

Threads

Chapter 4

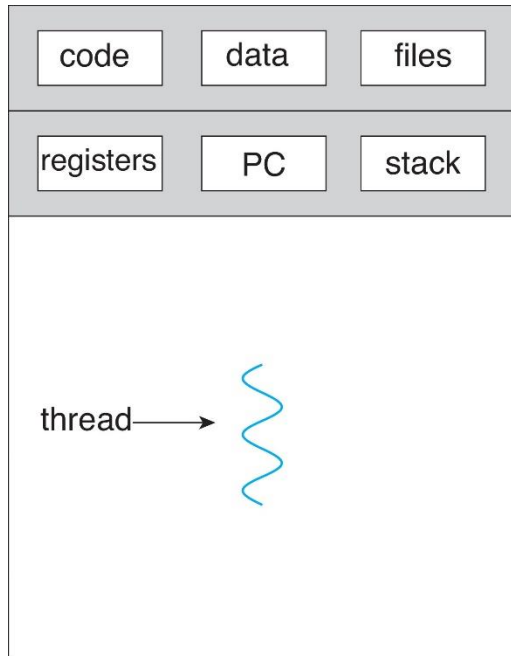
Review

- Process is a program in execution that can be in a number of states
 - New, running, waiting, ready, terminated
- Process creation
 - `fork()` and `exec()` system calls
- Inter-process communications
 - Shared memory, and message passing
- Client-server communication
 - Socket, RPC, ...

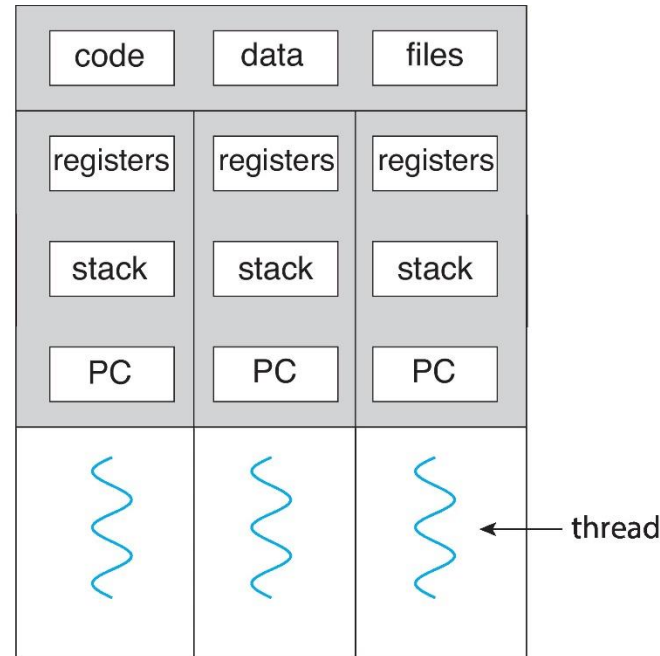
Threads

- Traditional processes have a single thread of control. Most modern applications are multithreaded
- Multi-threaded processes have multiple threads of control
 - The threads share the address space and resources of the process that owns them.
- A **thread** (a lightweight process) is a basic unit of CPU utilization.
 - A thread has a single sequential flow of control.
 - A thread is comprised of: A thread ID, a program counter, a register set and a stack.
- A **process** becomes the execution environment in which threads run.
 - (Recall previous definition of process: program in execution).
- The **process** has the code section, data section, OS resources (e.g. open files and signals).

Single and Multithreaded Processes



single-threaded process



multithreaded process

Threads encapsulate concurrency: “Active” component

Address spaces encapsulate protection: “Passive” part Keeps buggy program from trashing the system

Processes vs. Threads

Which of the following belong to the process and which to the thread?

Program code:	Process
local or temporary data:	Thread
global data:	Process
allocated resources:	Process
execution stack:	Thread
memory management info:	Process
Program counter:	Thread
Parent identification:	Process
Thread state:	Thread
Registers:	Thread

If the main thread of a process creates a new thread, which of the following are shared between the main thread and the new thread? Select one or more:

- a. Heap Section
- b. Open file descriptors
- c. code section
- d. Program counter
- e. Stack section
- f. Global data
- g. Registers

Control Blocks

- The thread control block (*TCB*) contains:
 - Thread state, Program Counter, Registers
- *PCB'* = everything else (e.g. process id, open files, etc.)
- The process control block (*PCB*) = *PCB'* \cup *TCB*

Why use threads?

- Because threads have minimal internal state, it takes less time to create a thread than a process (10x speedup in UNIX).
- It takes less time to terminate a thread.
- It takes less time to switch to a different thread.
- A multi-threaded process is much cheaper than multiple (redundant) processes.
- Threads share an address space and thus make it easy to share data

Benefits of Multi-threads

- **Responsiveness:**
 - Threads allow a program to continue running even if part is blocked.
 - For example, a web browser can allow user input while loading an image.
- **Resource Sharing:**
 - Threads share memory and resources of the process to which they belong.
- **Economy:**
 - Allocating memory and resources to a process is costly.
 - Threads are faster to create and faster to switch between.
- **Scalability:**
 - Multi-thread process can take advantage of multiprocessor architectures

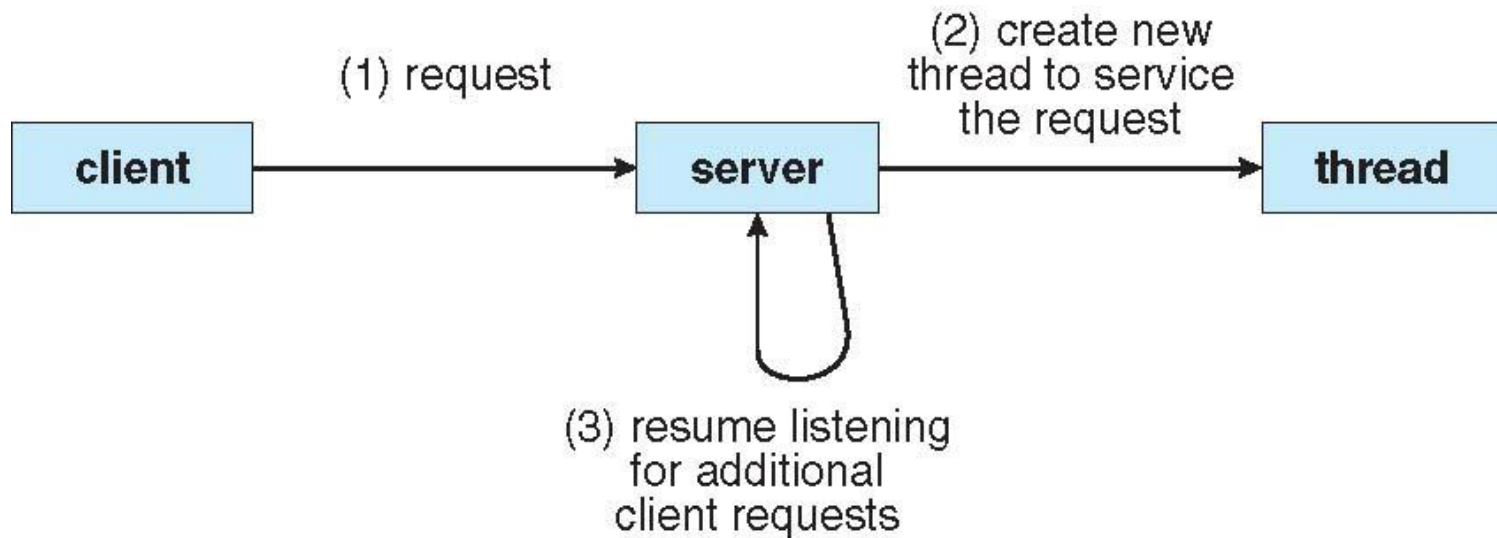
Examples of Using Threads

- Threads are useful for any application with multiple tasks that can be run with separate threads of control.
- A Word processor may have separate threads for:
 - User input
 - Spell and grammar check
 - displaying graphics
 - document layout
- A web server may spawn a thread for each client
 - Can serve clients concurrently with multiple threads.
 - It takes less overhead to use multiple threads than to use multiple processes.

Examples of multithreaded programs

- Most modern OS kernels
 - Internally concurrent because have to deal with concurrent requests by multiple users
 - But no protection needed within kernel
- Database Servers
 - Access to shared data by many concurrent users
 - Also background utility processing must be done
- Parallel Programming (More than one physical CPU)
 - Split program into multiple threads for parallelism. This is called Multiprocessing

Multithreaded Server Architecture

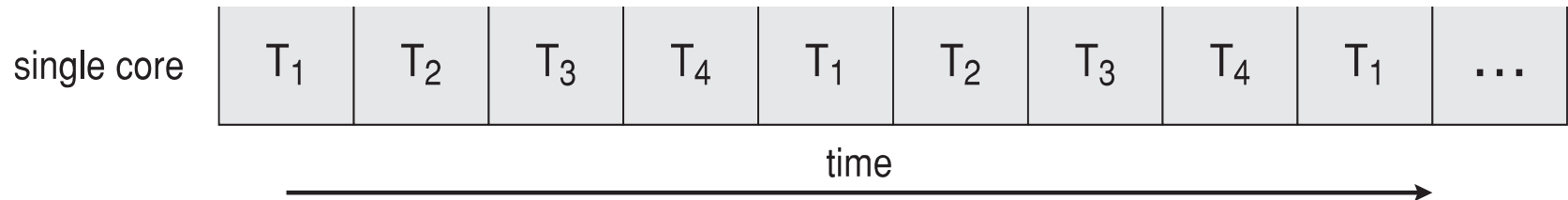


Parallel Programming

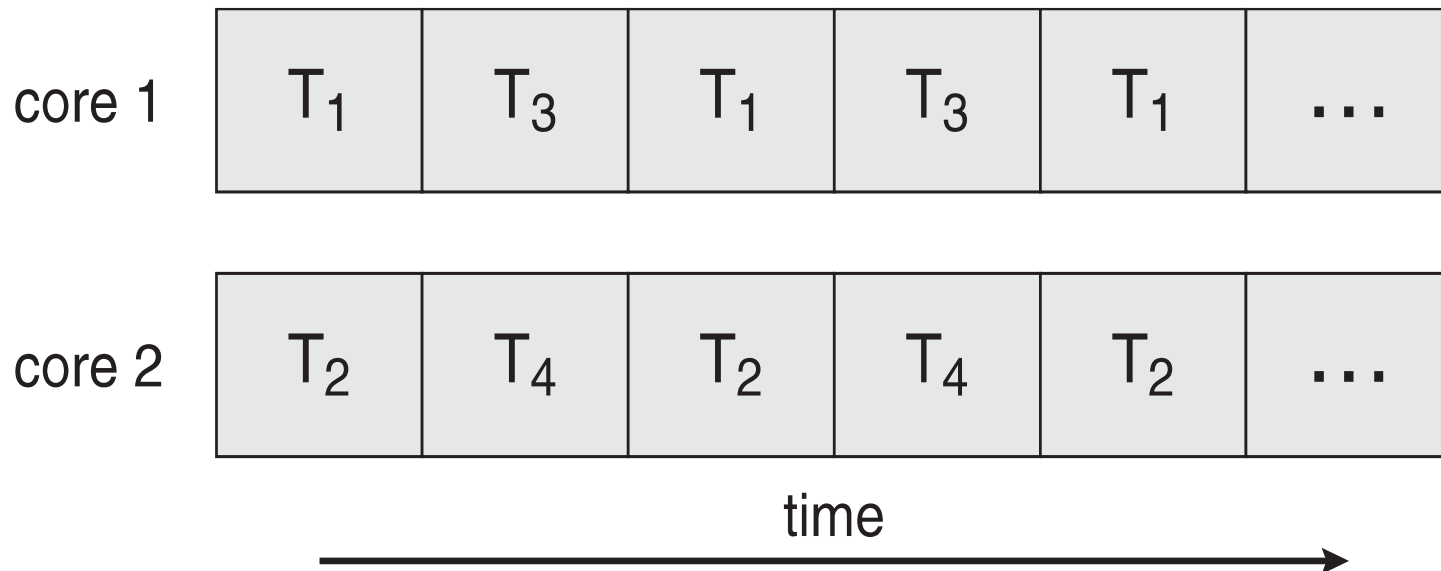
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

□ Concurrent execution on single-core system:



□ Parallelism on a multi-core system:



Thread Libraries

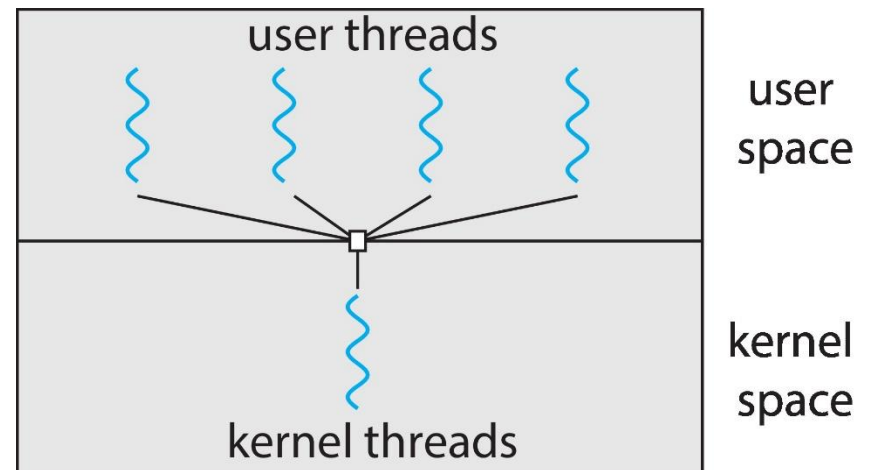
- **Thread library** provides programmer with API for creating and managing threads. Three primary thread libraries:
 - POSIX Pthreads
 - Win32 threads
 - Java threads
- Pthreads is a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
 - API specifies behavior of the thread library, implementation is up to development of the library
 - May be provided either as user-level or kernel-level
 - Common in UNIX operating systems (Solaris, Linux, Mac OS X)

User Threads and Kernel Threads

- Two primary ways of implementing Threads libraries
 - Library entirely in user space
 - Kernel-level library supported by the OS
- **User threads** - management done by user-level threads library
- **Kernel threads** - Supported by the Kernel
 - Threads managed by the OS Kernel.
 - Kernel does creation, scheduling and management of threads.
 - Examples – virtually all general purpose operating systems, including: Windows, Solaris, Linux, Mac OS X
 - User-level threads are mapped to kernel thread.

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
- Examples:
 - **Solaris Green Threads**
 - **GNU Portable Threads**



What happens when a process or thread is blocked in a many-to-one case?

- In a many-to-one thread mapping system, suppose process B has two user threads, of which thread 2 is running. If user thread 2 makes a blocking I/O system call, which of the following is true?

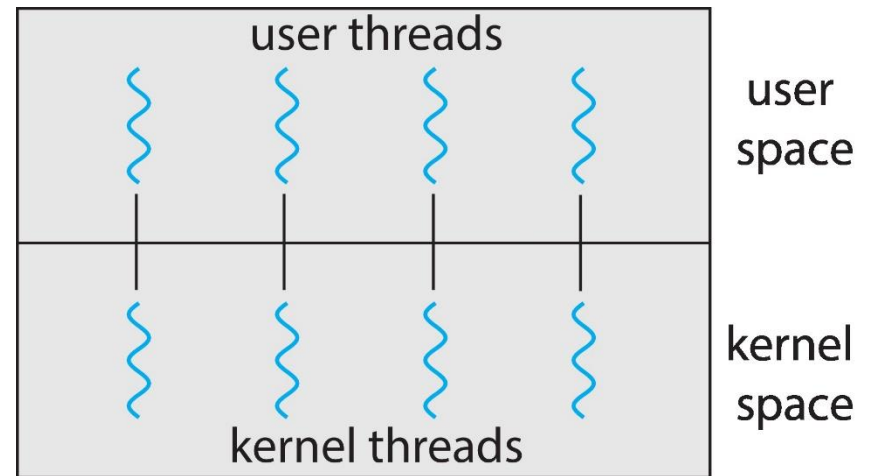
A	Thread 2 is in waiting state, but thread 1 starts running on the CPU
B	Thread 2 is not running, but the process is in ready state
C	Thread 2 is not running, but the process is in running state
<input checked="" type="radio"/> D	The entire process is in waiting state and blocked.
E	None of the above

Properties of User Level Threads and Many to One Mapping

- Pros:
 - Fast switch between threads (kernel is not needed).
 - Can have multi-threaded process on a system that does not understand threads.
 - Cheap to create and destroy threads.
 - User has complete control over threads (e.g. can assign priorities, etc.)
- Cons:
 - System call from one thread blocks all threads in the same process (many-to-one mapping).
 - Process cannot use multiple processors.
 - Extra work for user.

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
 - Linux
 - Solaris 9 and later

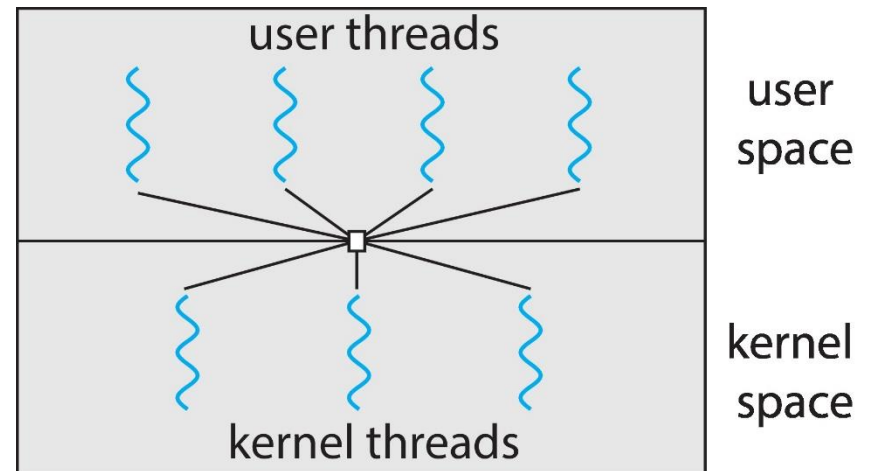


Pros and Cons of One-to-one Mapping

- Pros:
 - System call from a thread does not block other threads in the same process.
 - One process can use multiple processors.
 - Create/destroy/switch of threads is less expensive than for processes.
- Cons:
 - Create/destroy/switch of Kernel Level threads is more expensive than for user level threads.
 - CPU scheduling algorithms are unfair: Each thread is given the same time slice. Tasks with more threads are given more CPU time than those with fewer threads.

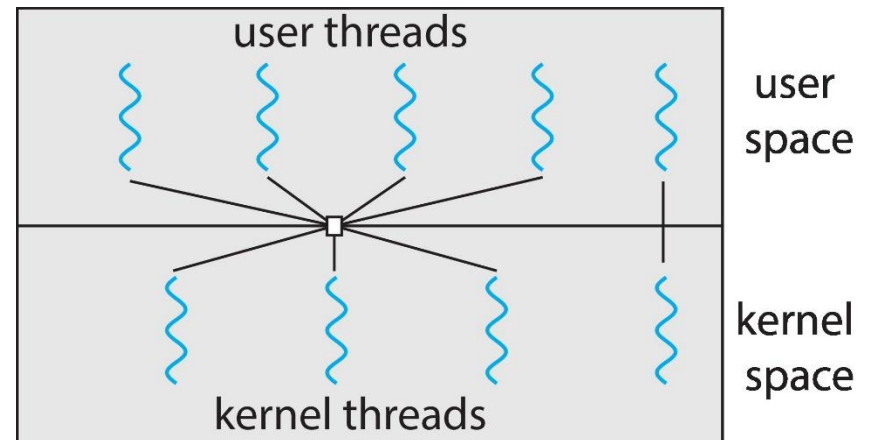
Many-to-Many Model

- 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
- Otherwise not very common



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



Summation Problem

- Compute summation $0+1+\dots+N$
- N is a non-negative number provided as a command-line argument
- We want to create a new thread to compute the summation and print out the result in the main thread.

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);
}
```

Figure 4.9 Multithreaded C program using the Pthreads API.

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

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

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

Figure 4.10 Pthread code for joining ten threads.

```
int max;  
int counter = 0; // shared global variable
```

```
void *mythread(void *arg) {  
    char *letter = arg;  
    int i; // stack (private per thread)  
    printf("%s: begin [addr of i: %p]\n", letter, &i);  
    for (i = 0; i < max; i++) {  
        counter = counter + 1; // shared: only one  
    }  
    printf("%s: done\n", letter);  
    return NULL;  
}
```

```
int main(int argc, char *argv[]) {  
    max = atoi(argv[1]);  
  
    pthread_t p1, p2;  
    printf("main: begin [counter = %d] [%x]\n", counter,  
        (unsigned int) &counter);  
    Pthread_create(&p1, NULL, mythread, "A");  
    Pthread_create(&p2, NULL, mythread, "B");  
    // join waits for the threads to finish  
    Pthread_join(p1, NULL);  
    Pthread_join(p2, NULL);  
    printf("main: done\n [counter: %d]\n [should: %d]\n",  
        counter, max*2);  
    return 0;  
}
```

What is the value of counter if
max = 1,000,000?

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
- Example methods explored
 - Thread Pools
 - OpenMP
- Other methods include Intel Threading Building Blocks (TBB), **`java.util.concurrent`** package

Thread Pools

- The **problem** with allowing a process (e.g. a multi-threading web server) to create as many threads as it wants:
 - It takes time to create a thread prior to handling a service request.
 - Unlimited number of threads could exhaust system resources.
- **Solution:** Thread pools.
- A **thread pool** contains a limited number of threads created at process startup.
 - When the program needs a thread, it takes one from the pool.
 - When the thread is done with its service, it is returned to the pool.
 - If no thread is available, the program must wait for one to be returned to the pool.

Thread Pools

- 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.  
    */  
}
```

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

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;  
}
```


Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread specific data

Semantics of fork() and exec()

- If one thread in a multi-threaded program calls fork(), does the new process duplicate all threads or only the thread that made the call?
- It depends on the version of UNIX.
- Some UNIX versions have 2 versions of fork()—
 - one that duplicates all threads and
 - one that duplicates only one thread.
- What happens if exec() is called from a thread?
- The same as before--the entire process is replaced.
- If exec() is called immediately after fork() then it is not necessary to duplicate all threads.

Thread Cancellation

- **Thread cancellation** is terminating a thread before it is completed.
- The thread to be cancelled is called the **target thread**.
- **Asynchronous cancellation:** One thread immediately terminates the target thread. This can cause problems:
 - Cancellation may occur while the thread is in the middle of updating shared data.
 - The OS may not reclaim all resources from the cancelled thread.
- **Deferred cancellation:** Target thread can periodically check to determine whether it should terminate
 - This allows the target thread an opportunity to terminate itself in an orderly fashion.
 - Threads are only terminated at **cancellation points**.
 - `pthread_cancel` and `pthread_testcancel`

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

```
pthread_t tid;

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

. . .

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

- 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

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

    . . .

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

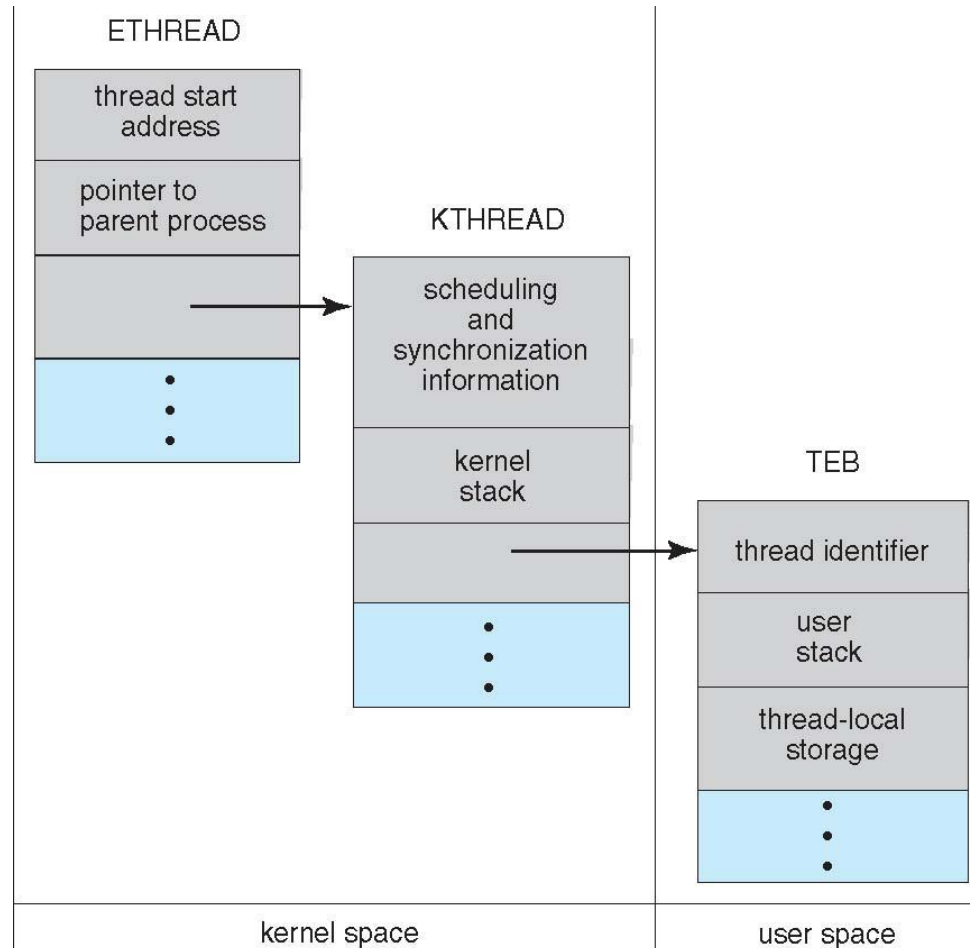
Thread-Local Storage

- **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
 - Use **thread_local** keyword in C++
 - `thread_local unsigned int rage = 1;`

Windows Threads

- 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 threads
- 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



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)