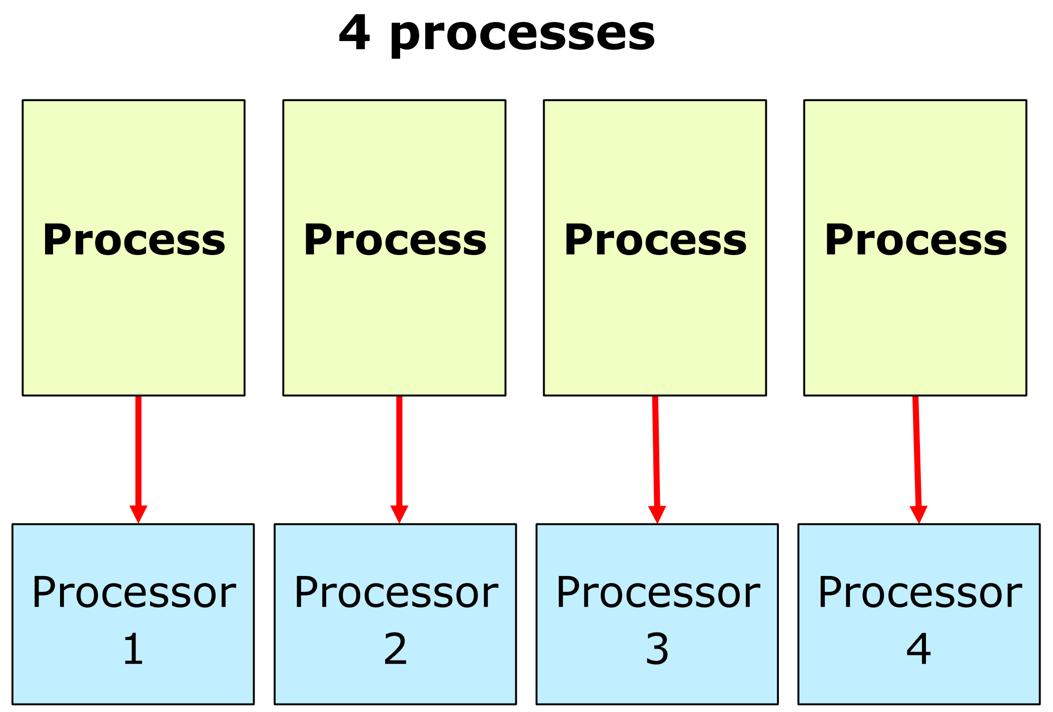
- Single-thread process cannot take advantage of multiple processors to speed up the computation.

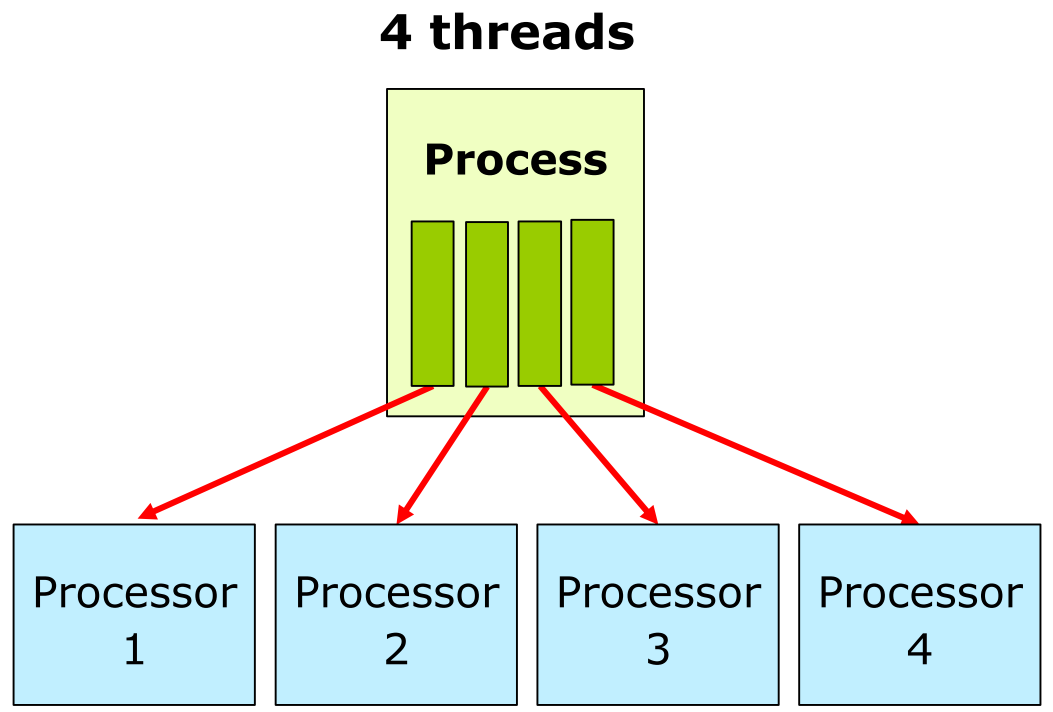


Consider this example

- Speed up the computation
  - Divide the loop into 4 loops, each does 1/4th of the work

➔ 10000000 / 4 = 2500000





Process creation is heavier
- Each process is isolated from each other
- Each has its own memory map
  (instruction, global data, heap, stack, etc.)

Use IPC to communicate the result

Thread creation is much lighter.
- The 4 threads share instructions, global data, and heap regions.
- Each thread has its own stack.

Use global data to communicate the result

Benefits – efficient communication

Example: Create 1 process with 4 threads; each does 1/4th of the work

➔ 10000000 / 4 = 2500000

```
#include <pthread.h>
#include <stdio.h>
unsigned long sum[4];

void *thread_fn( void *arg ) {
    long id  = (long) arg;
    int start  = id * 2500000;
    int i = 0;
    while( i < 2500000 ) {
        sum[id] += (i + start);
        i++;
    }
    return NULL;
}
```

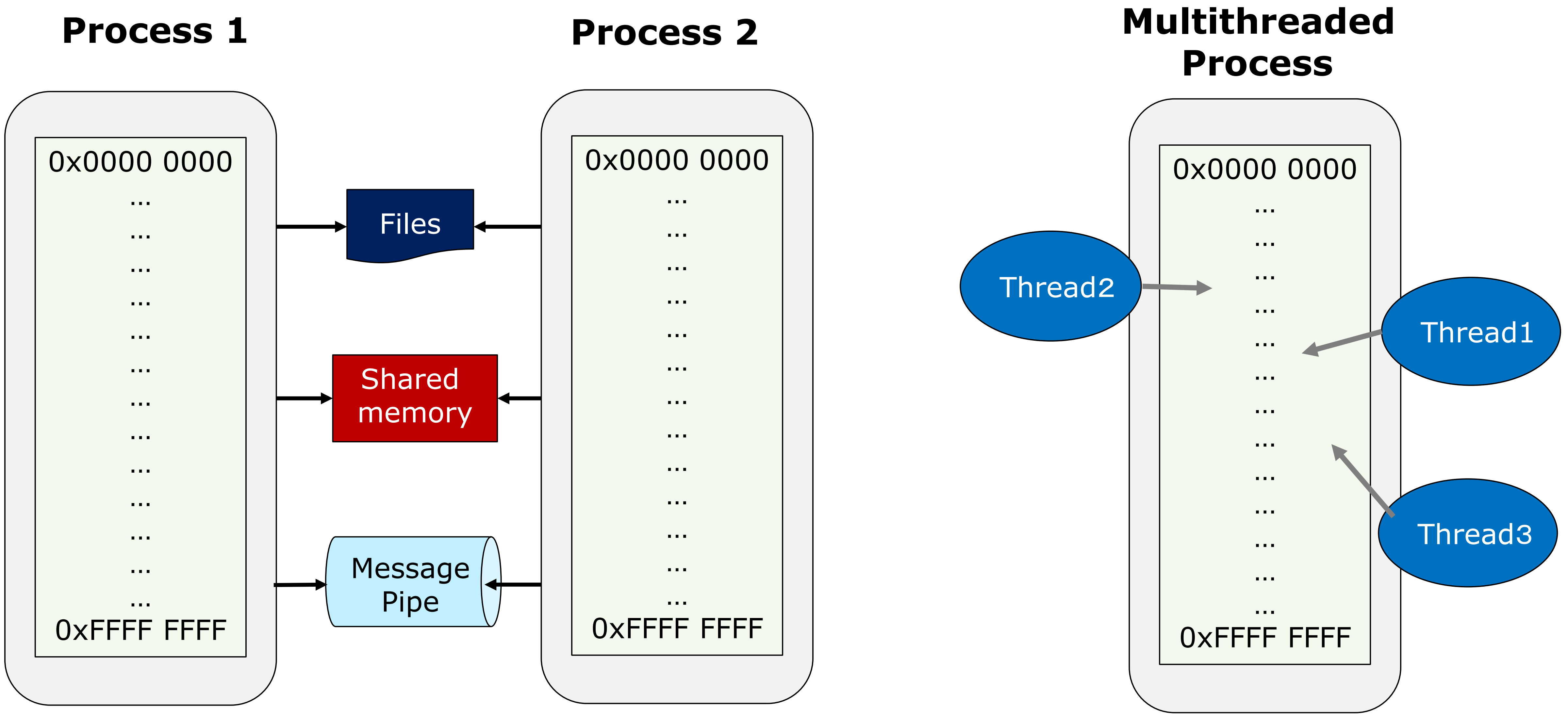
```
int main()
{
    pthread_t t1, t2, t3, t4;

    pthread_create( &t1, NULL, thread_fn, (void *)0 );
    pthread_create( &t2, NULL, thread_fn, (void *)1 );
    pthread_create( &t3, NULL, thread_fn, (void *)2 );
    pthread_create( &t4, NULL, thread_fn, (void *)3 );

    pthread_join( t1, NULL);
    pthread_join( t2, NULL);
    pthread_join( t3, NULL);
    pthread_join( t4, NULL);

    sum[0] = sum[0]+sum[1]+sum[2]+sum[3];
    return 0;
}
```

You need to link pthread library

```
$ gcc threads.c  -lpthread
$ ./a.out
```

Benefits – efficient communication

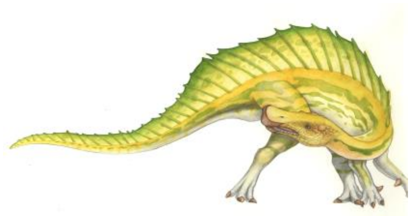
Processes vs Threads



Processes vs Threads

Processes vs Threads

- A process can live on its own.
  - A thread cannot live on its own. It needs to be attached to a process.
- A process has at-least one thread. There can be more than one thread in a process.
- A process has code, heap, stack, other segments.
  - Threads within a process share the same code, files. But, each thread has its own stack.
- If a process dies, all threads die.
  - If a thread dies, its stack is reclaimed.



Benefits

- **Responsiveness**
  - may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing (for efficient communication)**
  - threads share resources of process, easier than shared memory or message passing
- **Economy (lightweight)**
  - Thread creation is cheaper than process creation (30 times speedup)
    - allocating memory and resource for process creation is costly
  - Context switching: thread switching has lower overhead than process switching
    - threads share common code, data, address space, page table, and etc.
- **Scalability**
  - Multithread applications can take advantage of multiprocessor architectures. A single-threaded process can run on only one processor.

Chapter 4: Multithreaded Programming

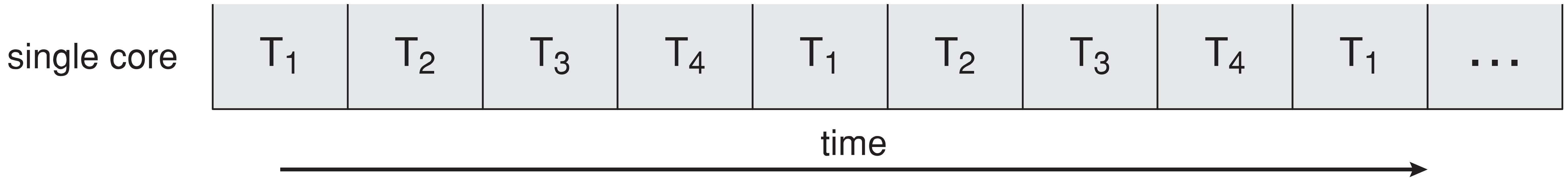
- Thread Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Multiprocessor Programming

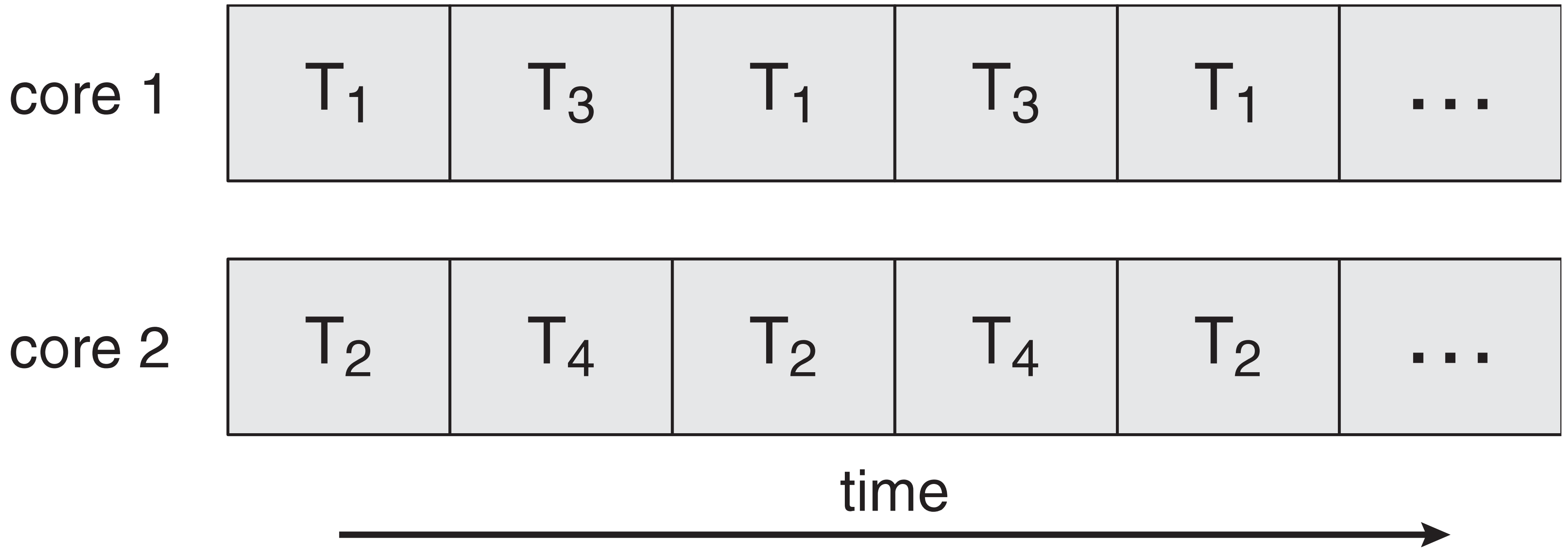
- **Multicore** or **multiprocessor** systems putting pressure on programmers
  - dividing activities, balance, data splitting, data dependency, and debug
- ***Concurrency*** : can support more than one tas k by allowing all the tasks to make progress by switching between tasks.
- ***Parallelism*** : can perform more than one task simultaneously
  - **Data parallelism** – distributes subsets of the data across multiple cores, same operation on each. (E.g., dividing a large image up into pieces and performing the same processing on each piece on different cores.)
  - **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***
  - e.g., Oracle SPARC T4: 8 cores, and 8 hardware threads per core

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



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

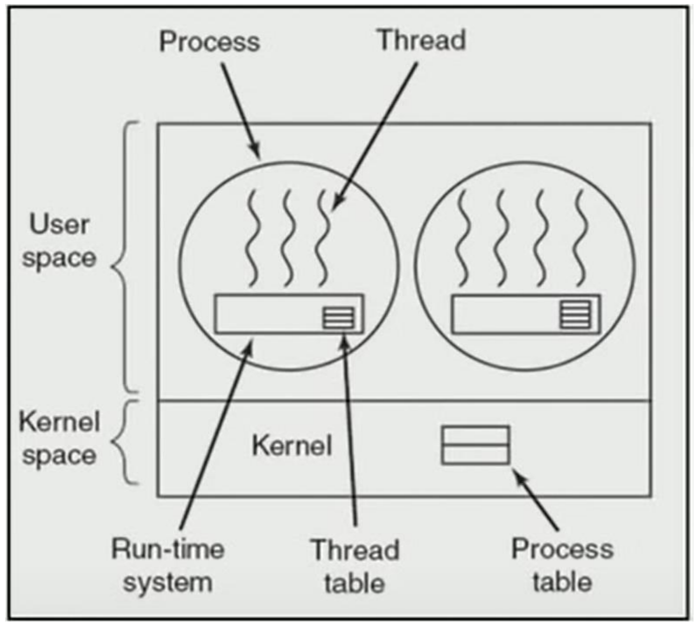
$$speedup \le \frac{1}{S + \frac{(1-S)}{N}}$$

- I.e. 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*

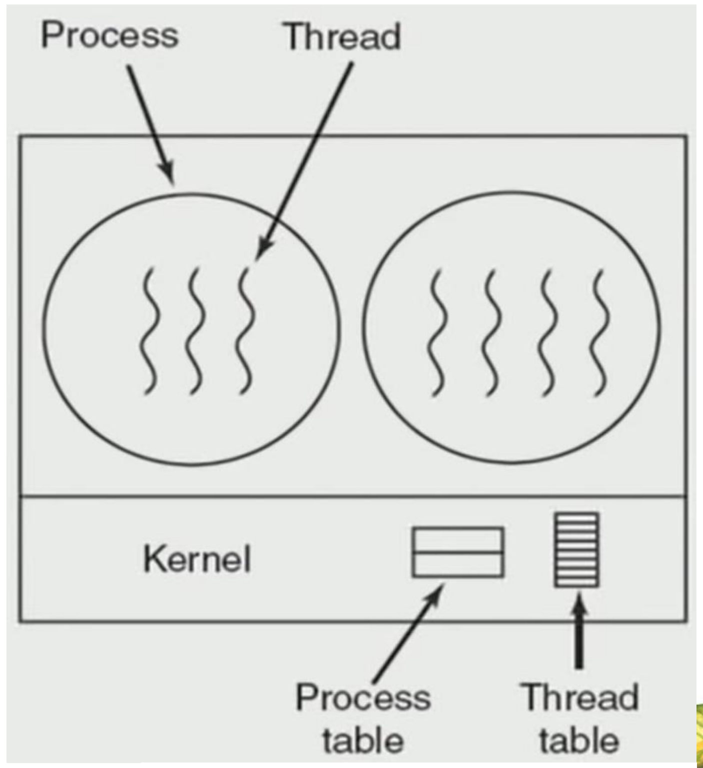
Serial portion of an application has disproportionate effect on performance gained by adding additional cores

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library. Kernel knows nothing about the thread.
- **Kernel threads** - Supported by the Kernel
  - Almost all general purpose OS support kernel threads
  - Windows, Solaris, Linux, Tru64 UNIX, Mac OS X
- Three primary thread libraries:
  - POSIX Pthread, Windows thread, Java thread
- For systems supporting both user and kernel threads, there are several different **Multithreading Models**
  - ***Many-to-one, One-to-one, Many-to-many, two-level model***



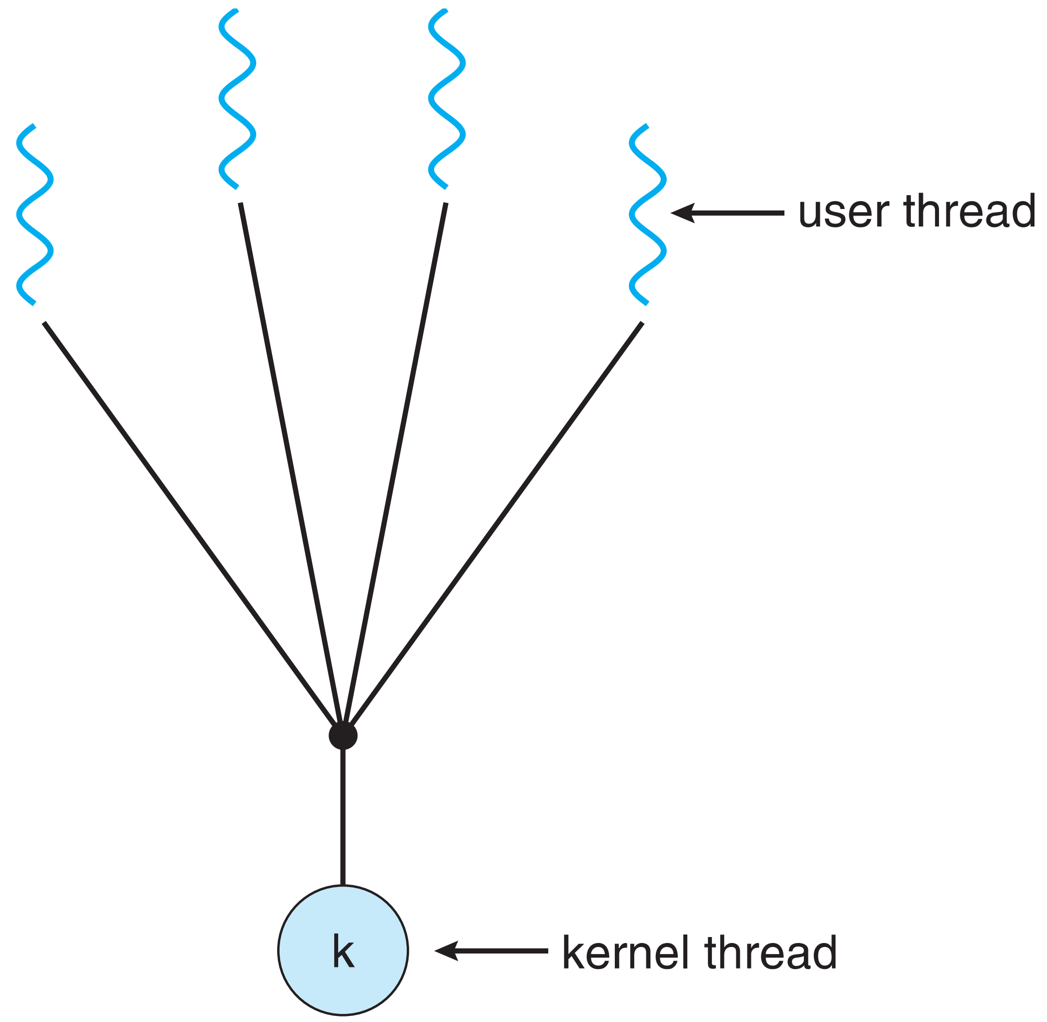
User threads



Kernel threads

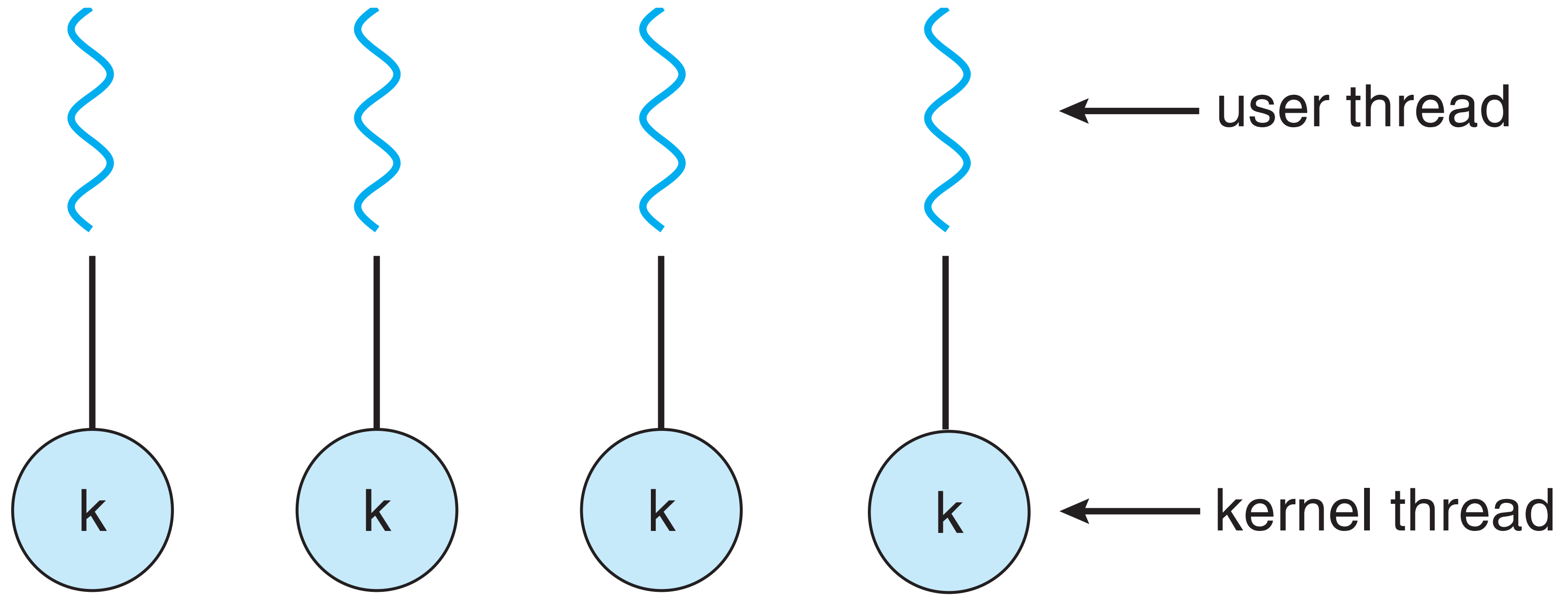
Multithreading Models: Many-to-One

- Many user-level threads mapped to single kernel thread
  - One thread blocking causes all to block
  - Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model (because of its inability to take advantage of multiple processing cores)
- Examples:
  - Solaris Green Threads



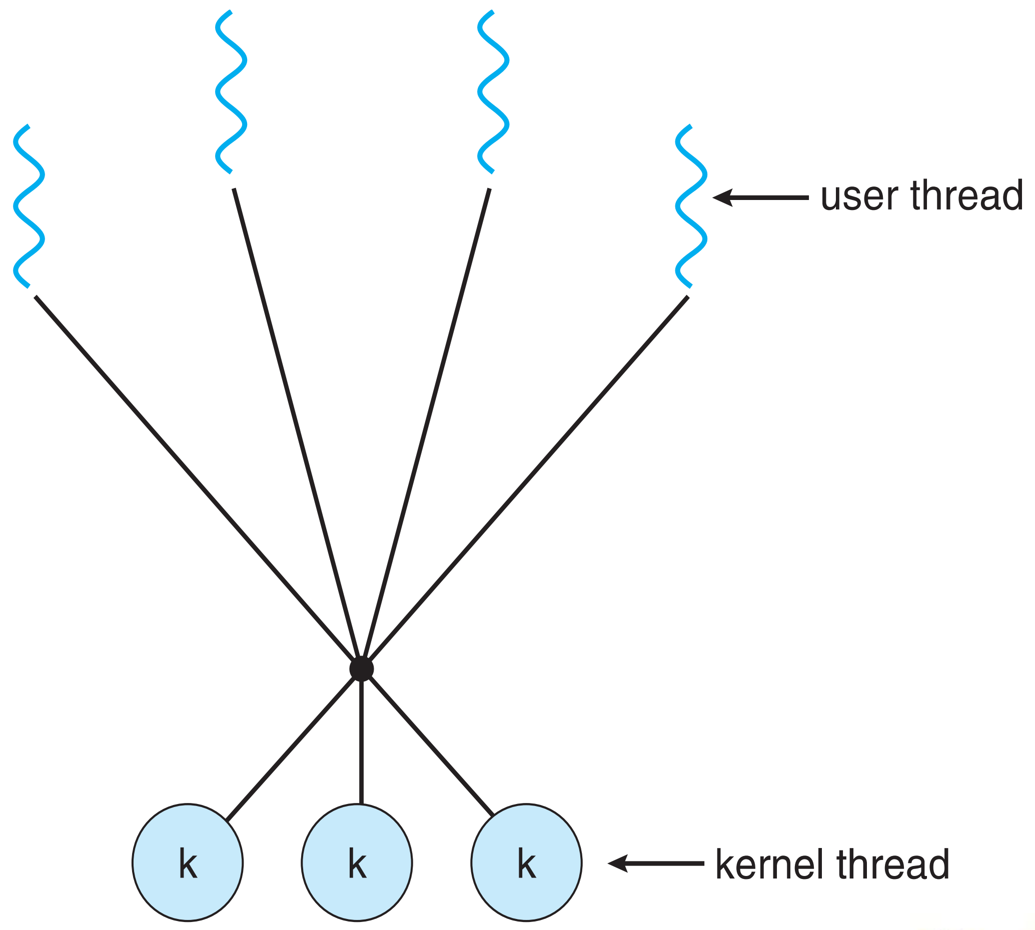
Multithreading Models: One-to-One

- Each user-level thread maps to kernel thread
  - Creating a user-level thread creates a kernel thread
  - Another thread can run when one thread is blocked
  - Allow multiple threads to run in parallel on multiprocessors
  - Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



Multithreading Models: Many-to-Many Model

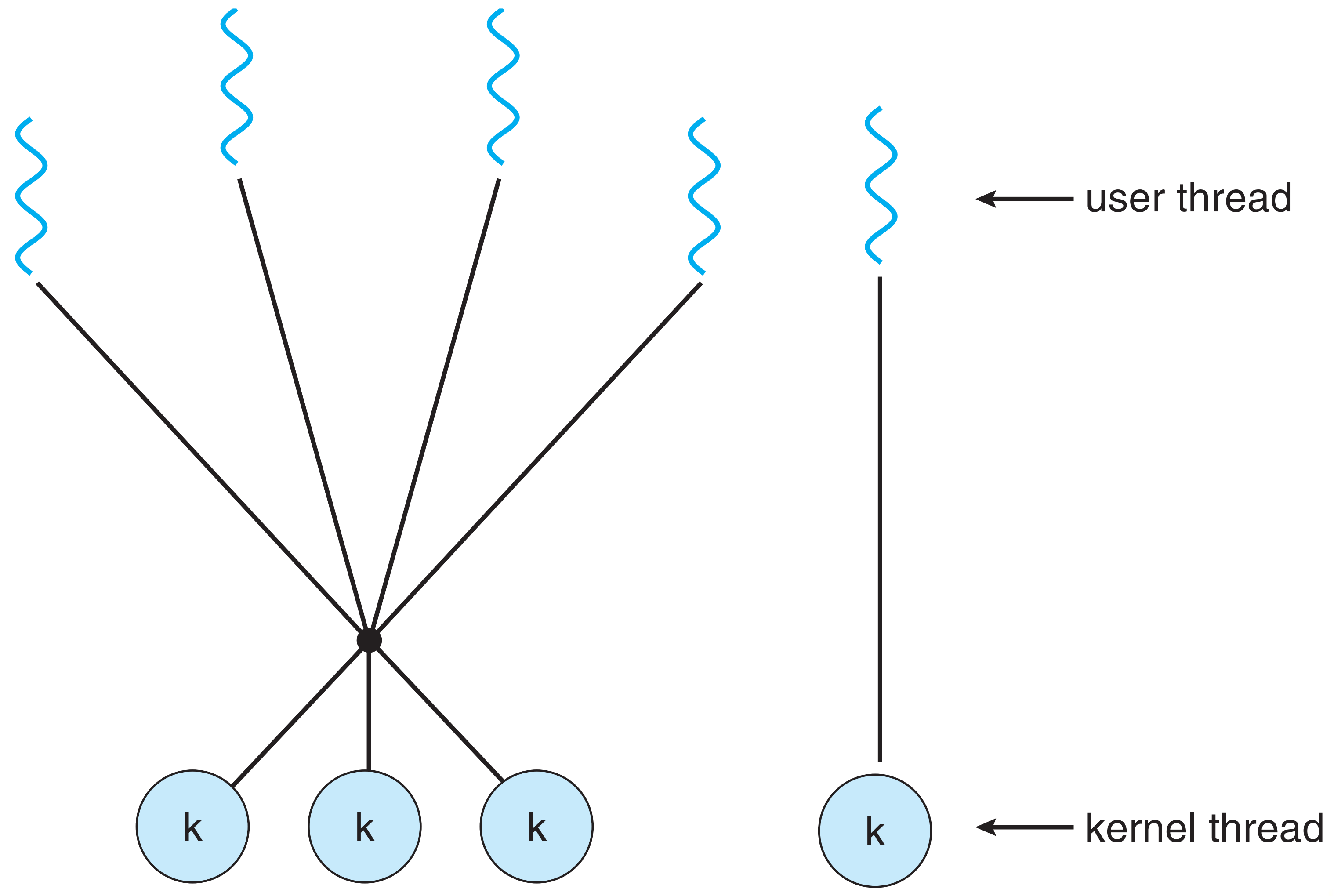
- Allows many user level threads to be mapped to many kernel threads
- Allows operating system to create a sufficient number of kernel threads
  - Equal or fewer than user threads.
- User threads can be created as many as necessary.
- Blocking kernel system calls do not block the entire process (The kernel can schedule another thread for execution).
- Example
  - Solaris prior to version 9
  - Windows NT/2000 with *ThreadFiber* package



Multithreading Models: Two-level Model

- Similar to M:M, except that it also allows a user thread to be **bound** to a kernel thread (one-to-one).

- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Chapter 4: Multithreaded Programming

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

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
    - All code and data structures for the library exist in user space.
    - Invoking a function in the library results in a local function call in user space.
  - Kernel-level library supported by the OS
    - A kernel-level library supported directly by the OS.
    - Invoking a function in the library results in a system call to the kernel

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

- Create a thread in a process

```
int pthread_create( pthread_t *thread,              → Thread identifier (TID)
                    const pthread_attr_t *attr,
                    void *(*start_routine)(void *), → Pointer to a function, which starts
                                                      execution in a different thread
                    void *arg);                     → Arguments to the function
```

- Wait for a specific thread to complete: **join**

```
int pthread_join(pthread_t thread, void **retval);
```

void **retval → Exit status of the thread

pthread_t thread → TID of the thread to wait for

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);
}
```

parent will wait for the termination of the thread and then continue its own execution

***sum** is a global variable shared by the two threads*

*The runner thread will complete by calling **exit**.*

Figure 4.9 Multithreaded C program using the Pthreads API.

Win32 API 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;
   }
```

Win32 API 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 two ways
  1. Extending Thread class
  2. Implementing the **Runnable interface**

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

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>"); }
}
```

shared object

which, after initialization, will call **run()**

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 (GCD)**
- Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package

Thread Pools

- Create a number of threads in a pool when the process first starts
  - Threads are allocated from the pool as needed, and returned to the pool when no longer needed.
  - When no threads are available in the pool, the process may have to wait until one becomes available.
- 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.
- 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
{
    /*some parallel code here */
}
```

Create as many threads as there are cores

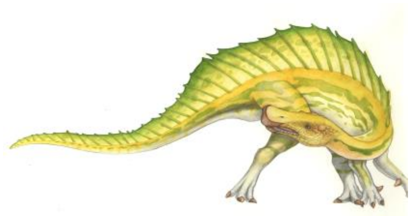
```
#pragma omp parallel for
 for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

Run for loop in parallel

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

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
     printf("parallel region.");
  }

  /* sequential code */

  return 0;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Allows developers to identify sections of code (tasks) to run in parallel
- GCD 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 the dispatch queue
- Two types of dispatch queues:
  - ***serial*** – blocks removed in FIFO order, queue is per process, called main queue
  - ***concurrent*** – removed in FIFO order but several may be removed at a time, thus allowing multiple tasks to execute in parallel.



Chapter 4: Multithreaded Programming

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

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

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:** 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: Default or 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
- **Issues: Where should a signal be delivered for multi-threaded?**
  - To the thread to which the signal applies – pthread_kill(pthread_t tid, int signal)
  - To every thread in the process - kill(pid_t pid, int signal) in UNIX
  - To certain threads in the process
  - Assign a specific thread to receive all signals for the process

Thread Cancellation

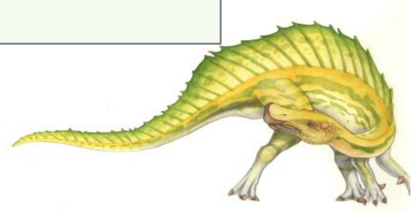
- Cancellation: to terminate a thread before it has completed
- Threads that are no longer needed may be cancelled by another thread:
  - **Asynchronous Cancellation** terminates the target thread immediately.
    - resource allocation and inter-thread data transfers can be problematic
  - **Deferred Cancellation** sets a flag indicating the target thread should cancel itself when it is convenient. It is then up to the target thread to check this flag periodically and exit nicely when it sees the flag set.

To cancel a thread in Pthread

```
pthread_t tid;

/* create the thread */
pthread.create( &tid, 0, worker, NULL );
…
/* cancel the thread */
pthread.cancel( tid );

/*wait for the thread to terminate*/
pthread_join( tid, NULL );
```

The created thread

```
while(1) {
    /* do some work for a awhile */

      ….

    /* check if there is a cancellation */
    pthread_testcancel();
}
```



Thread-Local Storage

- Threads belonging to a process share the data of the process, and this is one of the major benefits of using threads in the first place. However sometimes threads need thread-specific data also.

  ➔ Thread-local storage

- **Thread-local storage** (**TLS**) (thread-specific data)
  - allows each thread to have its own copy of data
  - Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data

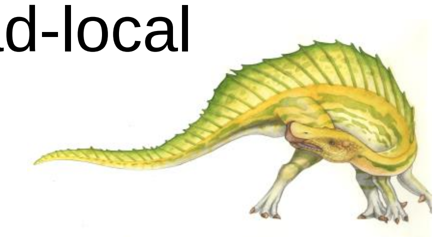
- **Different from local variables**
  - Local variables visible only during single function invocation
  - TLS visible across function invocations

Chapter 4: Multithreaded Programming

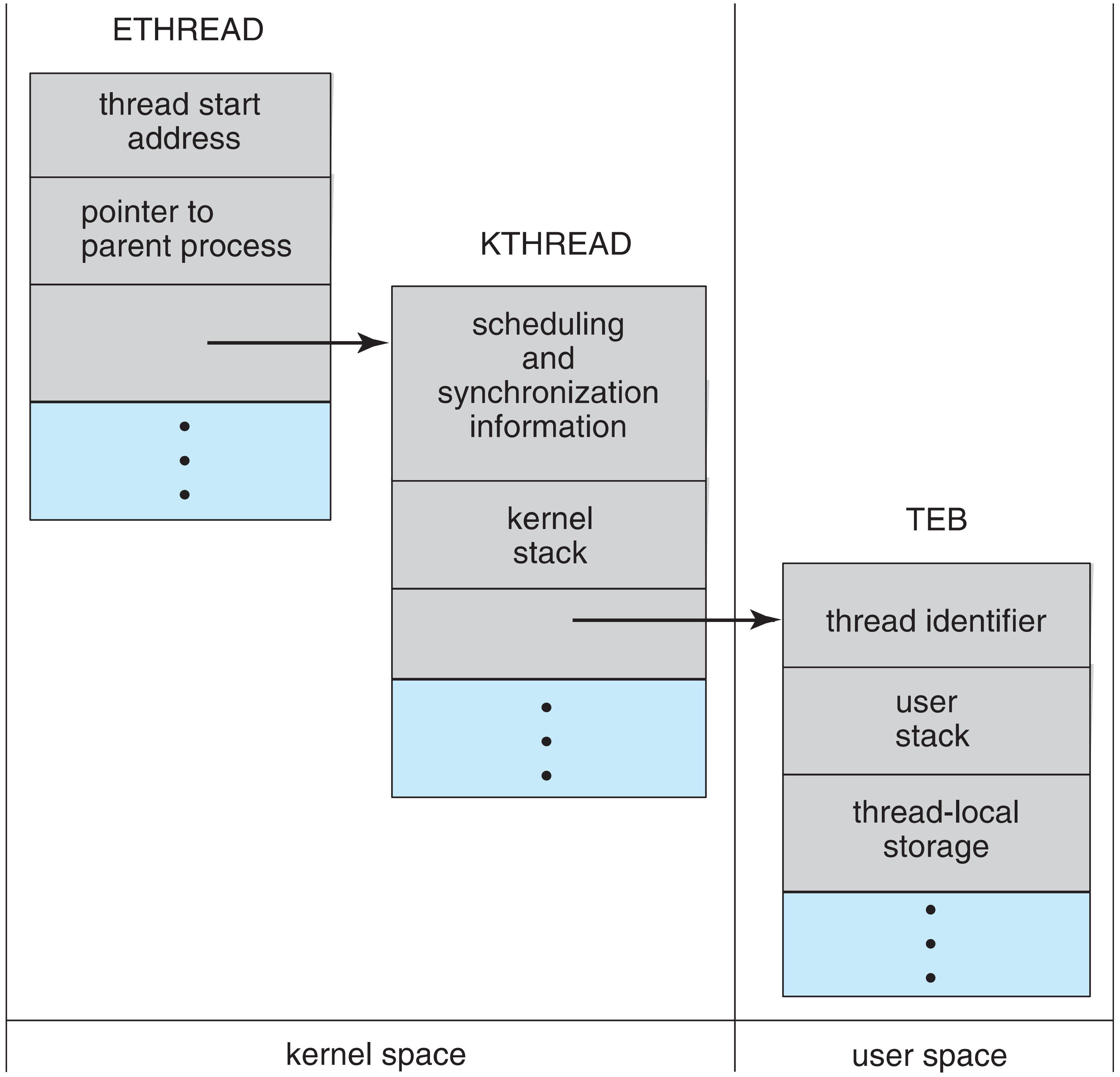
- Thread Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples
    - Windows XP Threads
    - Linux Thread

Windows Threads

- Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
  - Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks
  - Private data storage area used by run-time libraries and dynamic link libraries
- the **context** of the thread: the register set, stacks, and private storage area
- 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***
- The traditional fork( ) system call completely duplicates a process
- Thread creation is done through **`clone()`** system call, allowing for varying degrees of sharing between the parent and child tasks, controlled by flags shown below

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.

- Calling clone( )with no flags set is equivalent to fork( ).
- Calling clone( ) with all flags set is equivalent to creating a thread, as all of these data structures will be shared.

- **`struct task_struct`**
  - contains pointers to the data structures where these data are stored.
  - So varying level of sharing is possible.