Thread (I)

Kernel Threads

September 1, 2017

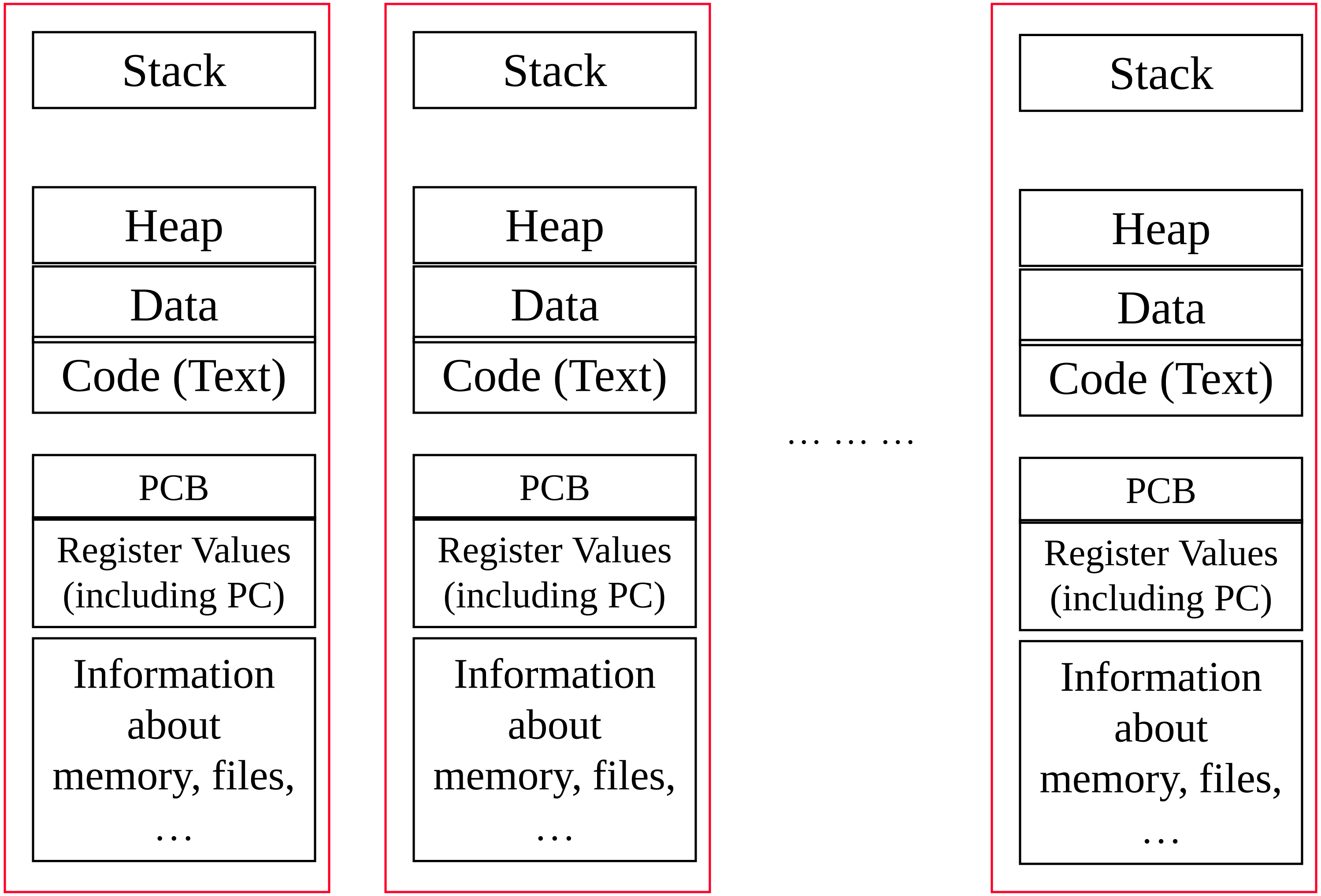
Example of Multi-Process Collaboration

- Finding the lines containing key word “Linux” in all .txt files in the current directory
- Linux command line:

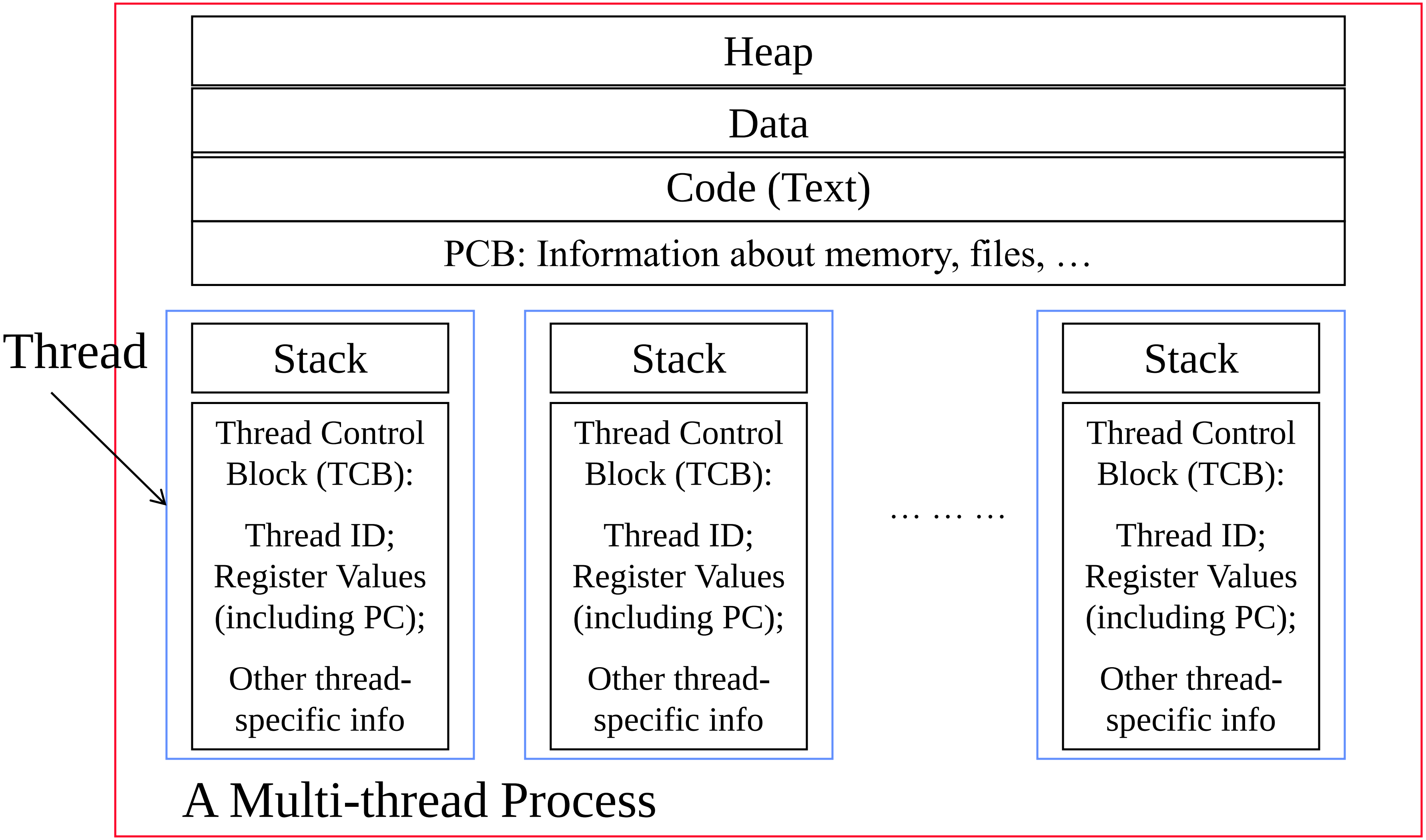
  grep “Linux” *.txt
- Using multi-process collaboration to do the job?

```
…
int numFiles; //total number of files
char fileNames[MAX_NUM_FILES][255]; //array storing the file names
int i;
for(i=0; i<numFiles; i++)
  if (fork( ) == 0){
        FILE *fp = fopen(fileNames[i], “r”);
        … //search for lines containing “Linux” in the file
        exit(0);
  }
```

Multi-Process



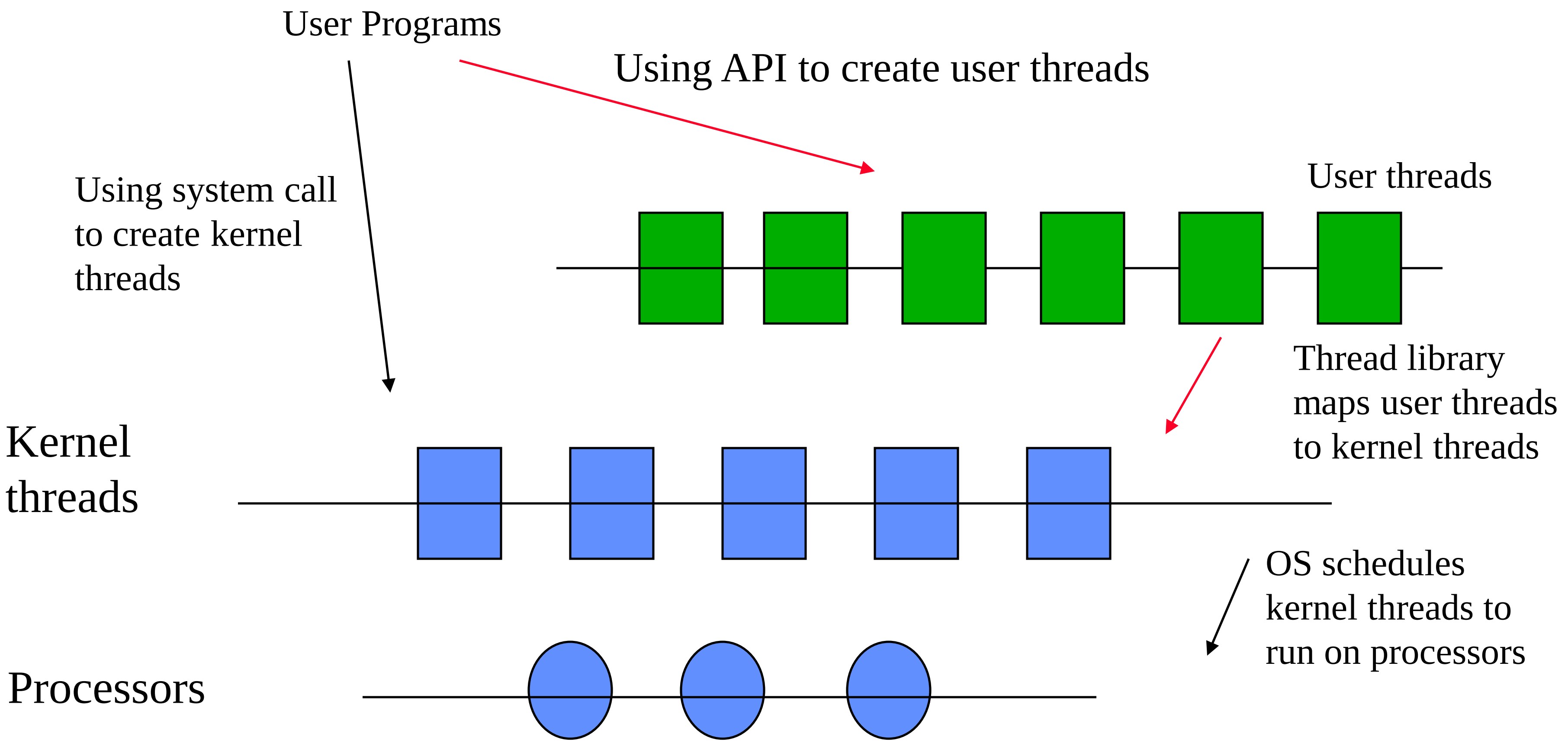
A Process with Multiple Threads



Benefits

- Responsiveness (similar to multi-process)
  - Allowing a program to continue running even if part of it is blocked
  - Allowing a program to perform multiple tasks concurrently
- Resource sharing
  - Threads of the same process share the memory and resources
- Economy
  - Allocating memory and resources for process creation is costly.
- Scalability (similar to multi-process)
  - A single-thread process can only run on one processor regardless how many are available; multi-threaded process can increase parallelism on a multi-processor machine.

Threads in a Computer System



Kernel Threads

- Directly created/managed/scheduled by the OS kernel
- Virtually all contemporary OSes support kernel threads

Linux Kernel Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call

```
#define _GNU_SOURCE
#include <sched.h>
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

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.

Example

```
#include _GNU_SOURCE
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>

int variable, fd;
int do_something(); //function to be executed by the new task (thread)
```

Example

```
int main( ) {
void *child_stack;
char tempch;
variable = 9;
fd = open("test.file", O_RDONLY);
child_stack = (void *)malloc(16384); child_stack += 16383;
printf("The variable was %d\n", variable);
clone(do_something, child_stack, CLONE_VM|CLONE_FILES, NULL); /*A*/
sleep(1);
printf("The variable is now %d\n", variable);
if (read(fd, &tempch, 1) < 1) {
        perror("File Read Error"); exit(1);}
        printf("Parent could read from the file\n");}
}
```

Example

```
int do_something() {
   char tempch;
   variable = 42;
   if (read(fd, &tempch, 1) < 1) {
      perror("File Read Error");
      _exit(1);
   }
   printf("Child could read from the file\n");
   close(fd);
   _exit(0);
}
```

Example

1. What is the output of the program?

2. What if Line A is changed to:

clone(do_something, child_stack, CLONE_VM, NULL);

or

clone(do_something, child_stack, CLONE_FILES, NULL);

or

Clone(do_something, child_stack, 0, NULL);

Thread (II)

User Threads & User/Kernel Mapping

September 1, 2017

User Threads

- Created by a thread library and scheduling is managed by the library itself in user space (the existence of user threads is unknown to the OS)

User Thread Libraries

- Pthreads (POSIX Threads)
  - 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)
- Java Threads
  - Managed by the JVM
  - May be created by:
    - Extending Thread class
    - Implementing the Runnable interface
- Win32 Threads
  - Similar to pthread, differing in function names

PThreads
include <pthread.h>

- int **pthread_create** **(** pthread_t * thread,
  pthread_attr_t * attr,
  void * (*start_routine)(void *),
  void *arg **)**
  - Creates a new thread
  - 1st argument: the address of the created thread
  - 2nd argument: the address of the structure containing the attributes of the created thread
    - For default attribute values, make this argument as ***NULL***
  - 3rd argument: the pointer to the function that is the code for the created thread
  - 4th argument: the pointer to the argument of the start_routine function. If multiple arguments are needed, define a data structure that contains all arguments.

PThreads
include <pthread.h>

- void **pthread_exit** **(**void *****status **)**
  - Terminate execution of the calling thread
  - 1st argument: Points to an optional termination status. If no termination status is desired, its value should be NULL.
- pthread_t **pthread_self** (void)
  - returns the *pthread* handle (ID) of the calling thread

PThreads
include <pthread.h>

- int **pthread_join** (pthread_t tid, void **status)
  - "Joining" is one way to accomplish synchronization between threads.
  - Blocks the caller until the specified thread terminates
  - 1st argument: the *id* of the thread to be waited
  - 2nd argument: the address of the variable to receive the thread's exit status
    - Usually set to ***NULL*** or ***int****

Example: Hello World!

- This simple example code creates 5 threads with the pthread_create() routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit().

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5


void *PrintHello(void *threadid)
{
        printf("\n%d: Hello World from Thread #%d !\n",
                threadid, pthread_self());
        pthread_exit(NULL);
}
```

Example: Hello World! (con't)

```
int main (int argc, char *argv[])
{
        pthread_t threads[NUM_THREADS];
        int rc, t;
        for (t=0;t < NUM_THREADS;t++) {
             printf("Creating thread %d\n", t);
             rc = pthread_create(&threads[t], NULL, PrintHello, (void *) t);
             if (rc){
                printf("ERROR; return code from pthread_create() is %d\n", rc);
                exit(-1);
             }
         }
         for(t=0; t<NUM_THREADS;t++) pthread_join(threads[t],NULL);
         pthread_exit(NULL);
}
```

Example: Compilation

Suppose the program file is named: HelloWorld.c

$gcc –o HelloWorld HelloWorld.c -lpthread

Java Threads

- Defining a class X that implements the Runnable interface to contain the implementation of a thread

  ```
  public interface Runnable
  {
      public abstract void run( );
  }
  ```
- Creating a Thread object to wrap the class X
- Manipulate the Thread object by calling its methods
  - run( ) method
  - join( ) method
  - … …

Example: Summation

```
class Summation implements Runnable
{
    private int upper;
    public int sum;
    public Summation(int upper, int sum){
            this.upper = upper;
            this.sum = sum;
    }
    public void run( ) {
            int sum = 0;
            for (int i=0; i<=upper; i++) sum += i;
            System.out.println("Sum of 1 through " + upper +" is "+sum);
    }
}
```

Example: Summation

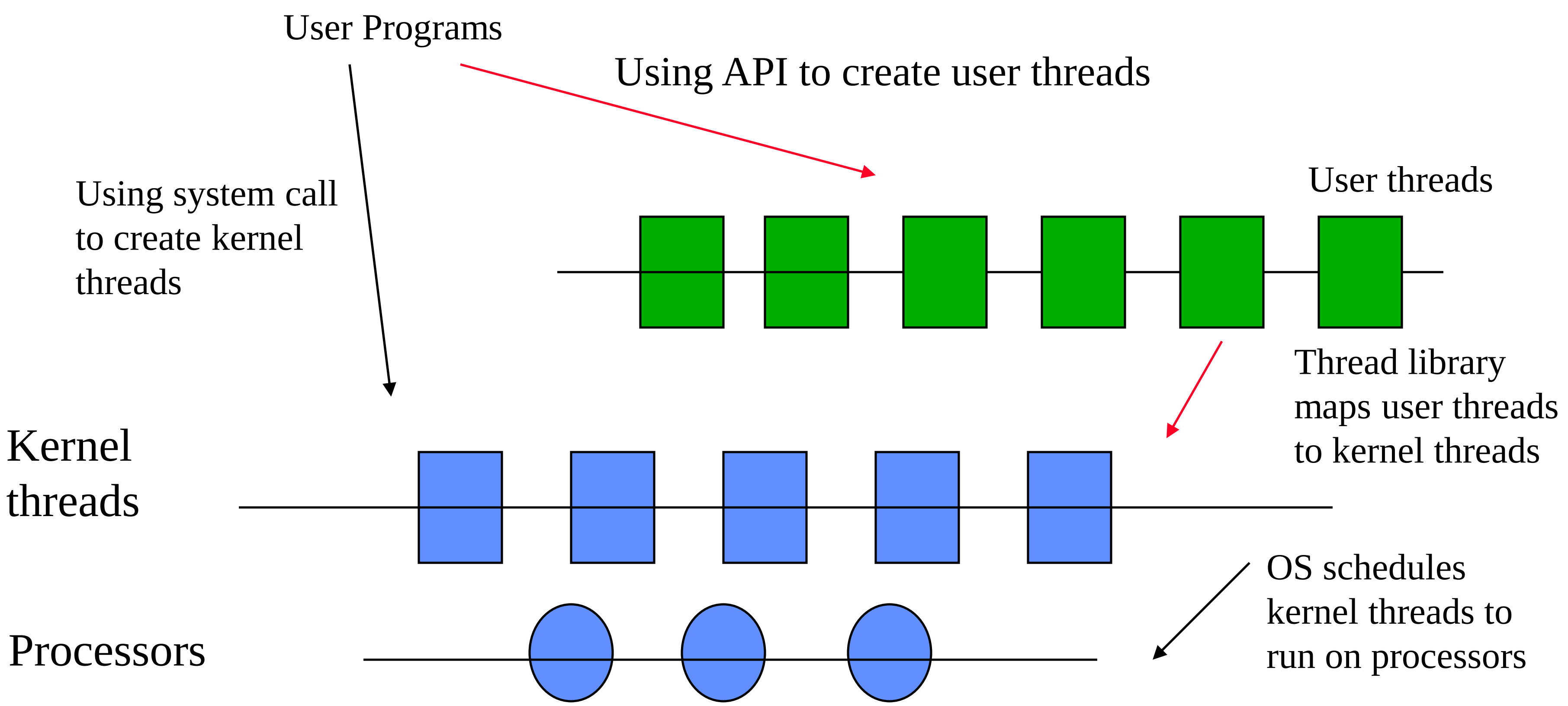
```
public class Driver{
    public static void main(String[ ] args){
        if (args.length >0){
          if (Integer.parseInt(args[0])<0)
            System.err.println(args[0]+"must be >=0");
          else{
            int upper = Integer.parseInt(args[0]);
            Summation sumObj = new Summation(upper,0);
            Thread thrd=new Thread(sumObj);
            thrd.start( );
            try{thrd.join( );}catch(InterruptedException ie){ }
        }
    }
}
```

Example: Summation

$javac Summation.java
$javac Driver.java
$java Driver 100
Sum of 1 through 100 is 5050

Threads in a Computer System

- A thread library provides the programmer with an API for creating and managing threads.



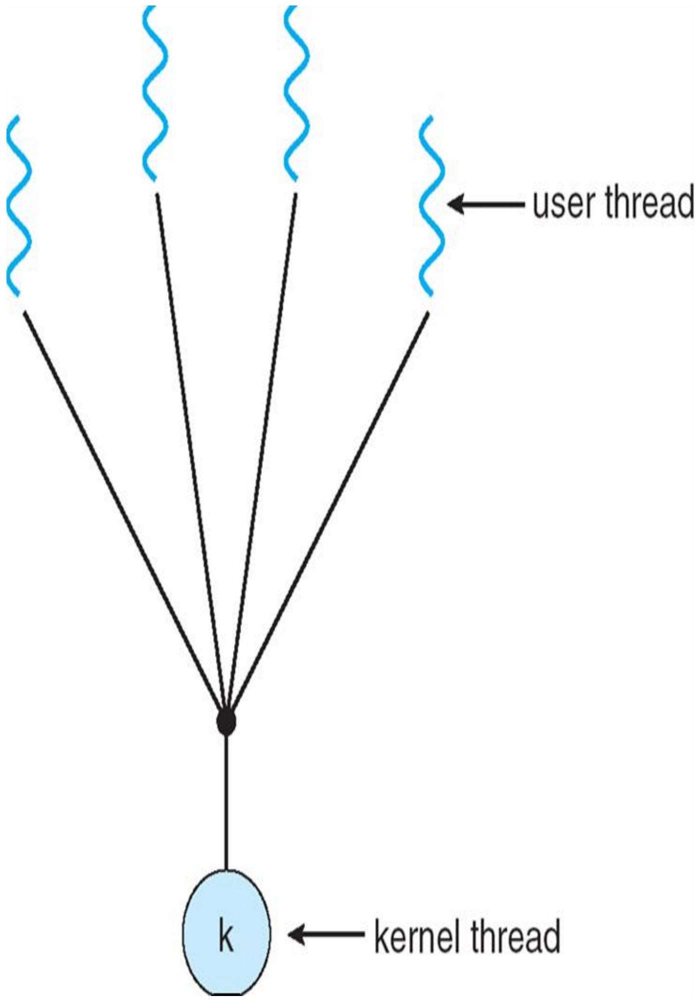
Kernel Threads vs. User Threads

- Kernel threads:
  - Directly created/managed by the OS kernel
- User threads
  - created by a thread library and scheduling is managed by the library itself (the existence of user threads is unknown to the OS)
  - low-cost in thread creation; portable
  - may not utilize multi-processors efficiently (depends on mapping strategy between user and kernel threads)
  - When executed, must be mapped to kernel threads

Mapping User Threads to Kernel Threads

- Many to One
- One to One
- Many to Many

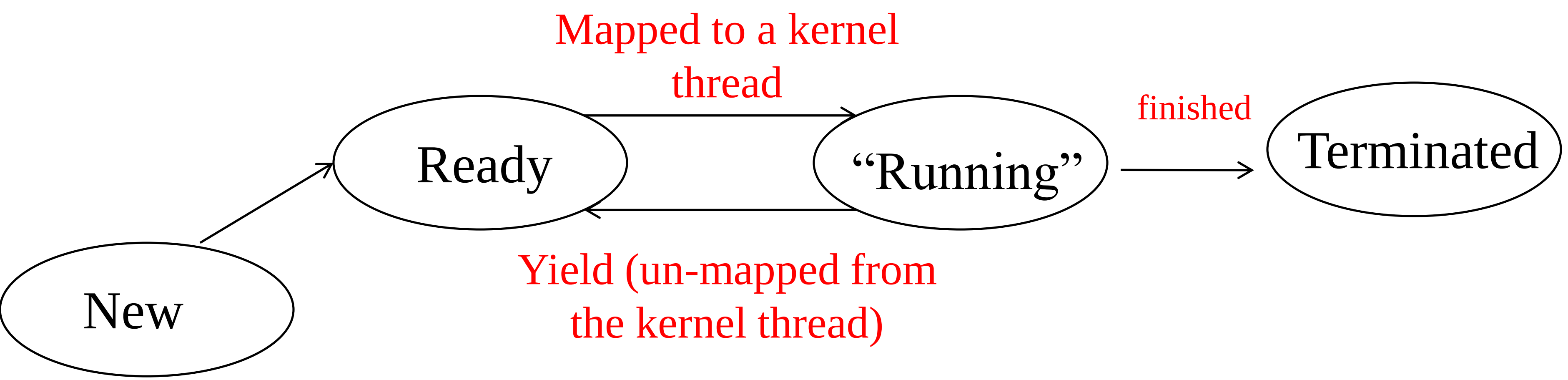
Many-to-One



- Many user-level threads within the same process are mapped to single kernel thread
- User thread library schedules user threads to the kernel thread
- The multiple threads “time-share” the single kernel thread
  - A kernel thread is created for a process
  - Upon the thread yields, switch to another thread in the same process (kernel is unaware of it)
  - Upon the thread blocks, all threads in the process block

Thread Scheduling in User Space

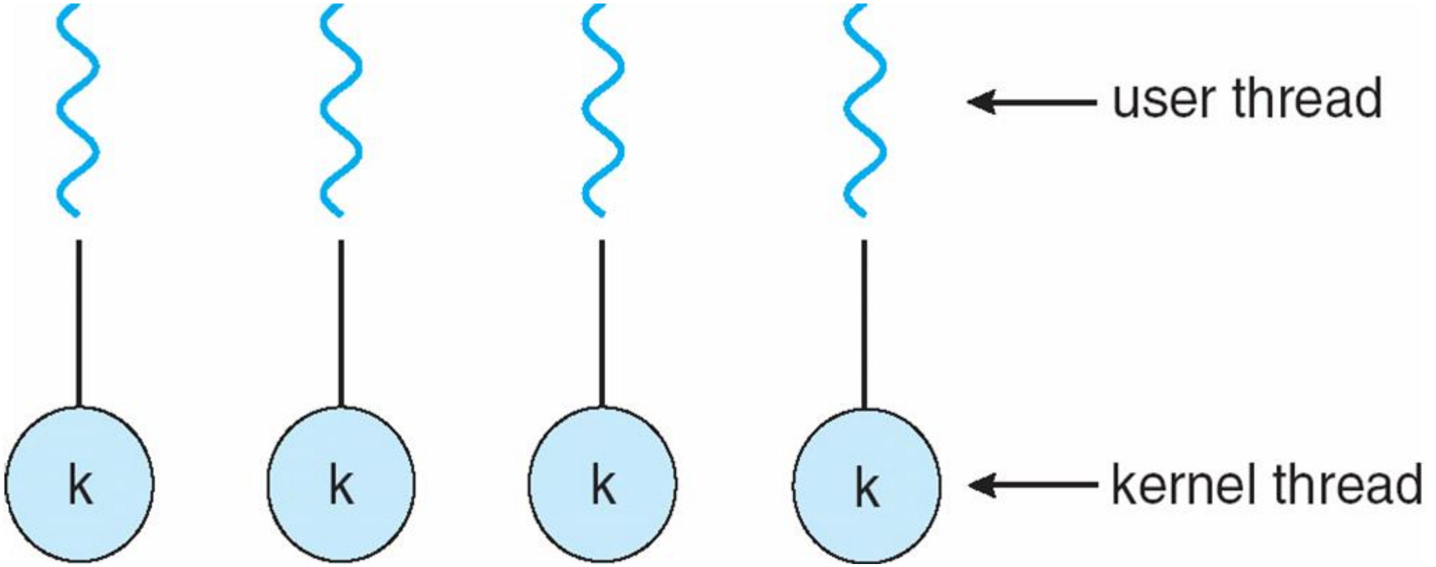
- User-space scheduling: managing the (dynamic) mapping from user threads to kernel threads.



Lifetime of a user thread

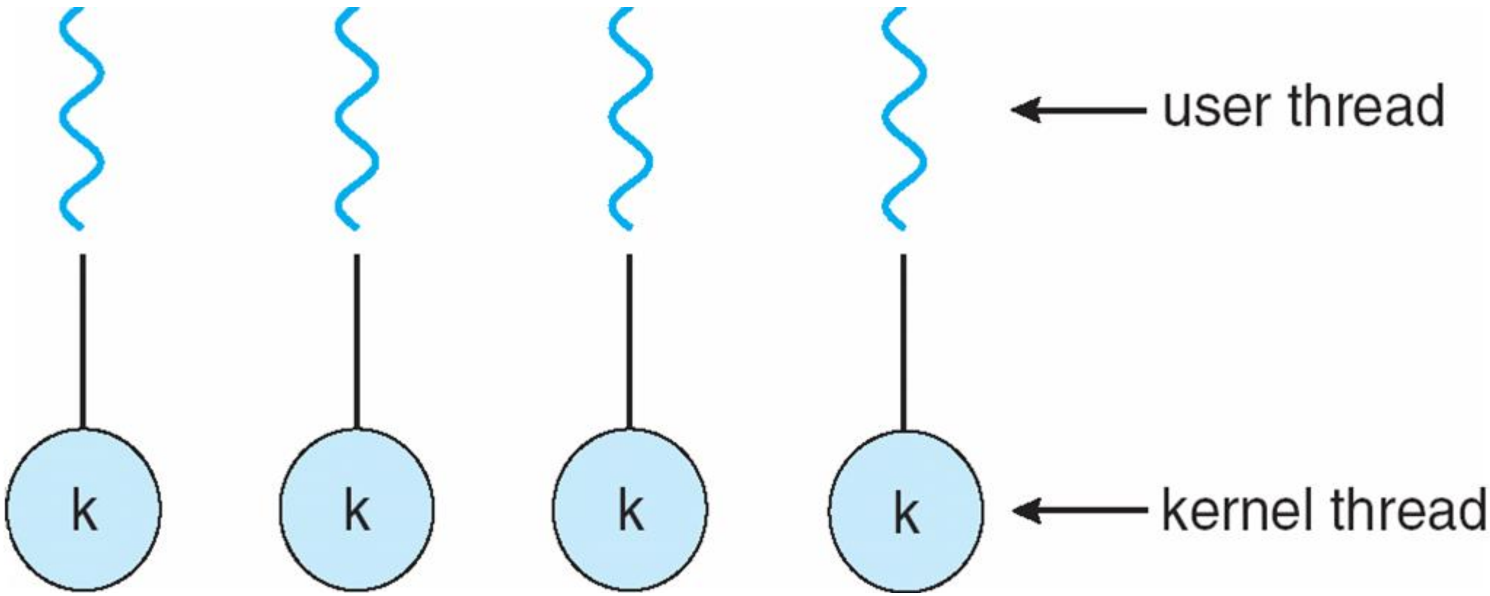
One-to-One

- Each user-level thread maps to a kernel thread
- User thread library simply provides a portable interface for thread creation/management, of which the implementation rely on the kernel thread (kernel of the OS)

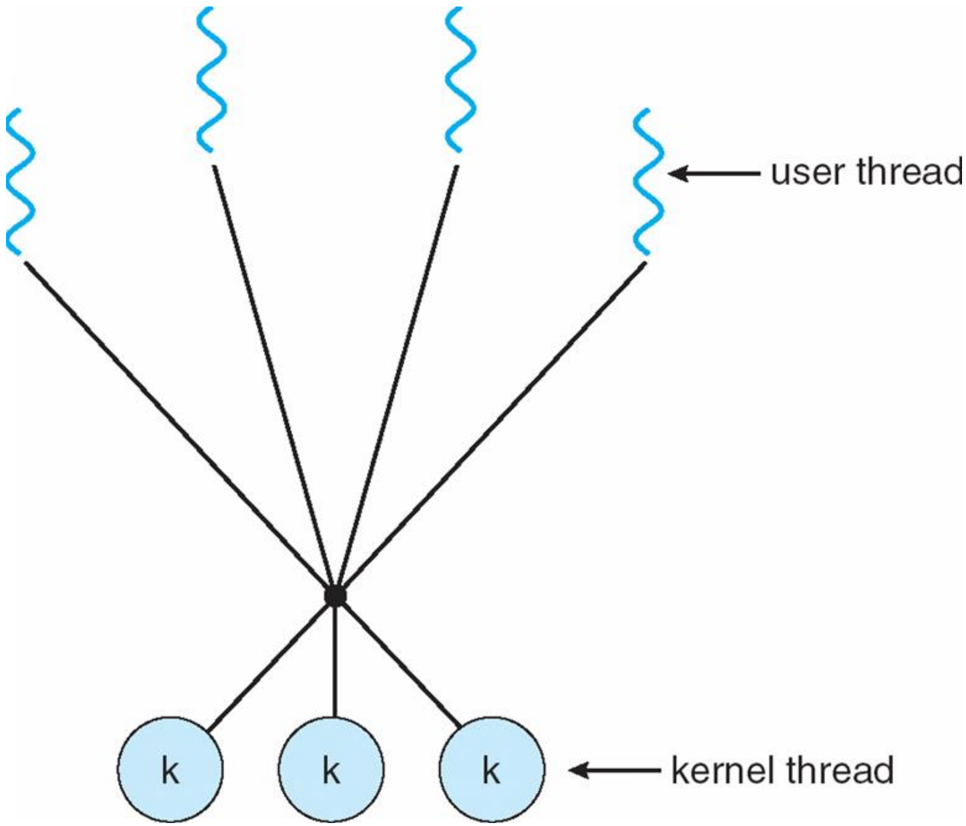


One-to-One

- Upon a thread currently running on a CPU yields or blocks, CPU can be switched to another thread (kernel is aware of this)

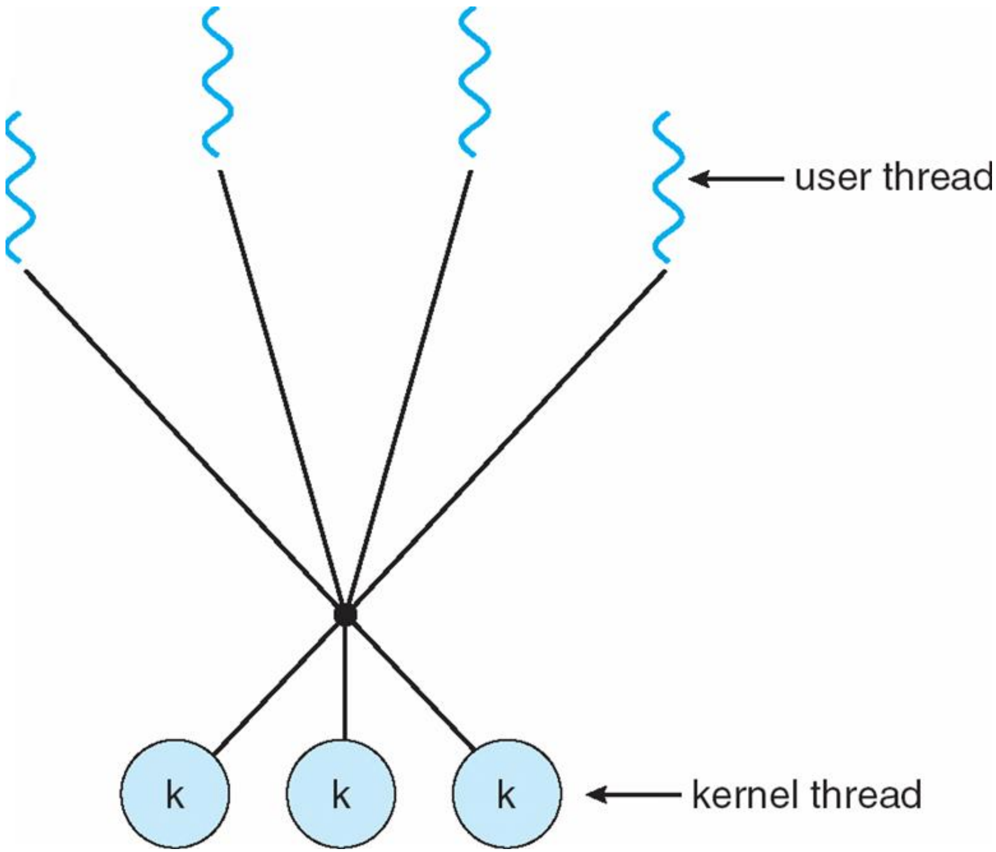


Many-to-Many Model



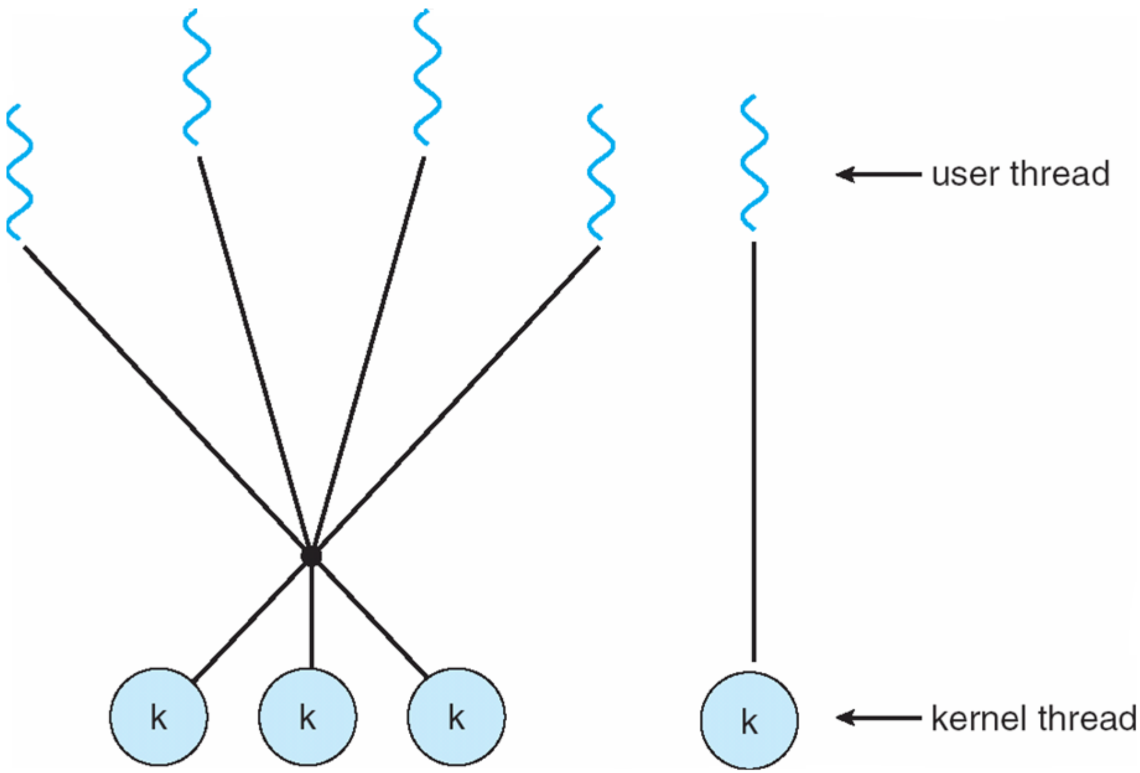
- Allows many user level threads to be mapped to many kernel threads
- User thread library schedules user threads to kernel threads

Many-to-Many Model



- Upon a currently-running user thread yields, another thread may be mapped to the yielding user thread's kernel thread and runs
- Upon a currently-running user thread blocks, other thread may run on the CPU

Two-level Model



- A sub-type of M:M
- It allows a user thread (e.g., with high priority) to be **bound** to a kernel thread