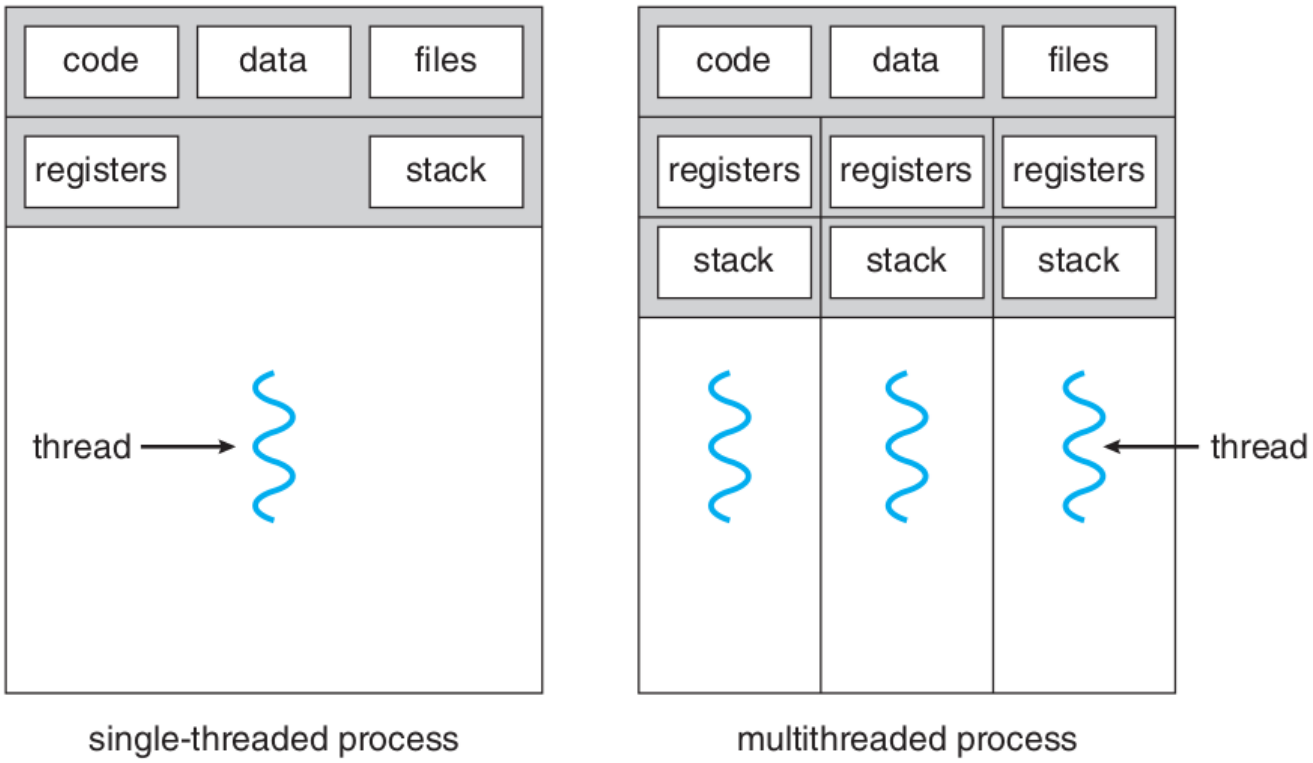
4 THREADS

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

An application typically is implemented as a separate process with several threads of control.



single-threaded process

multithreaded process

In certain situations, a single application may be required to perform several similar tasks. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced. When the server receives a request, it creates a separate process to service that request.

Threads also play a vital role in remote procedure call (RPC) systems. RPCs allow interprocess communication by providing a communication mechanism similar to ordinary function or procedure calls.

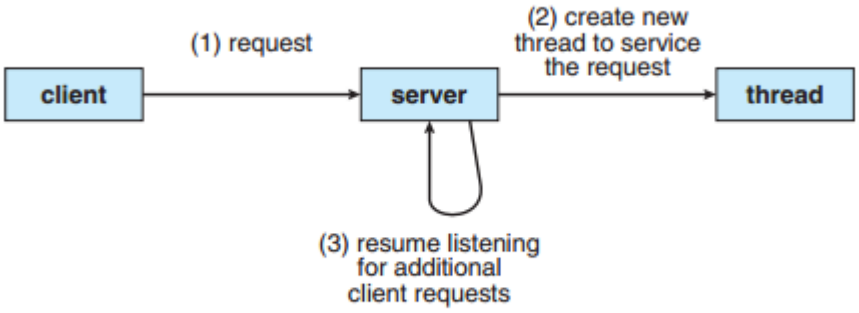


Figure 4.2 Multithreaded server architecture.

Most operating-system kernels are now multithreaded. Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling

Benefits

1. **Responsiveness:** allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
2. **Resource sharing:** Processes can only share resources through techniques such as shared memory and message passing. Threads share the memory and the resources of the process to which they belong by default. The benefit is that it allows an application to have several different threads of activity within the same address space.
3. **Economy:** Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads
4. **Scalability:** can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores

Multicore Programming

Whether the cores appear across CPU chips or within CPU chips, we call these systems **multicore** or **multiprocessor** systems. Multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency

On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

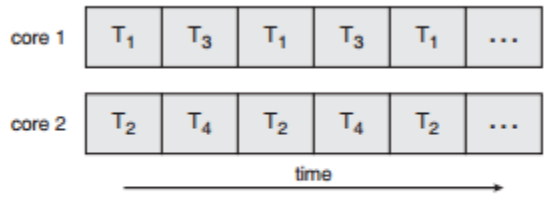


Figure 4.4 Parallel execution on a multicore system.

Notice the distinction between ***parallelism*** and ***concurrency*** in this discussion. A system is parallel if it can perform more than one task simultaneously. In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress. As systems have grown from tens of threads to thousands of threads, CPU designers have improved system performance by adding hardware to improve thread performance. Modern Intel CPUs frequently support two threads per core, while the Oracle T4 CPU supports eight threads per core.

AMDAHL'S LAW

Amdahl's Law is a formula that identifies potential performance gains from adding additional computing cores to an application that has both serial (nonparallel) and parallel components. If S is the portion of the application that must be performed serially on a system with N processing cores, the formula appears as follows:

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

Programming Challenges

Designers of operating systems must write scheduling algorithms that use multiple processing cores to allow the parallel execution shown in Figure 4.4. For application programmers, the challenge is to modify existing programs as well as design new programs that are multithreaded

1. **Identifying tasks:** Ideally, tasks are independent of one another and thus can run in parallel on individual cores.
2. **Balance:** Using a separate execution core to run that task may not be worth the cost
3. **Data splitting:** Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
4. **Data dependency:** When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency
5. **Testing and debugging:** Testing and debugging programs running in parallel on multiple cores, many different execution paths is inherently more difficult than testing and debugging single-threaded applications

Types of Parallelism

1. **Data parallelism:** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core
2. **Task parallelism:** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

Multithreading Models

User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems support kernel threads

Many-to-One Model

Thread management is done by the thread library in user space, so it is efficient, the entire process will block if a thread makes a blocking system call. **Green threads**—a thread library available for Solaris systems and adopted in early versions of Java. Very few systems continue to use the model because of its inability to take advantage of multiple processing cores.

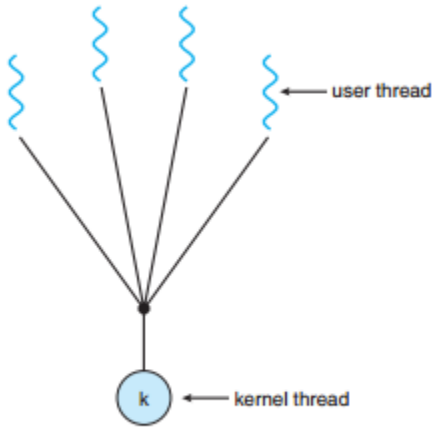


Figure 4.5 Many-to-one model.

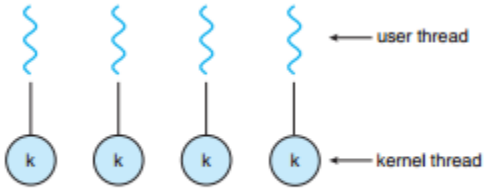


Figure 4.6 One-to-one model.

One-to-One Model

It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call and allows multiple threads to run in parallel on multiprocessors

Many-to-Many Model

The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor)

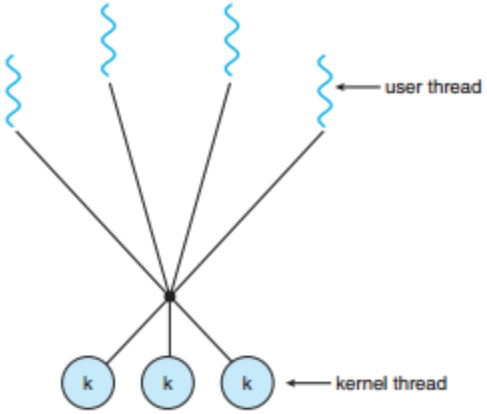


Figure 4.7 Many-to-many model.

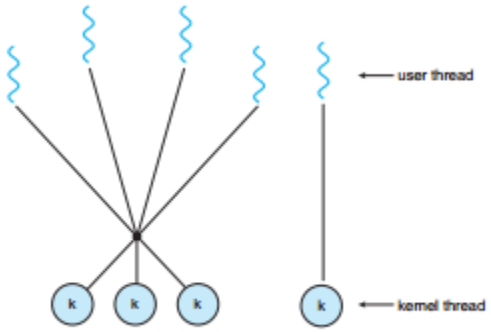


Figure 4.8 Two-level model.

When a thread performs a blocking system call, the kernel can schedule another thread for execution

Two-level model: One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread

Thread Libraries

They provides the programmer with an API for creating and managing threads.

The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.
The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java.

Pthreads

They refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a ***specification*** for thread behavior, not an ***implementation***. Operating-system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification; most are UNIX-type systems, including Linux, Mac OS X, and Solaris.

```c
#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;
   }

   /* 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);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

The C program shown in Figure 4.9 demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a nonnegative integer in a separate thread

```c
#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);
```

Figure 4.10 Pthread code for joining ten threads.

The pthread attr t attr declaration represents the attributes for the thread. We set the attributes in the function call pthread attr init(&attr). Because we did not explicitly set any attributes, we use the default attributes provided

Windows Threads

Notice that we must include the windows.h header file when using the Windows API. data shared by the separate threads—in this case, Sum—are declared globally (the DWORD data type is an unsigned 32-bit integer). We also define the Summation() function that is to be performed in a separate thread.

```
#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;
   }

   /* 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);
   }
}
```

Figure 4.11 Multithreaded C program using the Windows API.

These attributes include security information, the size of the stack, and a flag that can be set to indicate if the thread is to start in a suspended state.

In situations that require waiting for multiple threads to complete, the WaitForMultipleObjects() function is used. This function is passed four parameters:

1. The number of objects to wait for
2. A pointer to the array of objects
3. A flag indicating whether all objects have been signaled
4. A timeout duration (or INFINITE)

Java Threads

```
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);
  }
}

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>"); }
}
```

Figure 4.12 Java program for the summation of a non-negative integer.

the object instance of the Sum class.

All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a main() method runs as a single thread in the JVM. Figure 4.12 shows the Java version of a multithreaded program that determines the summation of a non-negative integer.

Calling the start() method for the new object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the run() method, making the thread eligible to be run by the JVM. (Note again that we never call the run() method directly. Rather, we call the start() method, and it calls the run() method on our behalf.)

If two or more threads are to share data in a Java program, the sharing occurs by passing references to the shared object to the appropriate threads. In the Java program shown in Figure 4.12, the main thread and the summation thread share

Implicit Threading

One way to address these difficulties and better support the design of multithreaded applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy, termed **implicit threading**, is a popular trend today

Thread Pools

Whereas creating a separate thread is certainly superior to creating a separate process, a mult-threaded server nonetheless has potential problems.

The first issue concerns the amount of time required to create the thread, together with the fact that the thread will be discarded once it has completed its work.

The second issue is if we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory. One solution to this problem is to use a **thread pool**

Thread pools offer these benefits:

1. Servicing a request with an existing thread is faster than waiting to créate a thread.

2. A thread pool limits the number of threads that exist at any one point. This is particularly important on systems that cannot support a large number of concurrent threads.

3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task. For example, the task could be scheduled to execute after a time delay or to execute periodically.

More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns. Such architectures provide the further benefit of having a smaller pool—thereby consuming less memory—when the load on the system is low

An example of invoking a function is the following:

```
QueueUserWorkItem(&PoolFunction, NULL, 0);
```

This causes a thread from the thread pool to invoke PoolFunction() on behalf of the programmer. In this instance, we pass no parameters to PoolFunction(). Because we specify 0 as a flag, we provide the thread pool with no special instructions for thread creation.

OpenMP

is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies **parallel regions** as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel. The following C program illustrates a compiler directive above the parallel region containing the `printf()` statement:

```
#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;
}
```

OpenMP provides several additional directives for running code regions in parallel, including parallelizing loops. In addition to providing directives for parallelization, OpenMP allows developers to choose among several levels of parallelism

Grand Central Dispatch

It is a combination of extensions to the C language, an API, and a run-time library that allows application developers to identify sections of code to run in parallel. Like OpenMP, GCD manages most of the details of threading, build for Apple's Mac OS X and iOS operative systems

When it removes a block from a queue, it assigns the block to an available thread from the thread pool it manages. GCD identifies two types of dispatch queues: ***serial*** and ***concurrent***

Blocks placed on a concurrent queue are also removed in FIFO order, but several blocks may be removed at a time, thus allowing multiple blocks to execute in parallel

Other Approaches

Other commercial approaches include parallel and concurrent libraries, such as Intel's Threading Building Blocks (TBB) and several products from Microsoft. The Java language and API have seen significant movement toward supporting concurrent programming as well.

Threading Issues

The fork() and exec() System Calls

If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call

The exec() system call typically works in the same way as described in Chapter 3. That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.

Signal Handling

All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

Typically, an asynchronous signal is sent to another process.
A signal may be ***handled*** by one of two possible handlers:

1. **A default signal handler:** that the kernel runs when handling that signal.
2. **A user-defined signal handler:** that is called to handle the signal. Signals are handled in different ways.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads.

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

*The method for delivering a signal depends on the type of signal generated

Thread Cancellation

It involves terminating a thread before it has completed.

A thread that is to be canceled is often referred to as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

1. **Asynchronous cancellation**. One thread immediately terminates the target thread. . Therefore, canceling a thread asynchronously may not free a necessary system-wide resource
2. **Deferred cancellation**. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The following code illustrates creating—and then canceling— a thread:

```
pthread_t tid;

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

. . .

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

Invoking pthread cancel()indicates only a request to cancel the target thread, however; actual cancellation depends on how the target thread is set up to handle the request

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

Pthreads allows threads to disable or enable cancellation. The cancellation requests remain pending, so the thread can later enable cancellation and respond to the request.

Thread-Local Storage

This data sharing provides one of the benefits of multithreaded programming. In some circumstances, each thread might need its own copy of certain data. We will call such data **thread-local storage** (or TLS.) Local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations

Scheduler Activations

It concerns communication between the kernel and the thread library. Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure—typically known as a **lightweight process**, or LWP—is shown in Figure 4.13

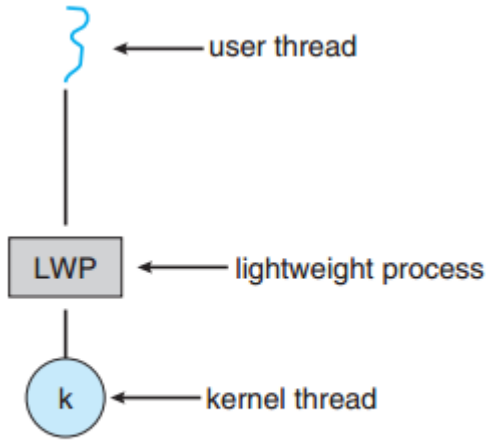


Figure 4.13 Lightweight process (LWP).

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**.

Operating-System Examples

Windows Threads

A Windows application runs as a separate process, and each process may contain one or more threads. Windows uses the one-to-one mapping

The general components of a thread include:

- A thread ID uniquely identifying the thread
- A register set representing the status of the processor
- A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
- A private storage area used by various 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
- KTHREAD—kernel thread block
- TEB—thread environment block

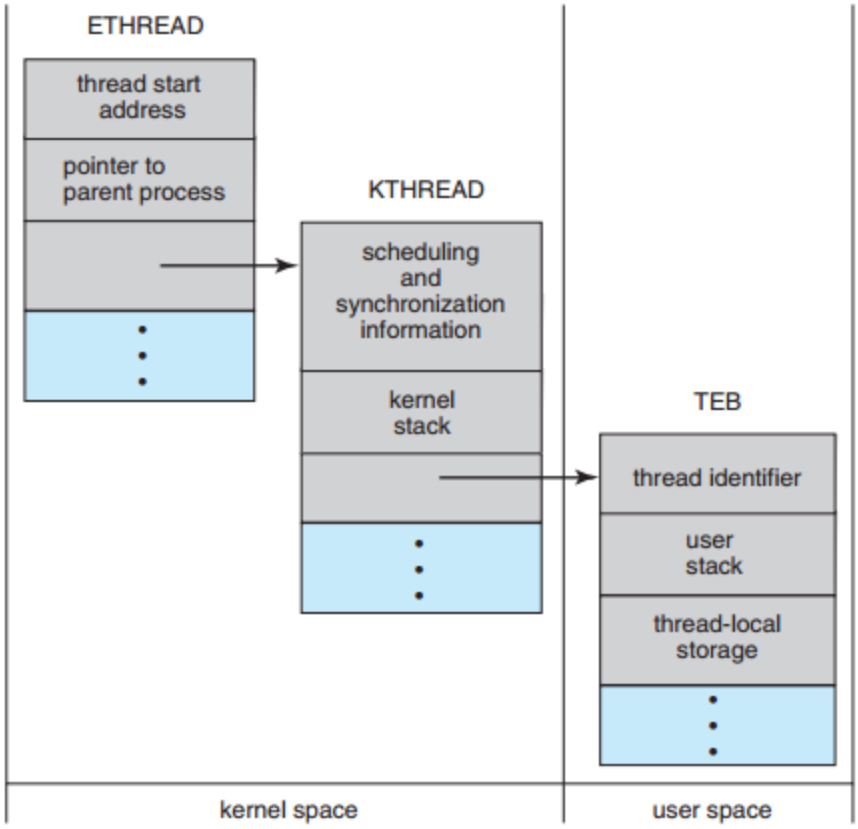


Figure 4.14 Data structures of a Windows thread.

Linux Threads

Linux does not distinguish between processes and threads. In fact, Linux uses the term ***task*** —rather than ***process*** or ***thread***— when referring to a flow of control within a program. The varying level of sharing is possible because of the way a task is
represented in the Linux kernel

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.

Figure 4.15 Some of the flags passed when `clone()` is invoked.