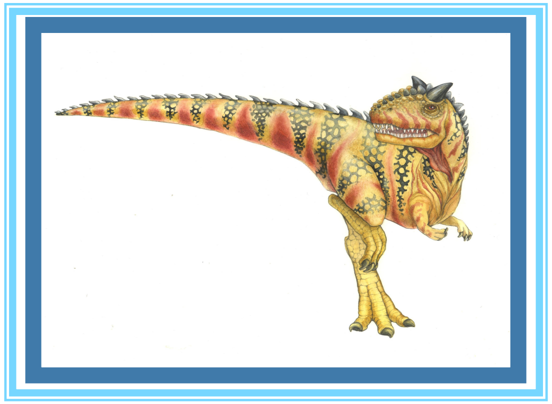
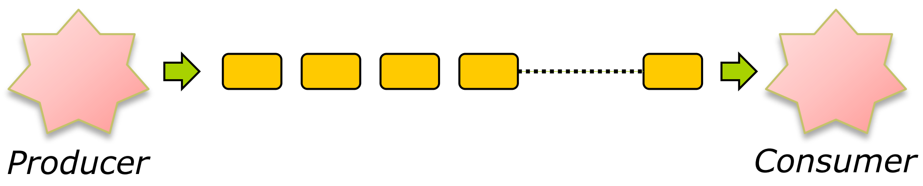
Chapter 3: Processes-Threads



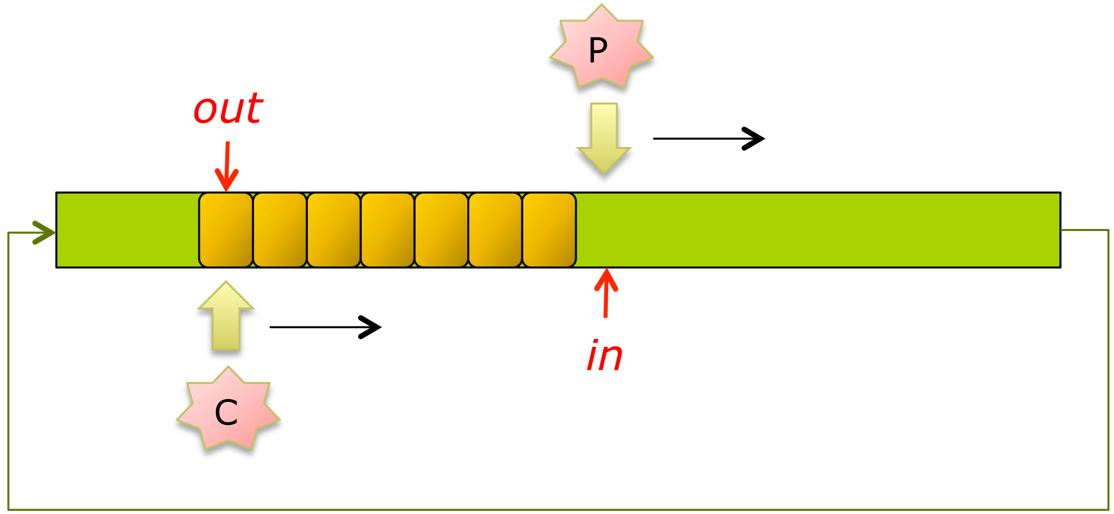
Producer-Consumer Model

- Producer-Consumer Model
  - Producer only produces (writes) information and Consumer only consumes (reads) the information



  - Use *Buffer* to deliver information from producer to consumer

Shared Buffer by Circular Array




```
#define BS 100
typedef struct {…} item;

item buf[BS]
int in = 0
int out = 0
```

* Buffer is empty if
  i == j
* Buffer is full if
  (in+1)%BS == out
* Maximum items count
  BS-1

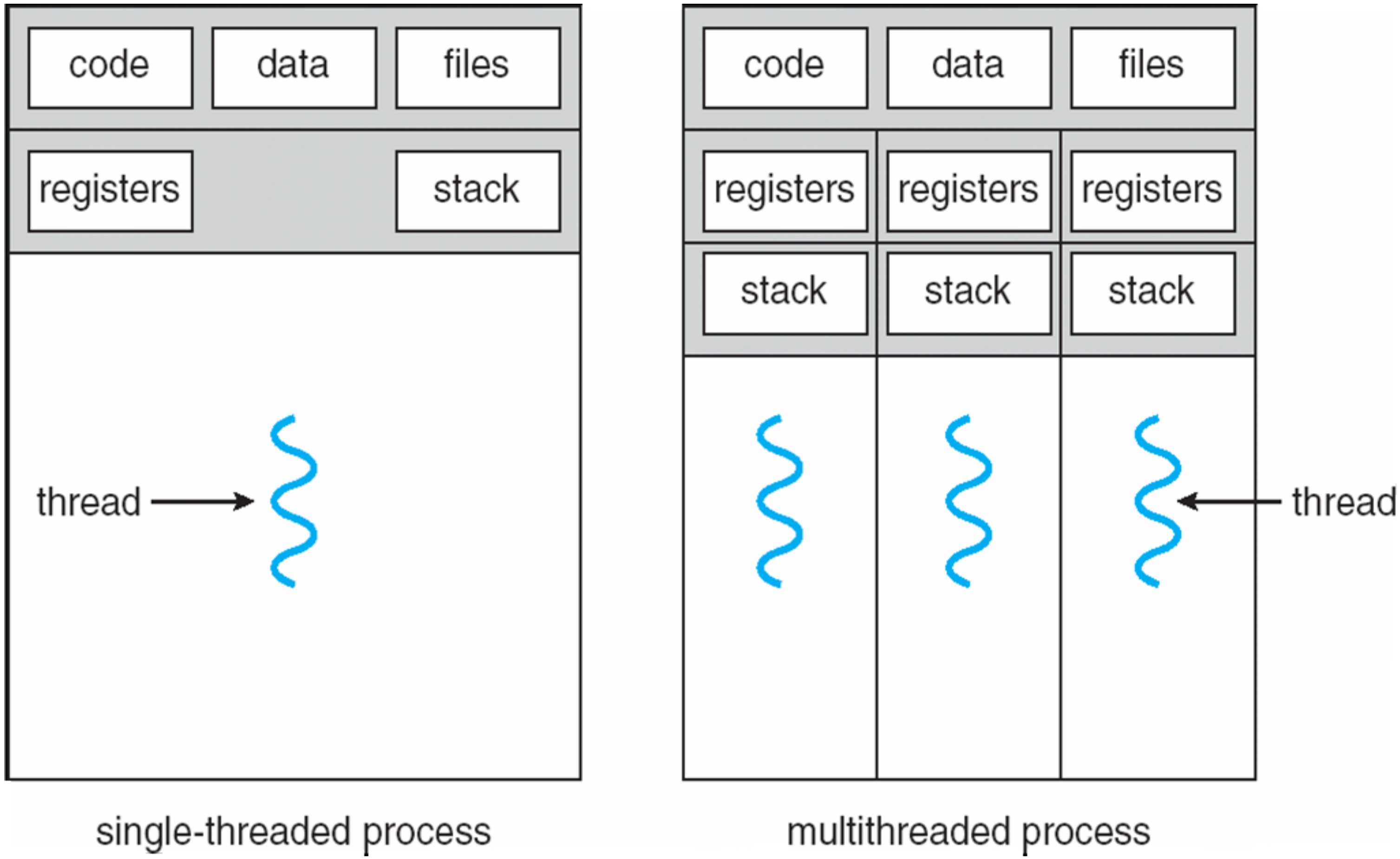
Motivation

- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Examples

- Web browser
  - Thread 1: display images
  - Thread 2: show text
  - Thread 3: retrieve data from the network
- Word processor
  - Thread 1: display graphics
  - Thread 2: respond to key strokes
  - Thread 3: spelling and grammar checking
- Web server
  - use thread instead of process
- Kernel

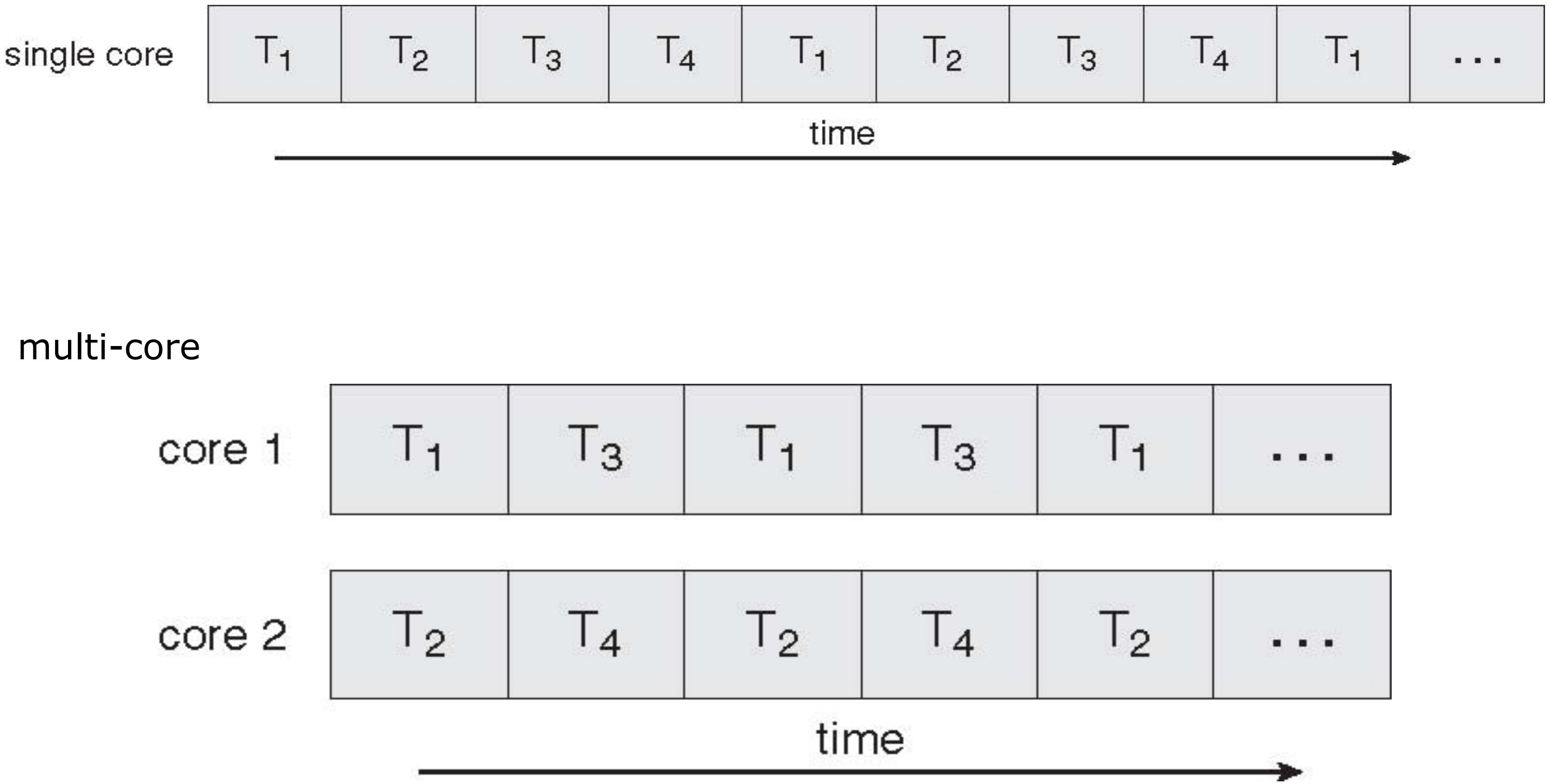
Single and Multithreaded Processes



Benefits

- Responsiveness
- Easy Resource Sharing
  - Threads use the same address space
- Economy
  - Cheaper creation, context switch
- Scalability
  - Threads running on different processors

Parallel Execution on a Multicore System



Multicore Programming

- Multicore systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

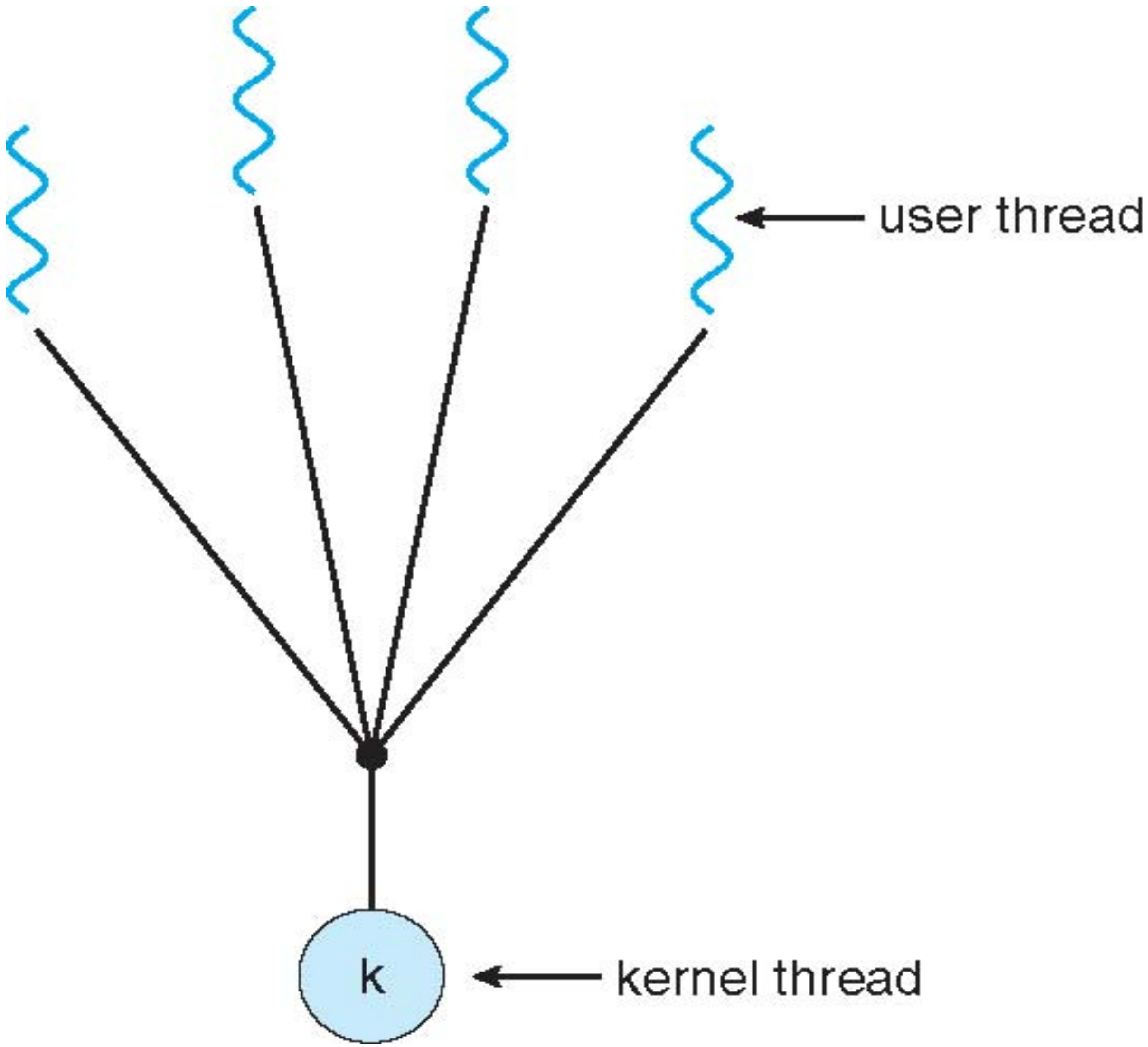
Multithreading Models

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

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

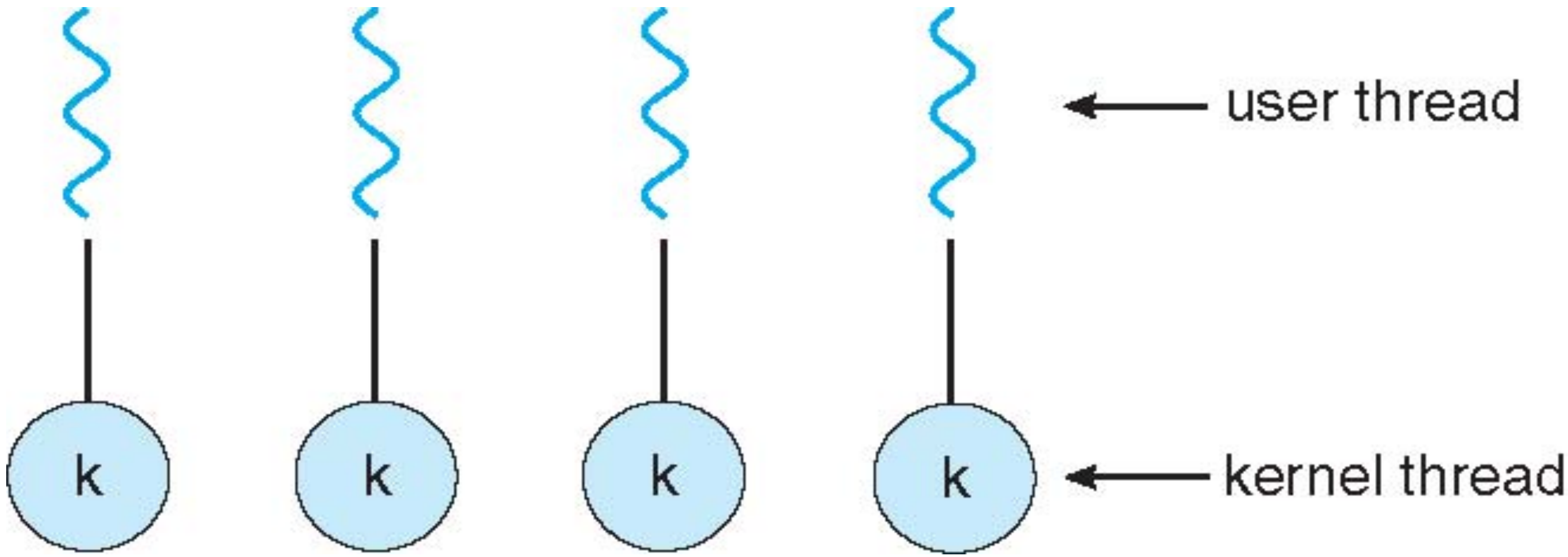
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

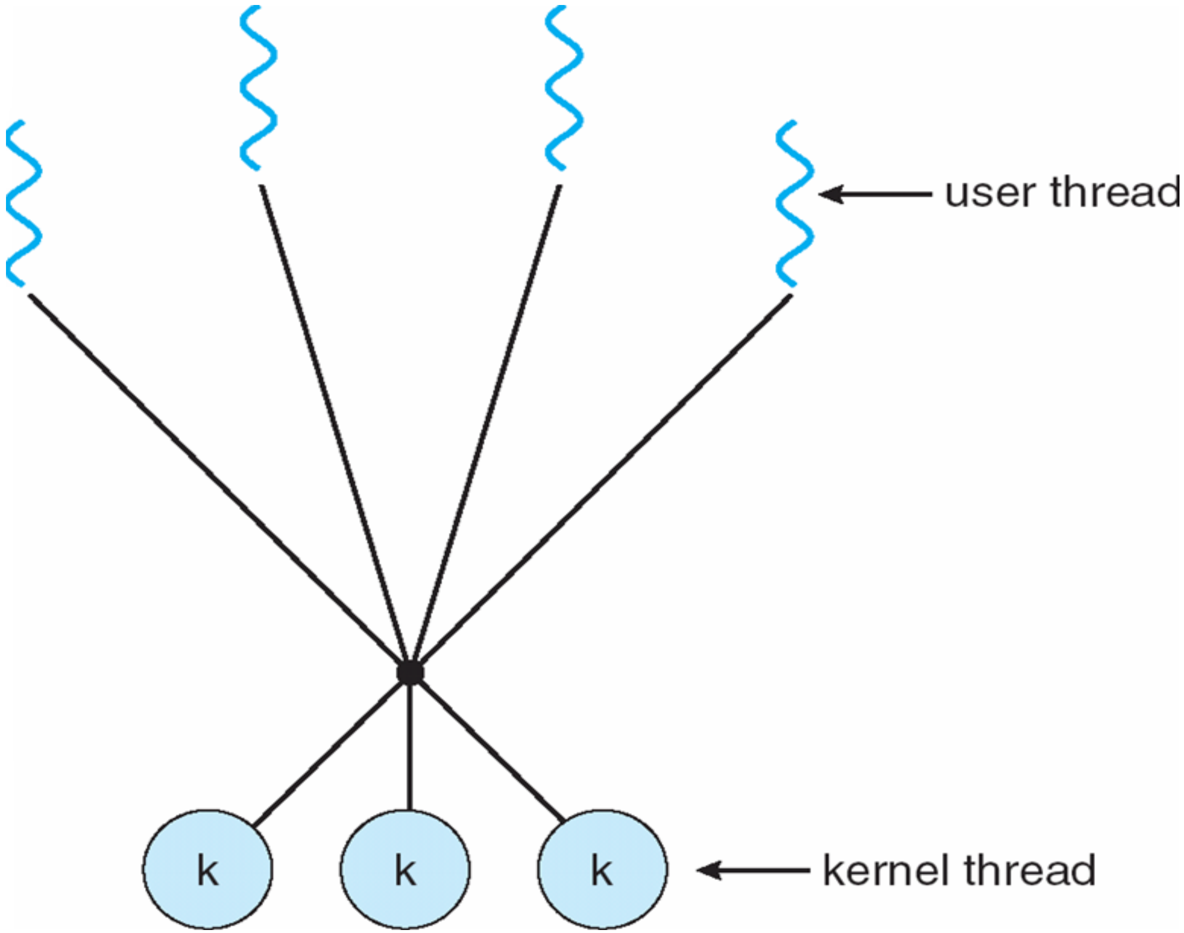
One-to-one Model



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
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

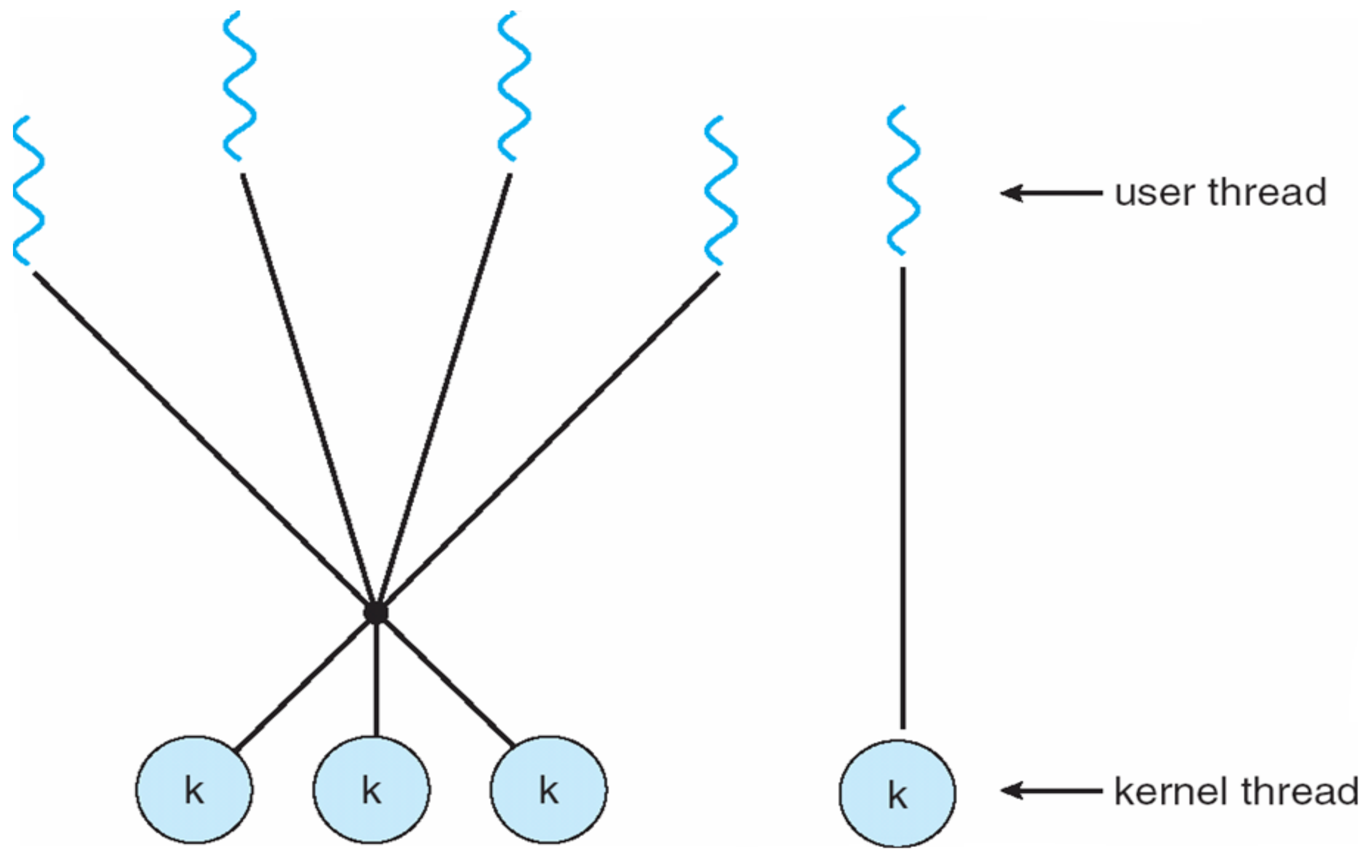
Many-to-Many Model



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

Two-level Model



Thread Libraries

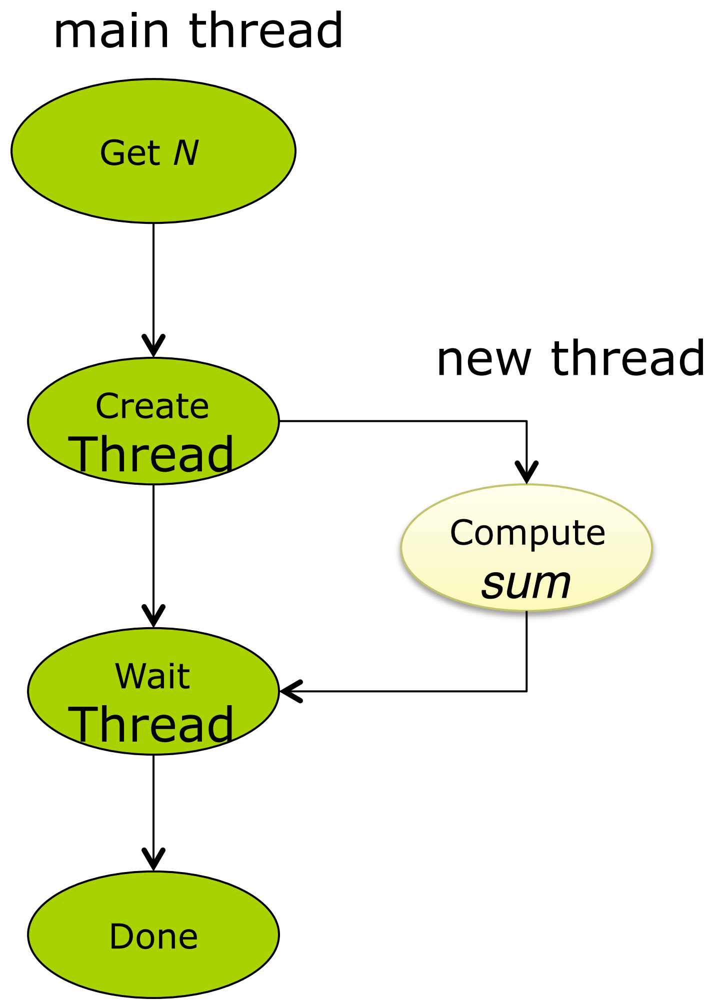
- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Three primary thread libraries:
  - POSIX **Pthreads**: user or kernel level
  - Win32 threads: kernel level
  - Java threads: depends on the host system

Example program

- Compute

$$sum = \sum_{i=0}^{N} i$$

- Execution
  - a.out N



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 */

   /* 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);
}
```

```
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Win32 API Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   // 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);
   }
}
```

```
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

Java Threads

- Java threads are managed by the JVM
  - based on the thread model of the host machine
- Rich support for threads
- All Java program is run as a thread in JVM
- Java threads may be created by
  - *Thread* Class
    - Extend *Thread* class
    - Implement *run*() function
  - Runnable interface
    - *Implement Runnable* interface,
    - Implement run() function

Java Multithreaded Program

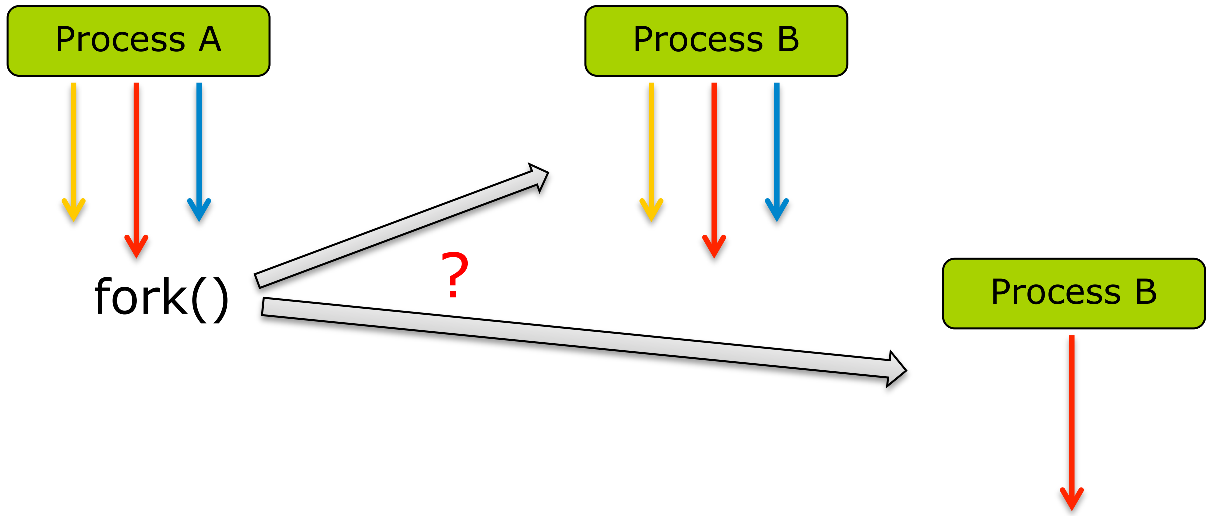
```
class Sum
{
  private int sum;

  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 {
     // create the object to be shared
     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>"); }
}
```

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- **Thread cancellation** of **target thread**
  - Asynchronous or deferred
- **Signal** handling
  - Synchronous and asynchronous
- **Thread pools**
- **Thread-specific data**
  - Create Facility needed for data private to thread
- **Scheduler activations**

Semantics of fork() and exec()



- When fork() in a multi-threaded process
  - Copy all the threads?
  - or only the fork-calling thread?
- exec() will replace all the threads
- So, if exec() is called after fork, no reason to copy all the threads

Thread Cancellation

- Terminating a thread before it has finished
- Ex
  - Parallel database searching
  - Stopping web browser loading
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
    - may not free resources, abruptly stop writing shared info
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.
    - check at safe cancellation points

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- Options:
  - 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

Signal to multi-threaded process

- Synchronous signal
  - delivered to the thread that caused the signal
- Asynchronous signal
  - delivered to some or all the threads
- Unix allows threads to choose signals to accept
  - but usually it is handled by the first class that accepts
- Signal generation in unix
  - kill(pid, signal)
  - pthread_kill(tid, signal)
- Windows doesn’t support signal, but emulate using APC (Asynchronous Procedure Call)

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
- Pool size:
  - Heuristic choice, or dynamic adjustment

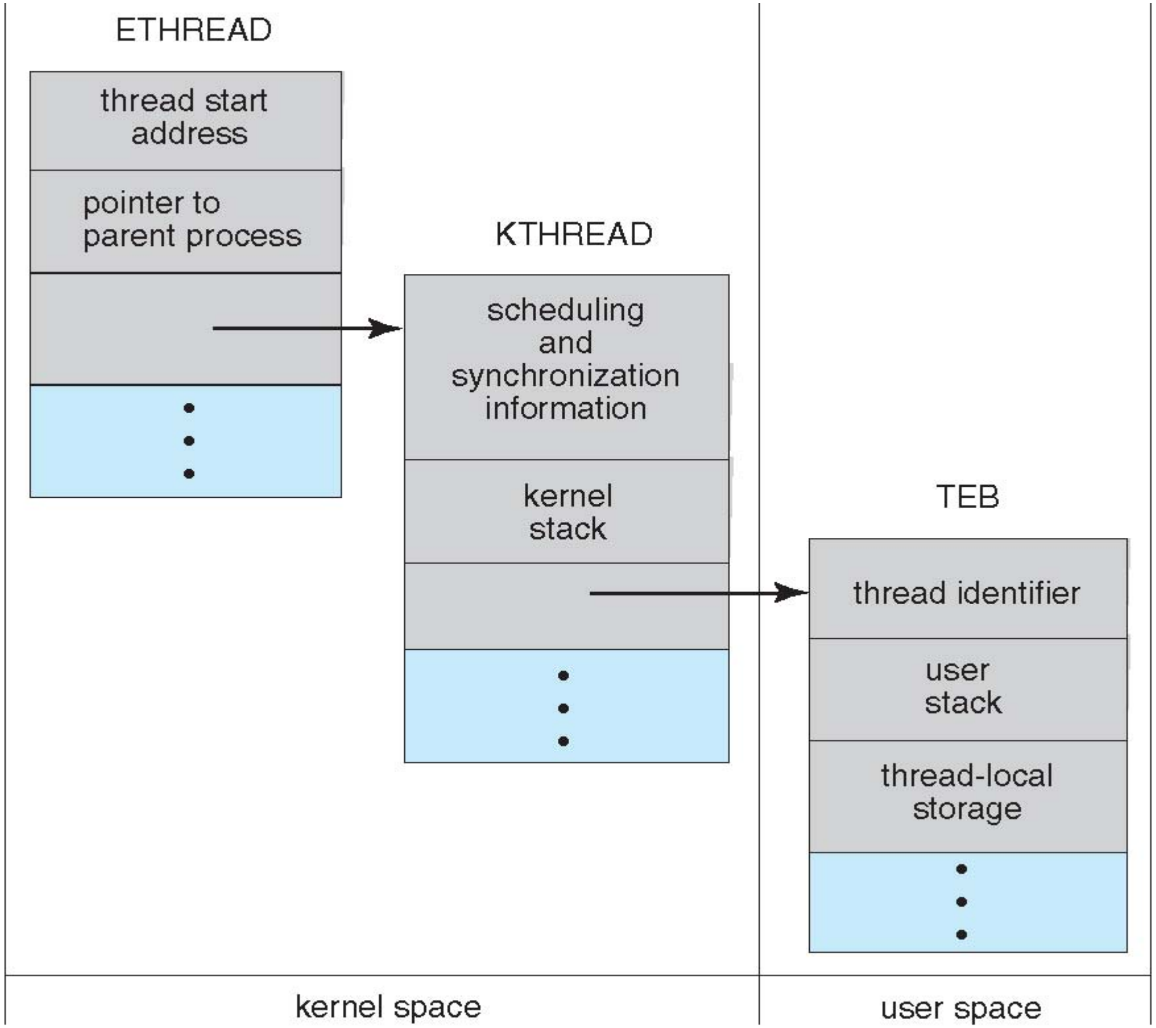
Thread Specific Data

- 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)

Operating System Examples

- Windows XP Threads
- Linux Thread

Windows XP Threads Data Structures



Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- 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)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

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)
- `struct task_struct` points to process data structures (shared or unique)

Linux Threads

- Doesn't distinguish between process and thread
  - Uses term *task* rather than thread
- `clone()` takes options to determine sharing on process create
- `struct task_struct` points to process data structures (shared or unique)

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.