Chapter 4: Threads

Zhi Wang
Florida State University

Contents

- Thread overview
- Multithreading models
- Thread libraries
- Threading issues
- Operating system examples
  - Windows XP threads
  - Linux threads

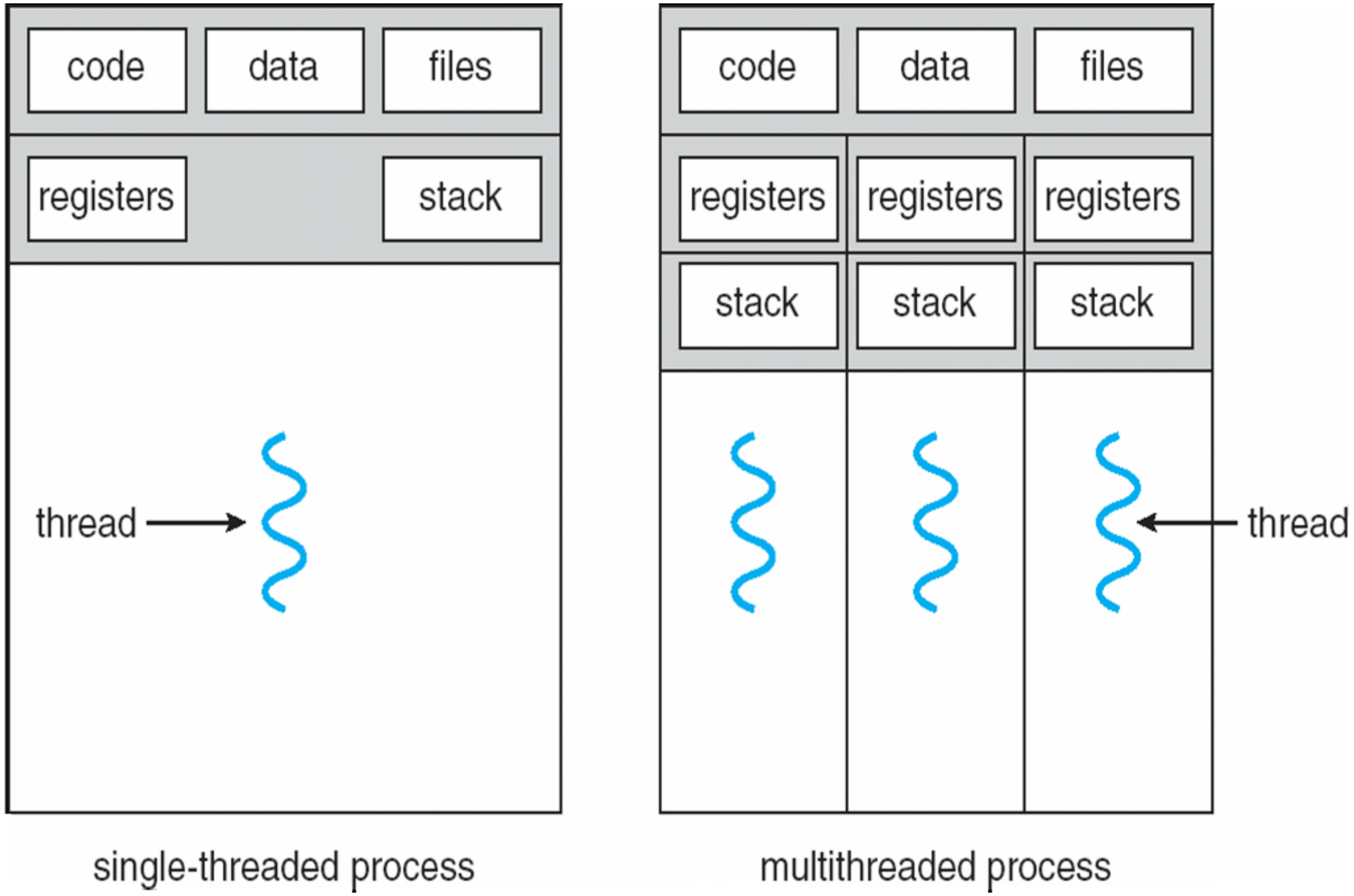
Motivation

- Why threads?
  - multiple tasks of an application can be implemented by threads
    - e.g., update display, fetch data, spell checking, answer a network request
  - process creation is heavy-weight while thread creation is light-weight
  - threads can simplify code, increase efficiency
- Kernels are generally multithreaded

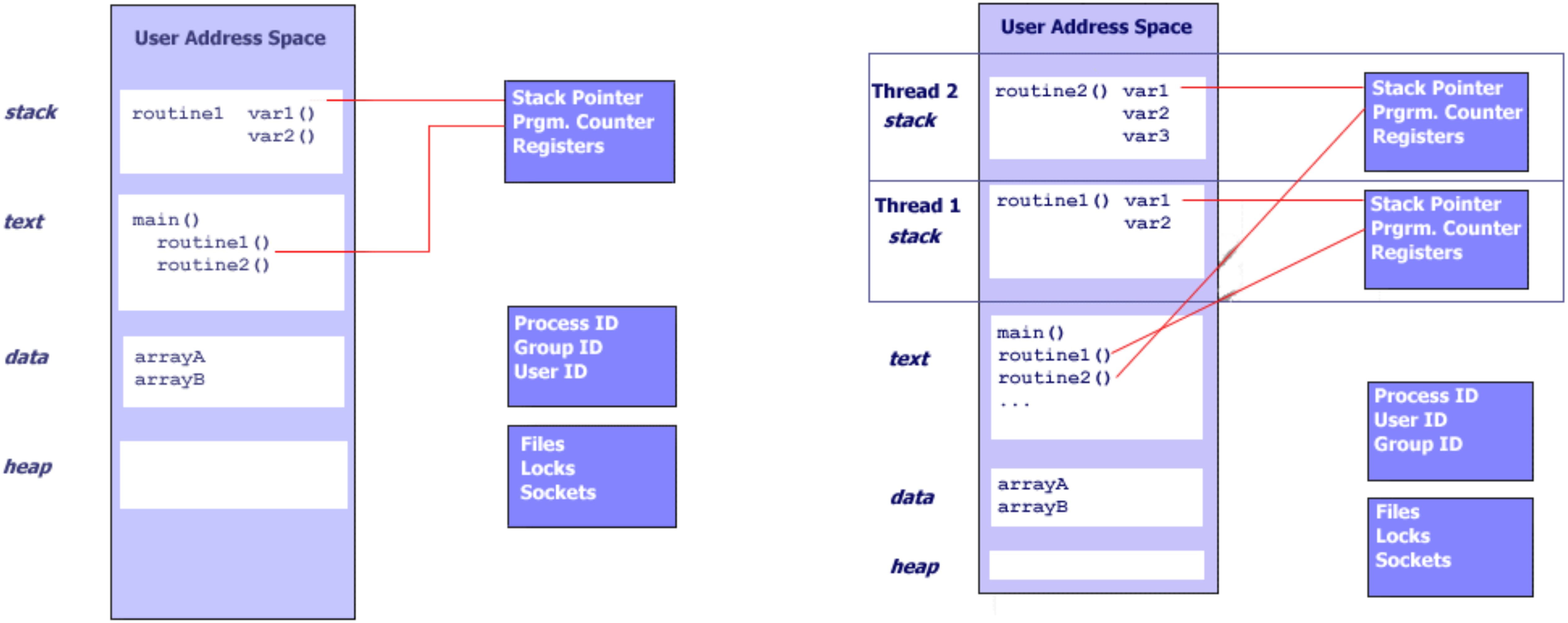
What is Thread

- A thread is an **independent stream of instructions that can be scheduled to run as such by the kernel**
- Process contains many states and resources
  - code, heap, data, file handlers (including socket), IPC
  - process ID, process group ID, user ID
  - stack, registers, and program counter (PC)
- **Threads exist within the process, and shares its resources**
  - each thread has its own essential resources (**per-thread resources**): **stack**, **registers**, **program counter**, **thread-specific data**…
  - access to shared resources need to be **synchronized**
- Threads are individually scheduled by the kernel
  - each thread has its own **independent flow of control**
  - each thread can be in any of the **scheduling states**

Single and Multithreaded Processes



UNIX Threads



User Address Space

stack

routine1 var1() var2()

text

main()
routine1()
routine2()

data

arrayA
arrayB

heap

Stack Pointer
Prgm. Counter
Registers

Process ID
Group ID
User ID

Files
Locks
Sockets

User Address Space

Thread 2 stack

routine2() var1 var2 var3

Thread 1 stack

routine1() var1 var2

text

main()
routine1()
routine2()
...

data

arrayA
arrayB

heap

Stack Pointer
Prgrm. Counter
Registers

Stack Pointer
Prgrm. Counter
Registers

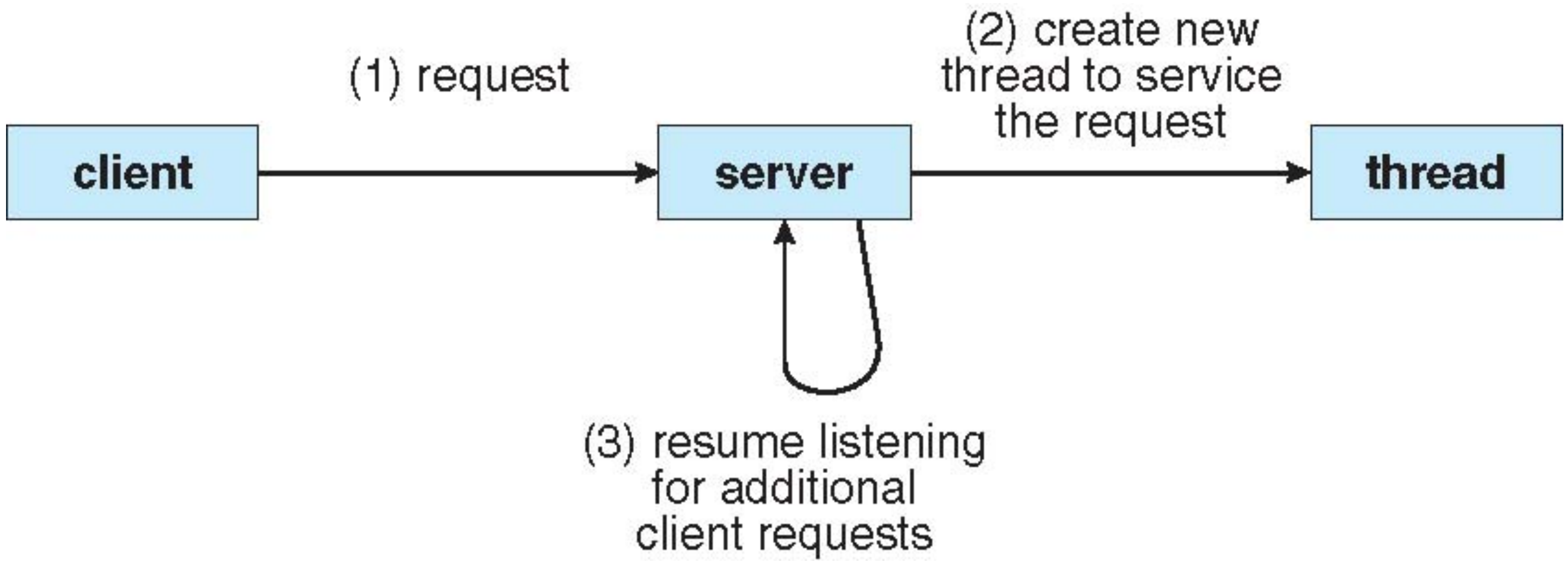
Process ID
User ID
Group ID

Files
Locks
Sockets

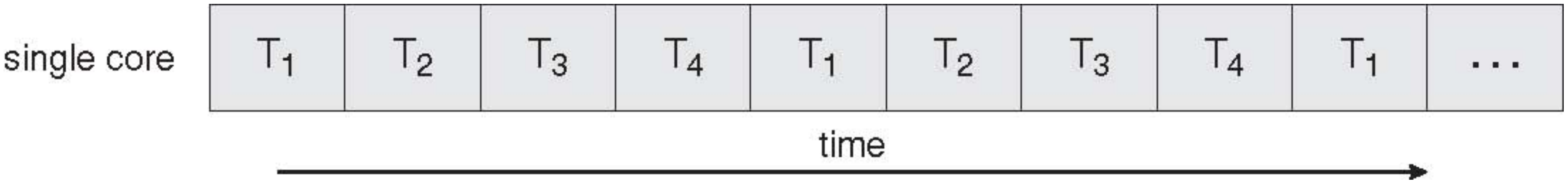
Thread Benefits

- **Responsiveness**
  - multithreading an interactive application allows a program to continue running even part of it is blocked or performing a lengthy operation
- **Resource sharing**
  - sharing resources may result in efficient communication and high degree of cooperation
- **Economy**
  - thread is more lightweight than processes
- **Scalability**
  - better utilization of multiprocessor architectures

Multithreaded Server Architecture



Concurrent Execution on a Single-core System

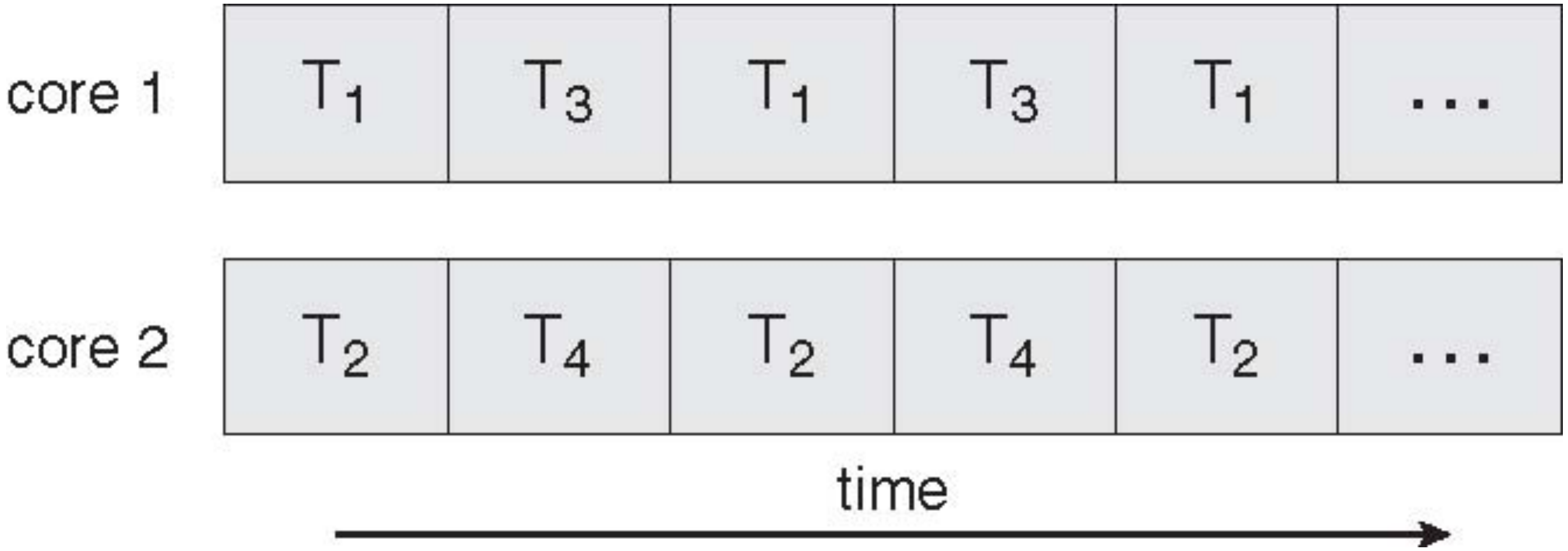


single core

T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4 T_1 . . .

time

Parallel Execution on a Multicore System



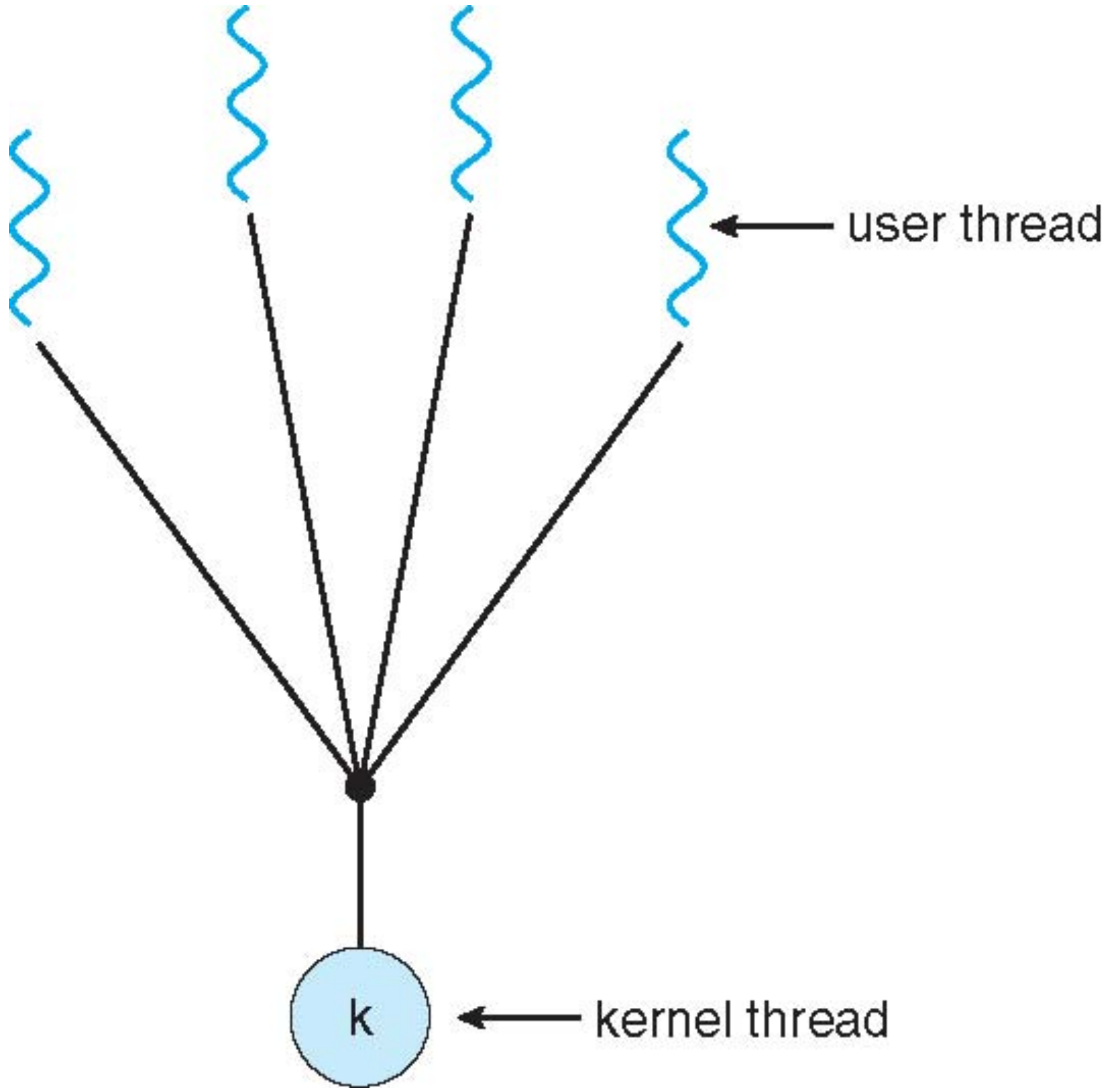
Implementing Threads

- Thread may be provided either at the user level, or by the kernel
  - **user threads** are supported above the kernel without kernel support
    - three thread libraries: **POSIX Pthreads**, **Win32 threads**, and **Java threads**
  - **kernel threads** are supported and managed directly by the kernel
    - all contemporary OS supports kernel threads
- Multithreading models: ways to map user threads to kernel threads
  - **many-to-one** model
  - **one-to-one** model
  - **many-to-many** model

Many-to-One

- Many user-level threads mapped to a single kernel thread
  - thread management is done by the thread library in **user space** (efficient)
  - entire process will block if a thread makes a blocking system call
    - convert blocking system call to non-blocking (e.g., select in Unix)?
  - multiple threads are unable to run in parallel on multi-processors
- Examples:
  - Solaris green threads
  - GNU portable threads

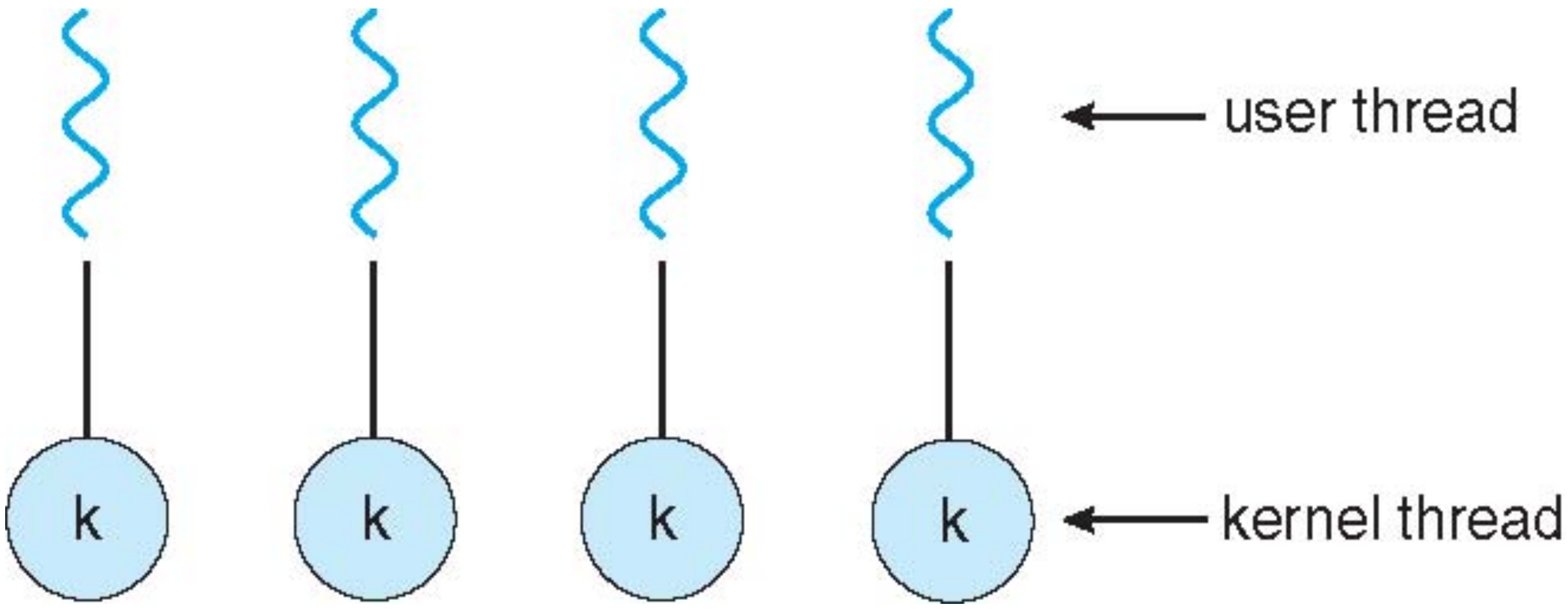
Many-to-One Model



One-to-One

- Each user-level thread maps to one kernel thread
  - it allows other threads to run when a thread blocks
  - multiple thread can run in parallel on multiprocessors
  - creating a user thread requires creating a corresponding kernel thread
    - it leads to overhead
  - most OSes implementing this model limit the number of threads
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

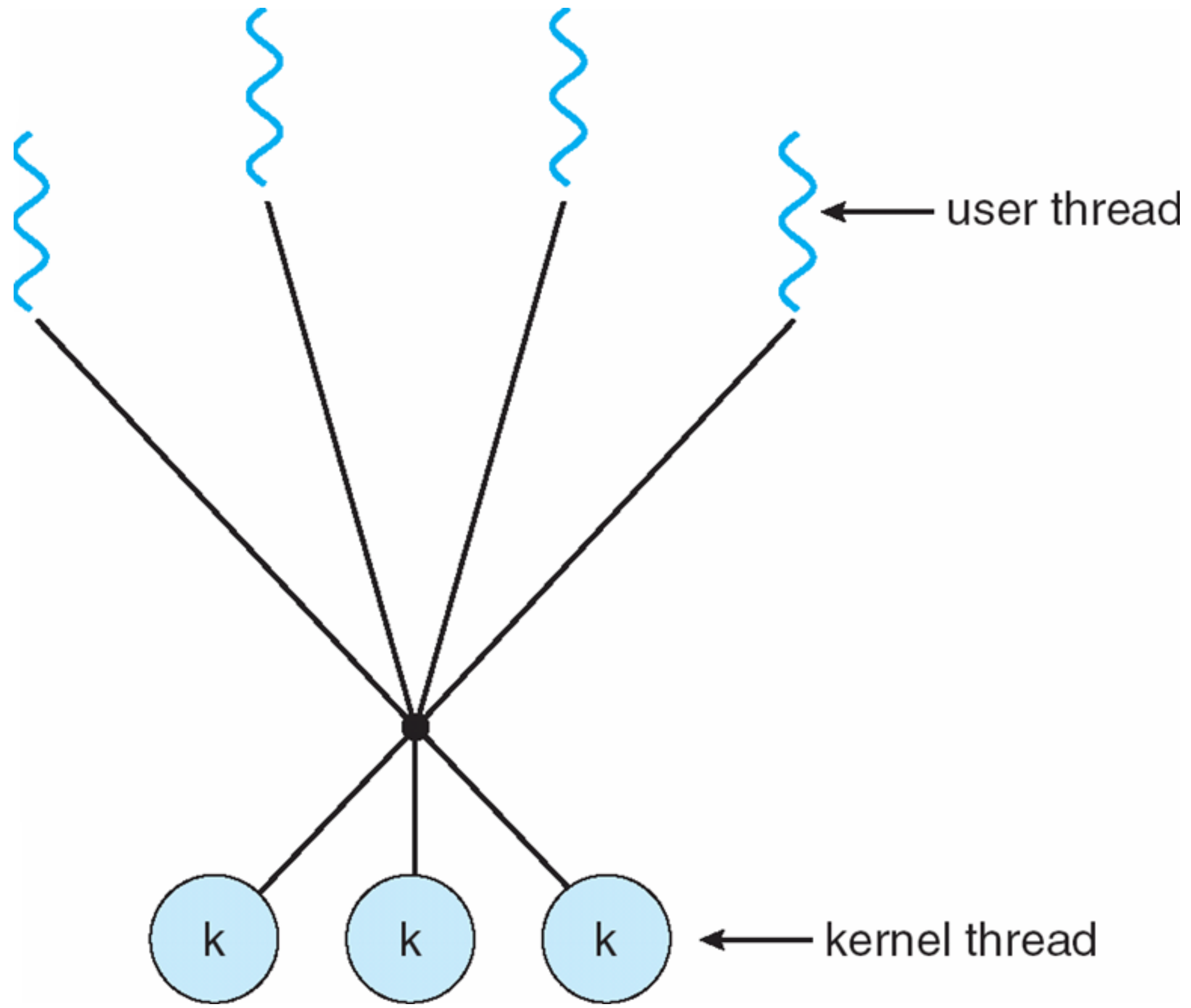
One-to-one Model



Many-to-Many Model

- Many user level threads are mapped to many kernel threads
  - it solves the shortcomings of 1:1 and m:1 model
  - developers can create as many user threads as necessary
  - corresponding kernel threads can run in parallel on a multiprocessor
- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the ThreadFiber package

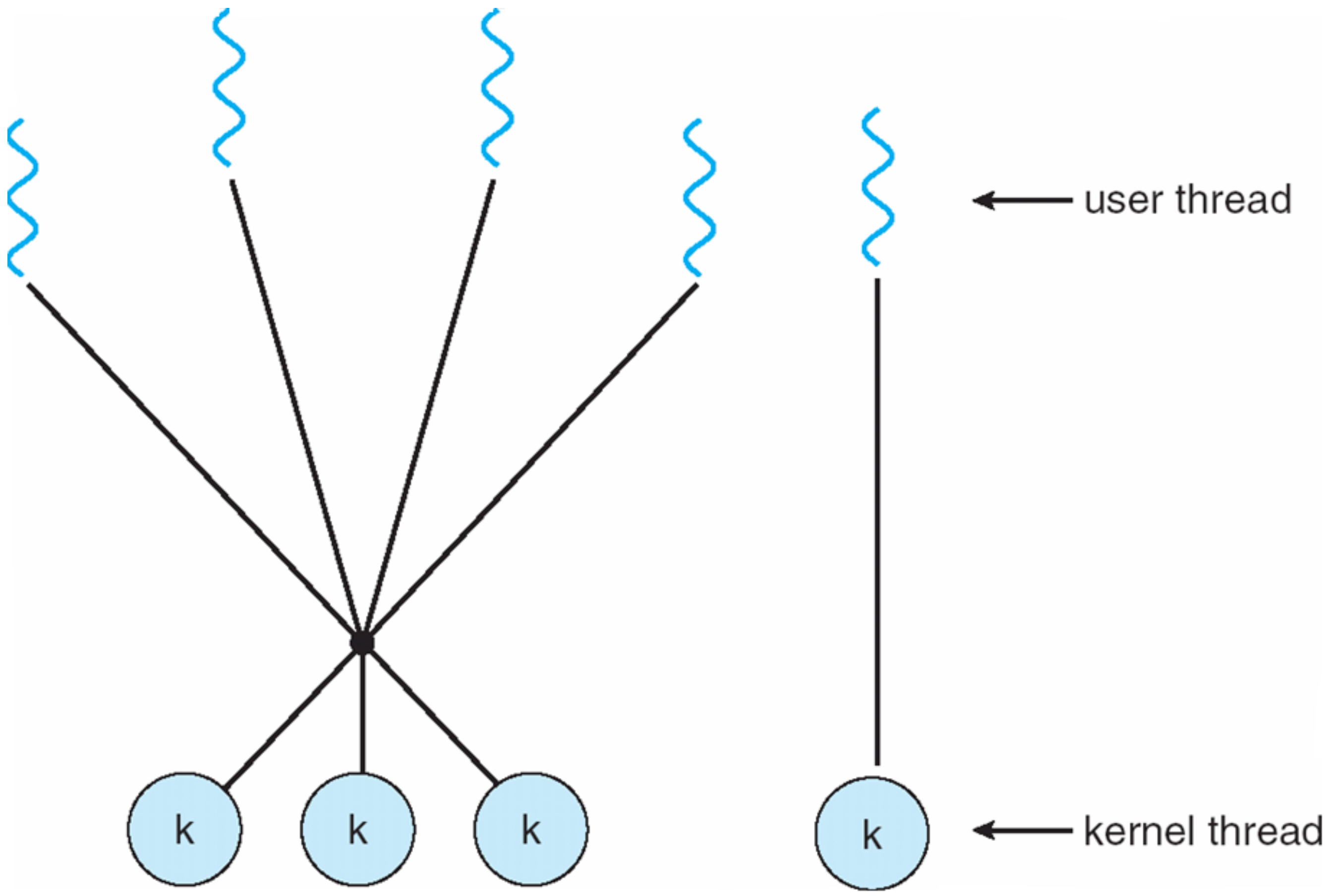
Many-to-Many Model



Two-level Model

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

Two-level Model



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - library entirely in user space with no kernel support
  - kernel-level library supported by the OS
- Three main thread libraries:
  - POSIX **Pthreads**
  - **Win32**
  - **Java**

Pthreads

- A POSIX standard API for thread **creation** and **synchronization**
  - common in UNIX operating systems (Solaris, Linux, Mac OS X)
  - Pthread is a specification for thread behavior
    - implementation is up to developer of the library
    - e.g., Pthreads may be provided either as user-level or kernel-level

Pthreads APIs

pthread_create	create a new thread
pthread_exit	terminate the calling thread
pthread_join	join with a terminated thread
pthread_kill	send a signal to a thread
pthread_yield	yield the processor
pthread_cancel	send a cancellation request to a thread
pthread_mutex_init	initialize a mutex
pthread_mutex_destroy	destroy a mutex
pthread_mutex_lock	lock a mutex
pthread_mutex_unlock	unlock a mutex
pthread_key_create	create a thread-specific data key
pthread_key_delete	delete a thread-specific data key
pthread_setspecific	set value for the thread-specific data key
pthread_getspecific	get value for the thread-specific data key

Pthreads Example

```
struct thread_info {        /* Used as argument to thread_start() */
    pthread_t thread_id;    /* ID returned by pthread_create() */
    int       thread_num;   /* Application-defined thread # */
    char     *argv_string;  /* From command-line argument */
};

static void *thread_start(void *arg)
{   struct thread_info *tinfo = (struct thread_info *) arg;
    char *uargv, *p;

    printf("Thread %d: top of stack near %p; argv_string=%s\n",
            tinfo->thread_num, &p, tinfo->argv_string);
    uargv = strdup(tinfo->argv_string);
    for (p = uargv; *p != '\0'; p++) {
          *p = toupper(*p);
    }
    return uargv;
}
```

Pthreads Example

```
int main(int argc, char *argv[])
{   …
    pthread_attr_init(&attr);
    pthread_attr_setstacksize(&attr, stack_size);

    /* Allocate memory for pthread_create() arguments */
    tinfo = calloc(num_threads, sizeof(struct thread_info));

    /* Create one thread for each command-line argument */
    for (tnum = 0; tnum < num_threads; tnum++) {
        tinfo[tnum].thread_num = tnum + 1;
        tinfo[tnum].argv_string = argv[optind + tnum];

        /* The pthread_create() call stores the thread ID into
           corresponding element of tinfo[] */
        pthread_create(&tinfo[tnum].thread_id, &attr,
                       &thread_start, &tinfo[tnum]);
    }

    pthread_attr_destroy(&attr);
```

Pthreads Example

```
   for (tnum = 0; tnum < num_threads; tnum++) {
        pthread_join(tinfo[tnum].thread_id, &res);
        printf("Joined with thread %d; returned value was %s\n",
                tinfo[tnum].thread_num, (char *) res);
        free(res);      /* Free memory allocated by thread */
    }

    free(tinfo);
    exit(EXIT_SUCCESS);
}
```

Win32 API Multithreaded C Program

```
typedef struct MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;

int _tmain()
{
    PMYDATA pDataArray[MAX_THREADS];
    DWORD   dwThreadIdArray[MAX_THREADS];
    HANDLE  hThreadArray[MAX_THREADS];

    // Create MAX_THREADS worker threads.
    for( int i=0; i<MAX_THREADS; i++ )
    {
        // Allocate memory for thread data.
        pDataArray[i] = (PMYDATA) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                sizeof(MYDATA));

        // Generate unique data for each thread to work with.
        …

        // Create the thread to begin execution on its own.
        hThreadArray[i] = CreateThread(
            NULL,                   // default security attributes
            0,                      // use default stack size
            MyThreadFunction,       // thread function name
            pDataArray[i],          // argument to thread function
            0,                      // use default creation flags
            &dwThreadIdArray[i]);   // returns the thread identifier
```

Win32 API Multithreaded C Program

```
       // Check the return value for success.
        // If CreateThread fails, terminate execution.
        // This will automatically clean up threads and memory.
        if (hThreadArray[i] == NULL)
        {
           ErrorHandler(TEXT("CreateThread"));
           ExitProcess(3);
        }
    } // End of main thread creation loop.

    // Wait until all threads have terminated.
    WaitForMultipleObjects(MAX_THREADS, hThreadArray, TRUE, INFINITE);

    // Close all thread handles and free memory allocations.
    for(int i=0; i<MAX_THREADS; i++)
    {
        CloseHandle(hThreadArray[i]);
        if(pDataArray[i] != NULL)
        {
            HeapFree(GetProcessHeap(), 0, pDataArray[i]);
            pDataArray[i] = NULL;    // Ensure address is not reused.
        }
    }

    return 0;
}
```

Java Threads

- Java threads are managed by the Java VM
  - it is implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - extending the **java.lang.Thread** class
  - then implement the **java.lang.Runnable** interface

Java Multithreaded Program

```java
public class SimpleThreads {
   static void threadMessage(String message) {
      String threadName = Thread.currentThread().getName();
      System.out.format("%s: %s%n", threadName, message);
   }

   private static class MessageLoop implements Runnable {
      public void run() {
         string importantInfo[] = { "Mares eat oats", "Does eat oats",
         "Little lambs eat ivy", "A kid will eat ivy too" };
         try {
            for (int i = 0; i < importantInfo.length; i++) {
               Thread.sleep(4000);
               threadMessage(importantInfo[i]); }
            } catch (InterruptedException e) {
               threadMessage("I wasn't done!");
            }
         }
      }
   }
```

Java Multithreaded Program

```
   public static void main(String args[]) throws InterruptedException {
      long patience = 1000 * 60 * 60;
      threadMessage("Starting MessageLoop thread");
      long startTime = System.currentTimeMillis();
      Thread t = new Thread(new MessageLoop());
      t.start();
      threadMessage("Waiting for MessageLoop thread to finish");
      while (t.isAlive()) {
         threadMessage("Still waiting...");
         t.join(1000);
         if (((System.currentTimeMillis() - startTime) > patience) &&
            t.isAlive()) {
            threadMessage("Tired of waiting!");
            t.interrupt();
            t.join();   // threads will exit soon
         }
      }
   }
}
```

Threading Issues

- Semantics of **fork** and **exec** system calls
- Thread cancellation of target thread
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Semantics of Fork and Exec

- **Fork** duplicates the whole single-threaded process
- Does fork duplicate only the **calling** thread or **all** threads for multi-threaded process?
  - some UNIX systems have two versions of fork, one for each semantic
  - if fork all, how to handle multiple threads running on CPUs?
- **Exec** typically replaces the **entire** process, multithreaded or not
  - use “fork the calling thread” if calling exec soon after fork
- Activity: review **man fork**

Thread Cancellation

- **Thread cancellation**: terminating a (target) thread before it has finished
  - does it cancel the target thread immediately or later?
- Two general approaches:
  - **asynchronous cancellation**: terminates the target thread ***immediately***
    - what if the target thread is in critical section requesting resources?
  - **deferred cancellation**: allows the target thread to periodically check if it should be cancelled
    - Pthreads: cancellation point

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred. It follows the same pattern:
  - a signal is generated by the occurrence of a particular event
  - a signal is delivered to a process
  - once delivered, the signal must be handled
- A signal can be **synchronous** (exceptions) or **asynchronous** (e.g., I/O)
  - **synchronous signals are delivered to the same thread causing the signal**
- Asynchronous signals can be delivered to:
  - the thread to which the signal applies
  - every thread in the process
  - certain threads in the process (signal masks)
  - a specific thread to receive all signals for the process

Thread Pools

- One thread per request model has **scalability** problems
  - overhead to create a thread before each request
  - it may exhaust the resource (Denial-of-Service attack)
- **Thread pool** solves the problem by:
  - create a number of threads in a pool, where they sit and wait for work
  - when received a request, the server wakens a thread and pass it the request
  - thread returns to the poll after completing the job
- Advantages:
  - servicing a request with an existing thread is usually faster than waiting to create a new thread
  - thread pool limits the number of threads that exist at any one point

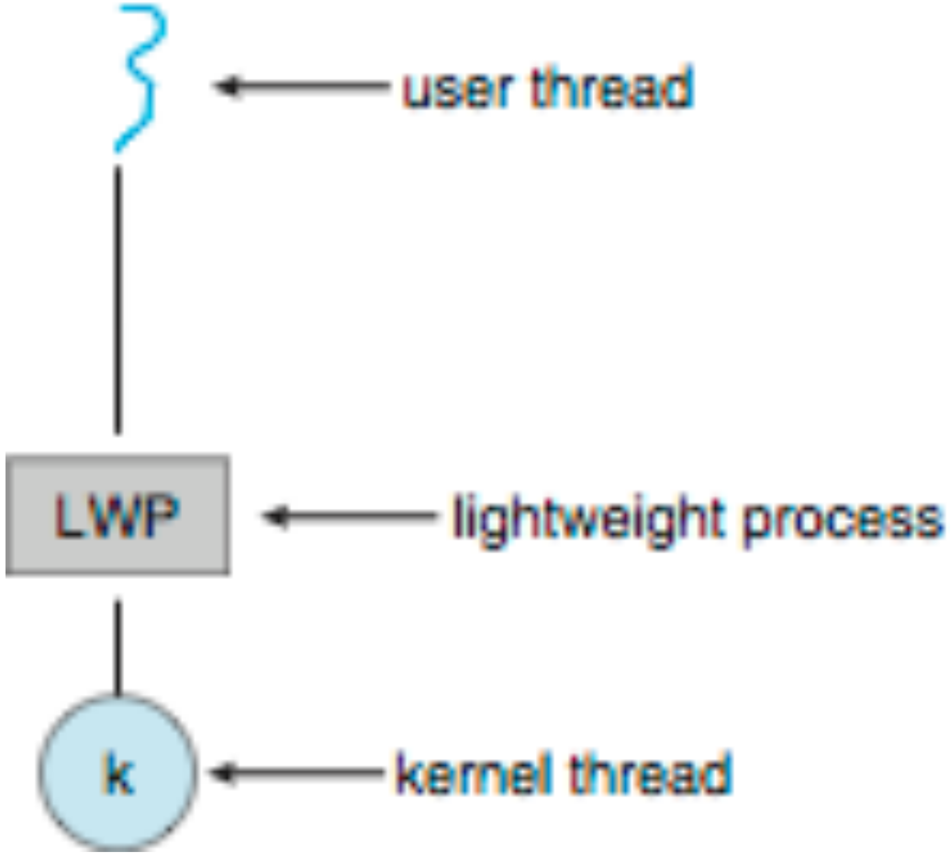
Thread Specific Data

- Data is shared in multithreaded programs
- Thread specific data allows each thread to have its own copy of data
- In kernel, there are usually CPU-specific data

Lightweight Process & Scheduler Activations

- Lightweight process (LWP) is an intermediate data structure between the user and kernel thread in **many-to-many** and **two level** models
  - to the user-thread library, it appears as **virtual processors** to schedule user threads on
  - each LWP is attached to a **kernel thread**
    - kernel thread blocks —> LWP blocks —> user threads block
  - kernel schedules the kernel thread, thread library schedules user threads
    - thread library may make sub-optimal scheduling decision
    - solution: **let the kernel notify the library of important scheduling events**
- **Scheduler activation** notifies the library via **upcalls**
  - upcall: the kernel call a upcall handler in the thread library (similar to signal)
  - e.g., when a thread is about to block, the library can pause the thread, and schedule another one onto the virtual processor

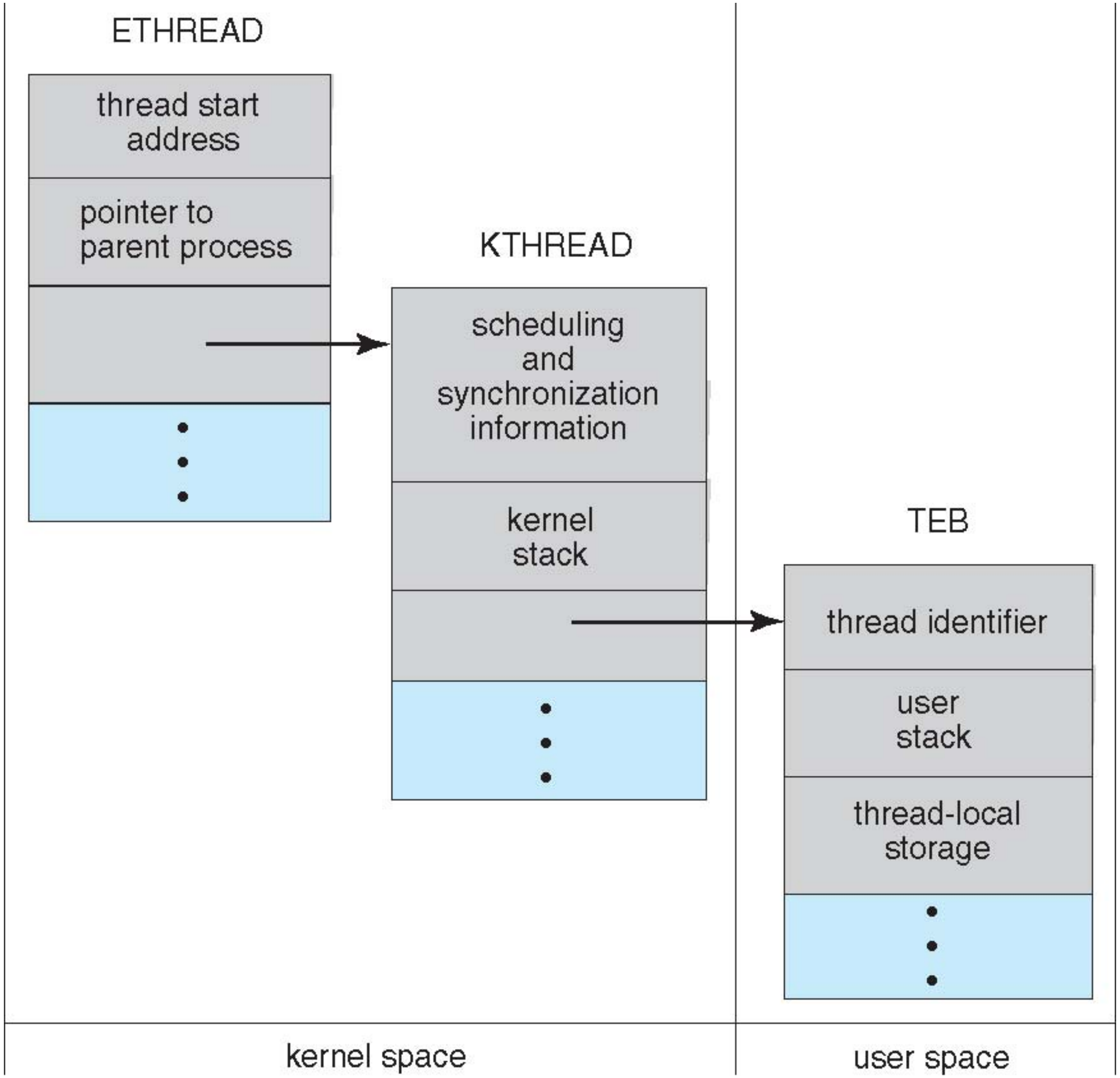
Lightweight Processes



Windows XP Threads

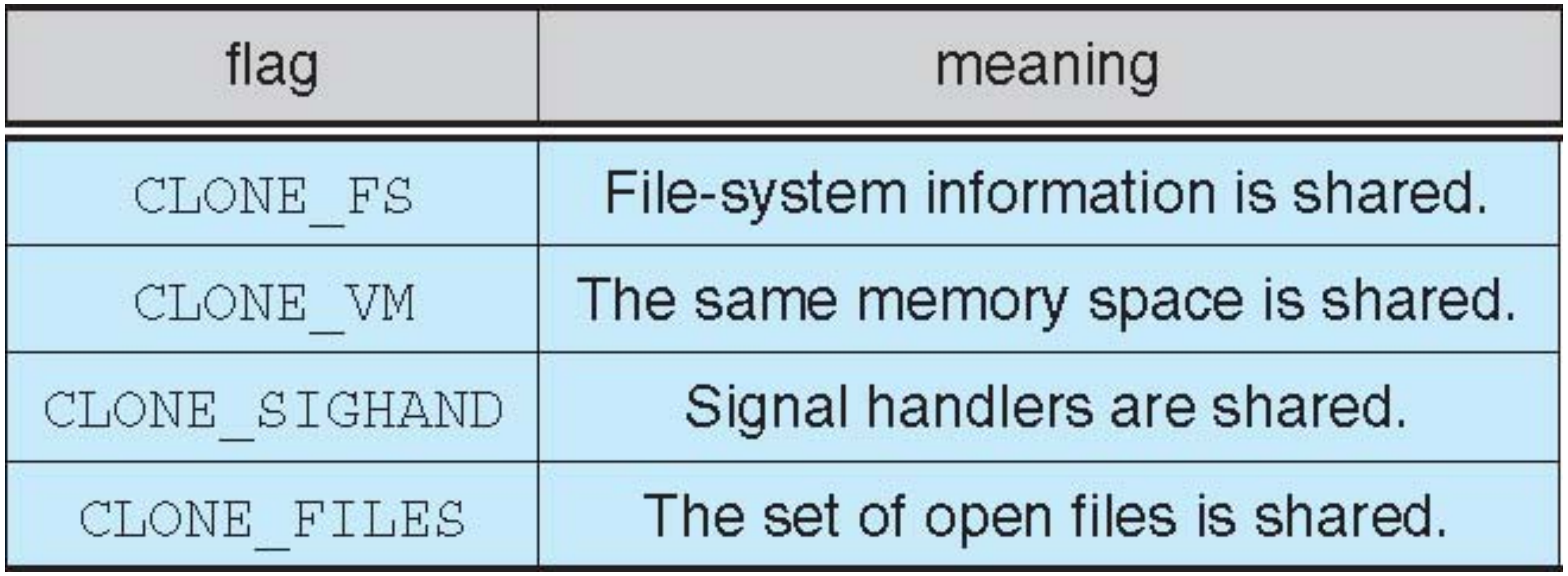
- Win XP implements the one-to-one mapping thread model
  - each thread contains
    - a thread id
    - a register set for the status of the processor
    - a separate user stack and a kernel stack
    - a private data storage area
- The primary data structures of a thread include:
  - **ETHREAD**: executive thread block (kernel space)
  - **KTHREAD**: kernel thread block (kernel space)
  - **TEB**: thread environment block (user space)

Windows XP Threads Data Structures



Linux Threads

- Linux has both fork and clone system call
- Clone accepts a set of **flags which determine sharing between the parent and children**
  - **FS/VM/SIGHAND/FILES** —> equivalent to **thread creation**
  - **no flag set no sharing** (copy) —> equivalent to **fork**
  - Linux doesn’t distinguish between process and thread, uses term **task** rather than thread



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.

End of Chapter 4