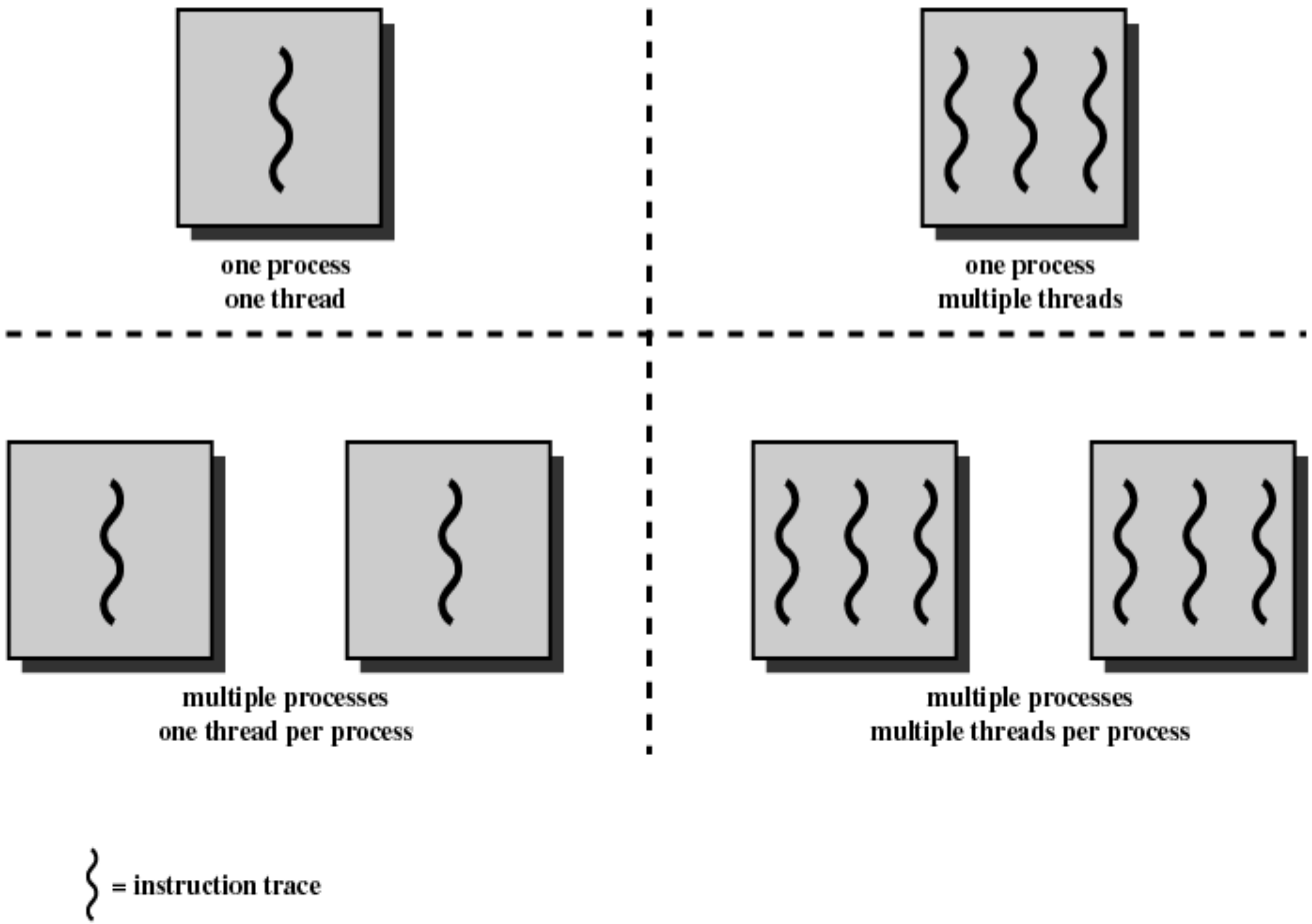
Threads

Chapter 4

Definition of Threads

- Basic unit of CPU utilization.
- Execution context that is independently scheduled but shares a single address space with other threads.
- Traditional Process = Single threaded process (single thread of execution per process – The concept of thread not recognized)
- Multithreaded Process =Multiple threads of execution.


one process
one thread
one process
multiple threads
multiple processes
one thread per process
multiple processes
multiple threads per process
= instruction trace

What is associated with a process?

- A virtual address space which holds the process image
  - It includes:
    - PCB,
    - user-address space (data & codes segments),
    - user and kernel stack
    - OS resources (open files & signals)
- Protected access to processor, other processes (IPC), files & I/O resources.

Thread ?

- An execution state (running, ready, etc.)
- Saved thread context when not running
- **<u>Separate for each thread</u>**
  - Thread ID, PC, register set, scheduling properties

    (TCB = thread control block)
  - User and kernel stack (execution stack; some per-thread static storage for local variables)
- **<u>Shared Among Threads</u>**
  - Access to the memory and resources of its process
  - User address space (data & code segment)
  - OS resources (open files & signals)

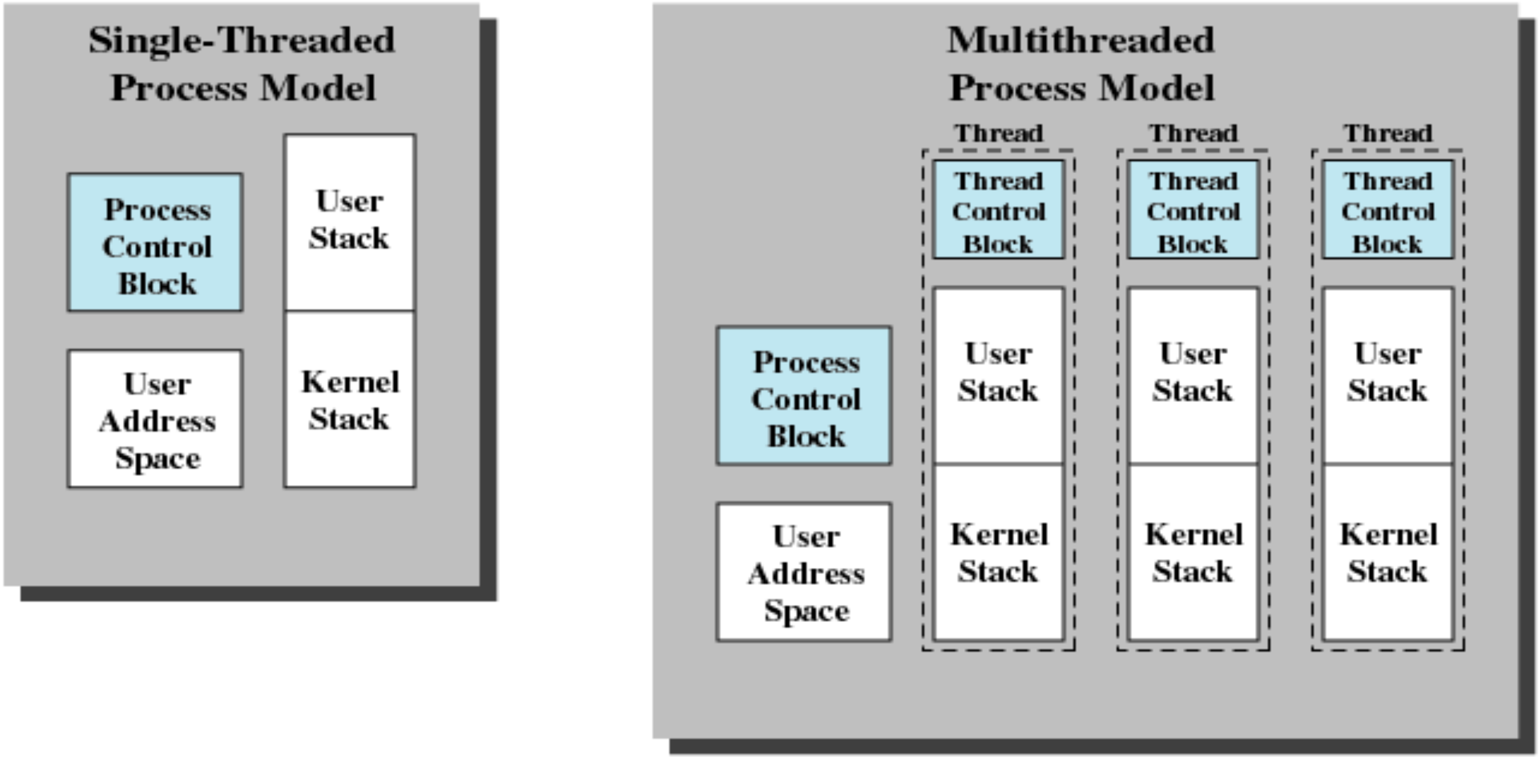


Figure 4.2 Single Threaded and Multithreaded Process Models

Benefits of Threads

1. Economy
   - Takes less time to create a new thread than a process
   - Less time to terminate a thread than a process
   - Less time to switch between two threads within the same process
   - Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

Benefits of Threads (2)

2. Responsiveness
   - allow a program to continue running even if part of it blocked
3. Resource Sharing
   - Share memory and resources of the process

Examples (1)

- Foreground to background work
  - web browser = one thread display image / text; another thread retrieve data from network
  - spreadsheet program = one thread display menu & read user input; another execute user command and update the spreadsheet
  - web server = when it receives a request:
    - create a separate process to service the request (extra overhead)
    - Solution: multiple threads to serve the same purpose

Examples (2)

- Asynchronous processing
  - Word processor: write its RAM buffer to disk once every minute.
- Speed of execution
  - Compute one batch of data while reading the next batch from a device.

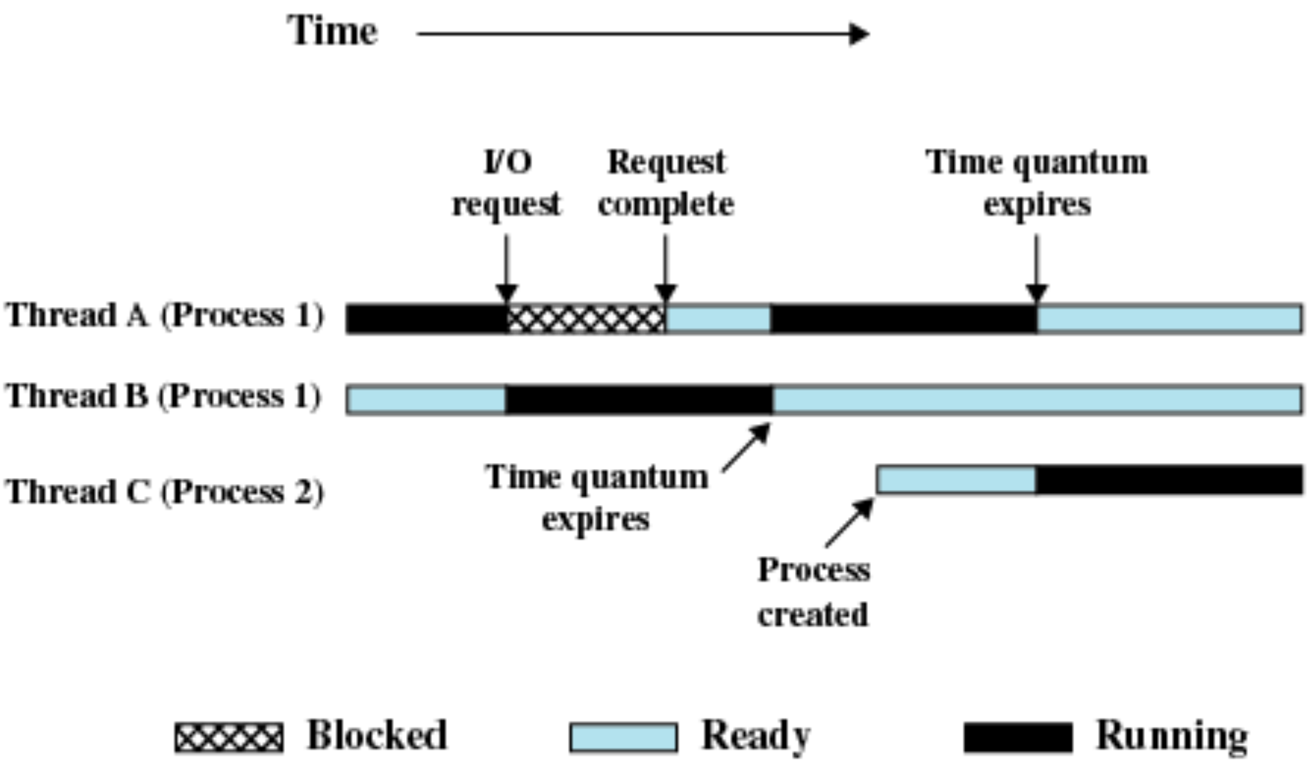
Suspension and Termination of Threads

- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process, terminates all threads within the process

Thread States

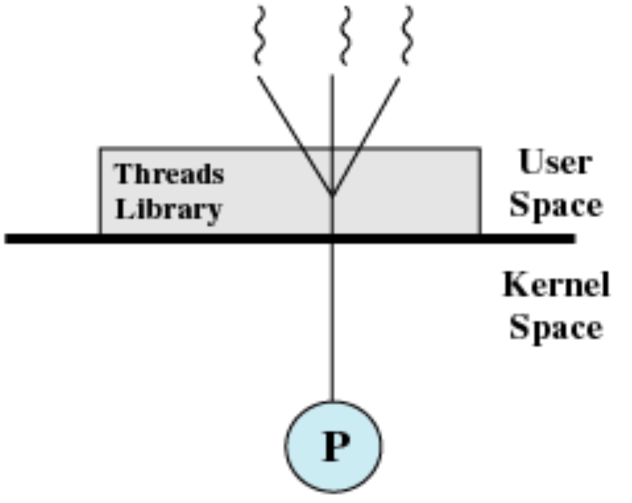
- States (running, ready, blocked)
- Basic thread operations associated with a change in thread state
  - Spawn (spawn another thread)
  - Block
  - Unblock
  - Finish (deallocate register context and stacks)

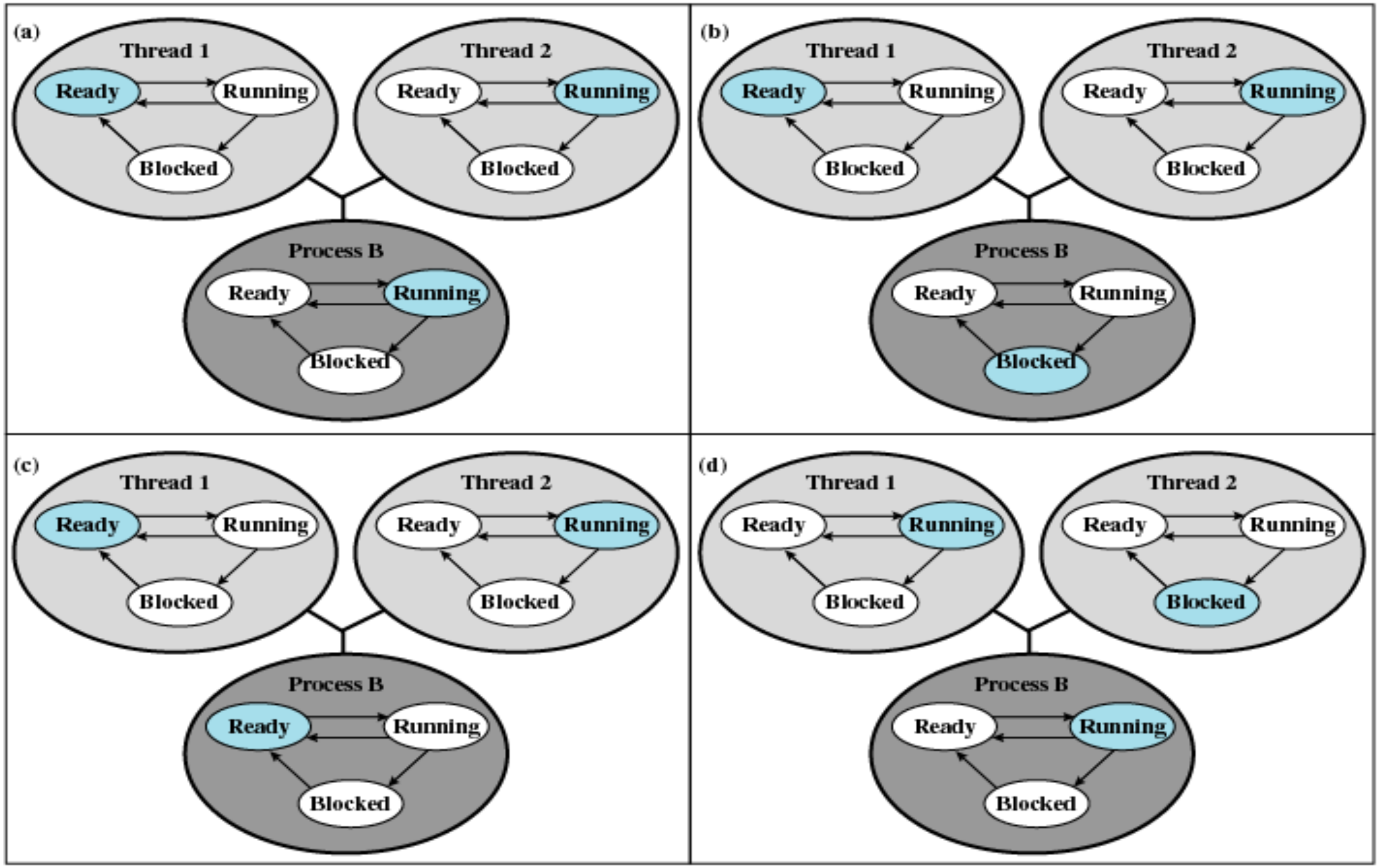
Multithreading



User-Level Threads

- All thread management is done by the application
- The kernel is not aware of the existence of threads



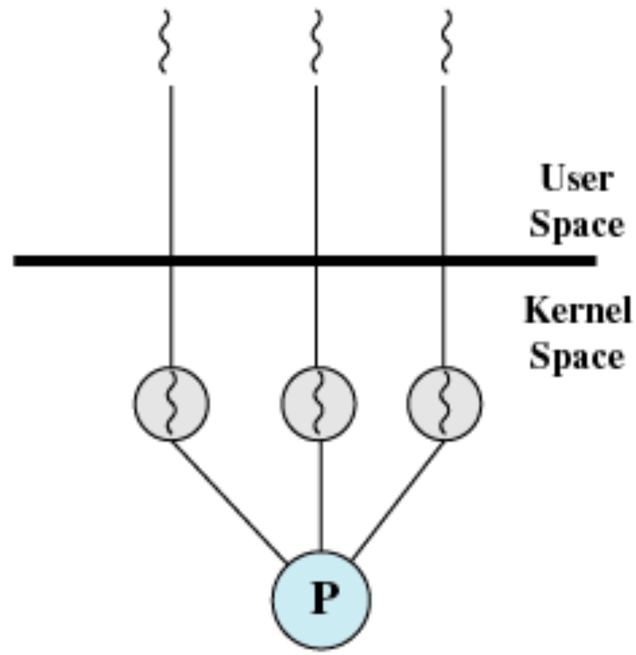


Colored state
is current state

Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

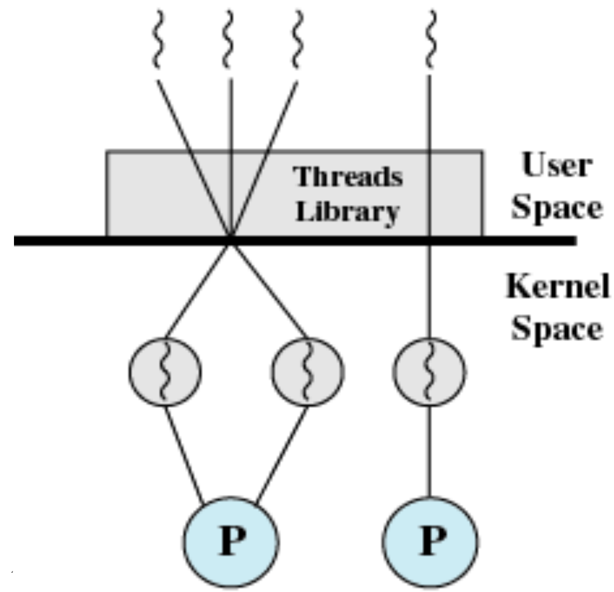
Kernel-Level Threads

- All contemporary OS support kernel-level threads (such as Windows XP, Linux, Mac OS)
- Kernel maintains context information for the process and the threads
- Scheduling is done on a thread basis



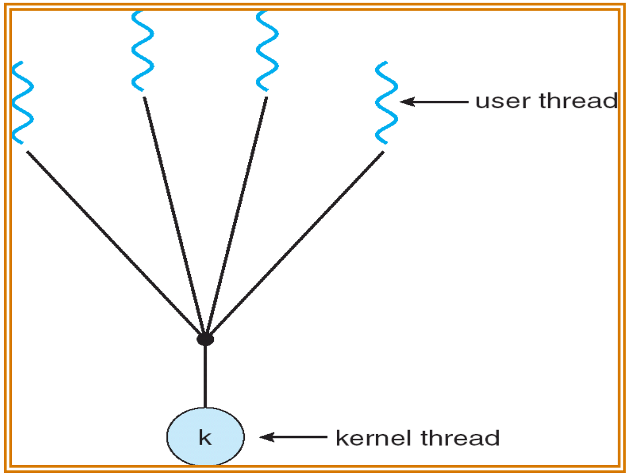
Combined Approaches

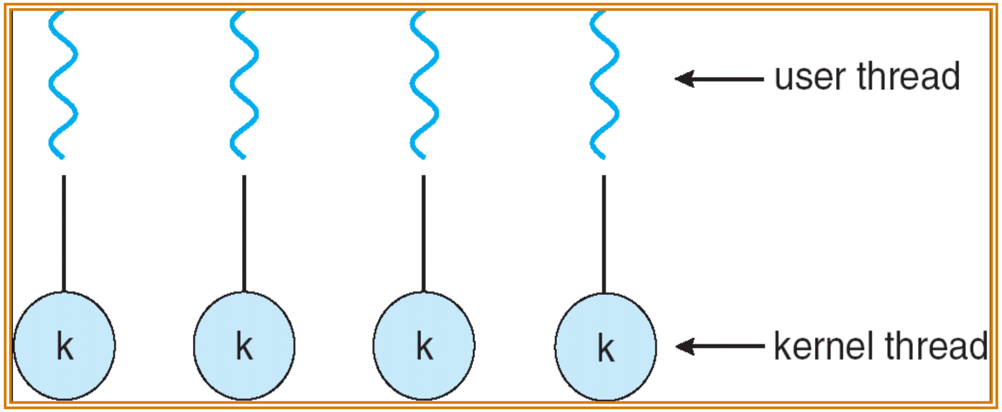
- Example is Solaris
- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads within application



Multithreading Models (Relationship between KLT and ULT)

- Many-to-One Model
  - Many user-level threads mapped to single kernel thread

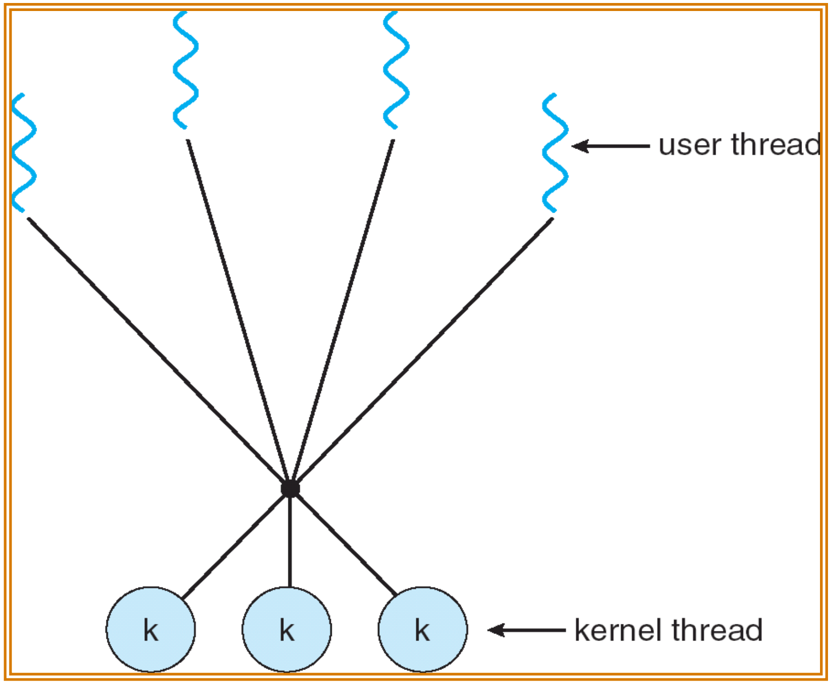




- One-to-One Model

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



Thread Cancellation

- Terminating a thread before it has finished
  - Ex: User presses a stop button on a web browser that stop a web page from loading any further
- Cancellation of a target thread may occur in two different scenarios:
  - **Asynchronous cancellation.** One thread terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

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

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

- May be provided either as user-level or kernel-level
- 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
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

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.

Signal Handling

- Used in UNIX systems to notify a process that a particular event has occurred
- Synchronous Signals : delivered to the same process that performed the operation
  - illegal memory access, division by zero
- Asynchronuos Signals: generated by an event external to a running process
  - terminating a process with specific keystrokes (such as <control><C>)

Signal Handling (2)

- All signals follow the following pattern:
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Once delivered, the signal must be handled.

- Handling Signals in multithreaded programs
  - 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