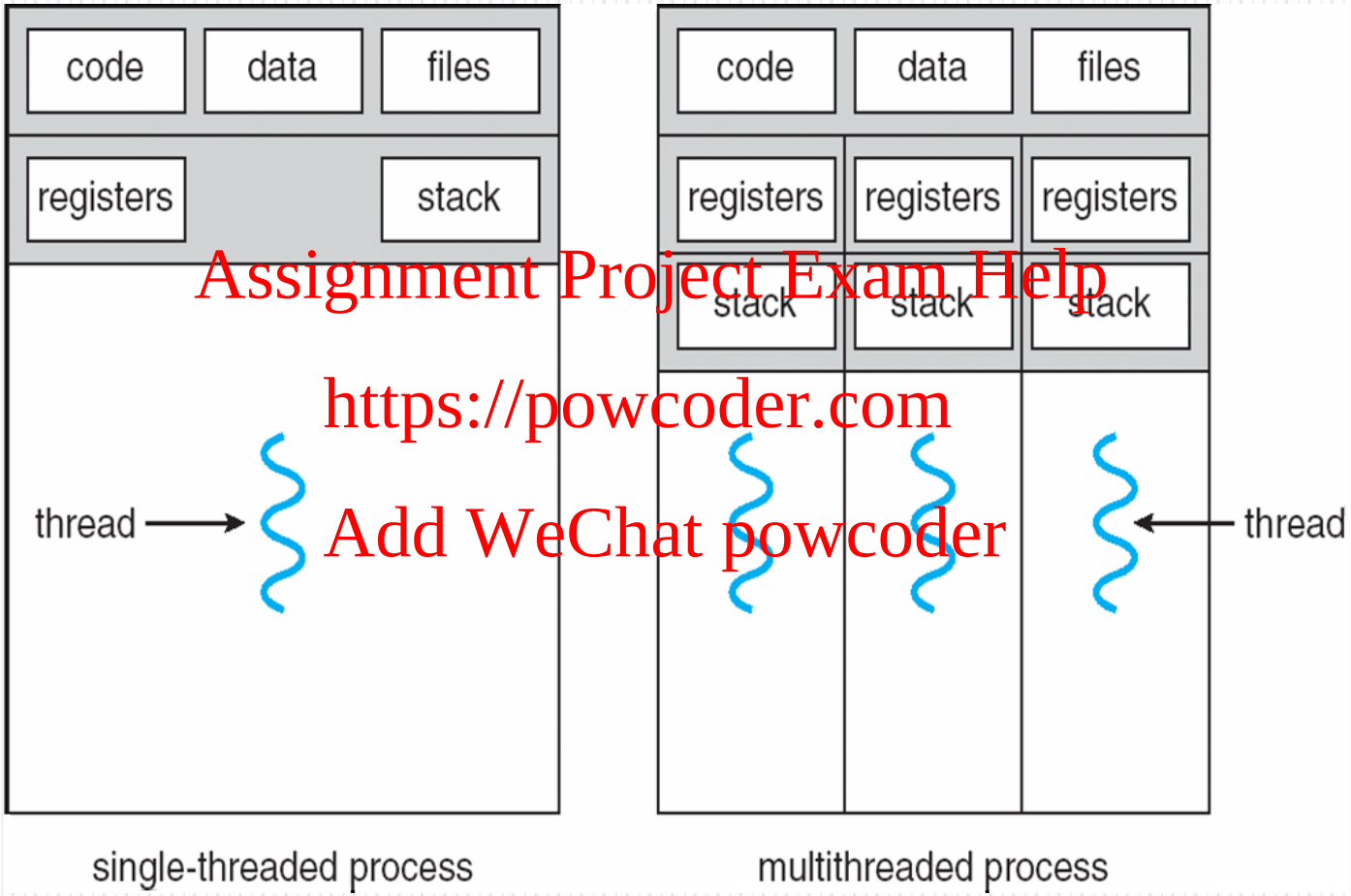
Chapter 4: Threads & Concurrency

Threads

- A thread is a basic unit of CPU utilization; it consists of:
  - its own copy of registers and stack space
  - its own program counter
  - a thread ID
- Threads within a same process or *task* execute in the same address space. A thread shares with its peer threads:
  - code section
  - data section
  - operating system resources, e.g. open files.

A traditional or heavyweight process is equal to a task with one thread.

Single and Multithreaded Processes



User Level Threads

Threads can be implemented at *user* or *kernel* level

- User level threads are implemented in user-level libraries, rather than via system calls, so thread switching does not need to call the operating system and generate an interrupt; kernel has no knowledge of user-level threads

User level threads:

- Advantage: no kernel involvement in switching; fast
- Disadvantage: since the kernel is not aware of user-level threads, if one user level thread blocks when making a *system call*, all user level threads in the same task may block too

Benefits of Threads

- One multi-threaded process uses fewer resources than multiple redundant processes, including memory, open files and CPU scheduling
  - Low overhead for creation
  - Low overhead for context switch from one thread to another peer thread
- Since threads within the same process share memory, sharing of information between peer threads is easier (but may require the use of synchronization)

Benefits of Threads (Cont.)

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
  - Cooperation of multiple threads in same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.
- Kernel level threads provide a mechanism that allows sequential processes to make *blocking system calls* while also achieving parallelism.

Potential Pitfalls

Potential pitfalls in using threads

- Because peer threads execute in the same address space, synchronization is often needed to protect shared data against simultaneous access

General Benefits of Threads

- Responsiveness

  - multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy application.

- Resource Sharing

  - threads share code and data within the same address space.

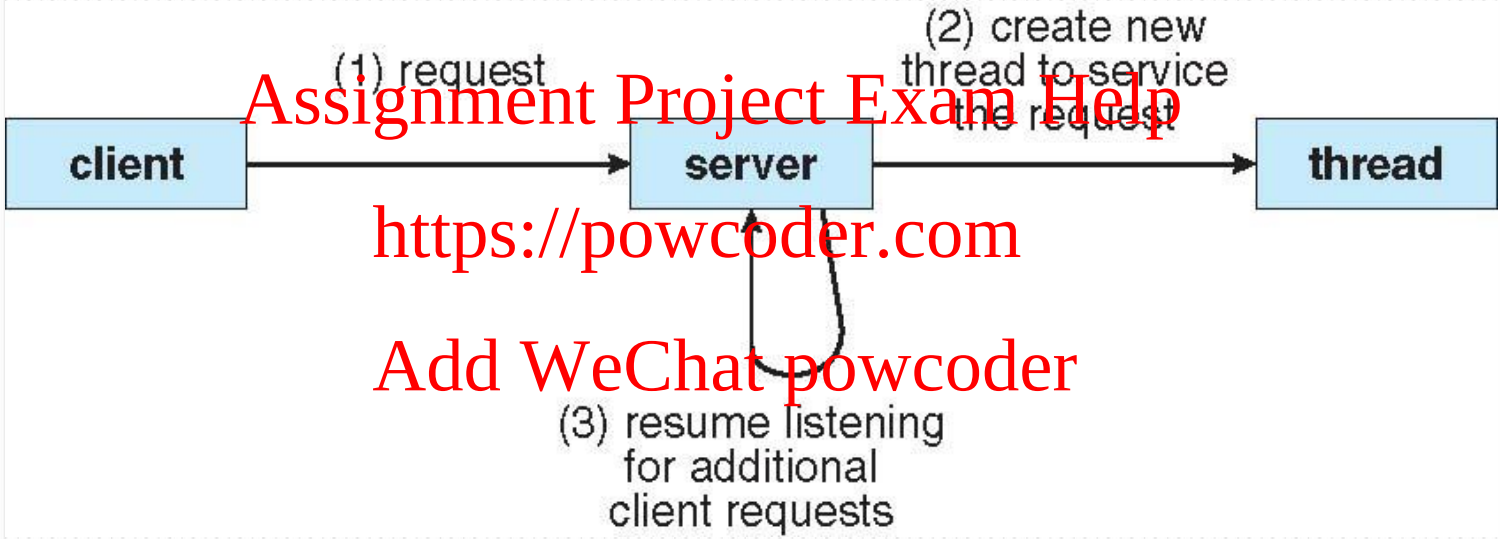
- Economy

  - it is less costly to create and context switch threads than processes.

- Scalability

  - multithreading on a multi-CPU machine increases parallelism.

Multithreaded Server Architecture



Multicore Programming Issues

- **Dividing activities**

  - how to divide an application into separate tasks that can be run in parallel on individual cores.

- **Balance**

  - need to ensure that the tasks that run in parallel have the same work load.

- **Data splitting**

  - need to divide the data that is accessed and manipulated by the tasks that run in parallel.

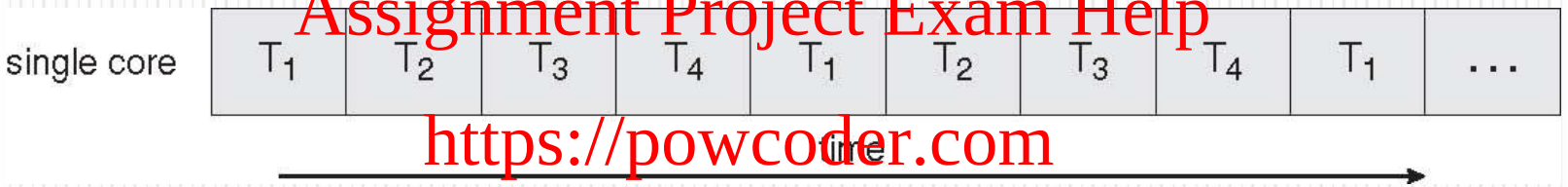
- **Data dependency**

  - when one task depends on data produced by another task, the execution of the two tasks must be synchronized.

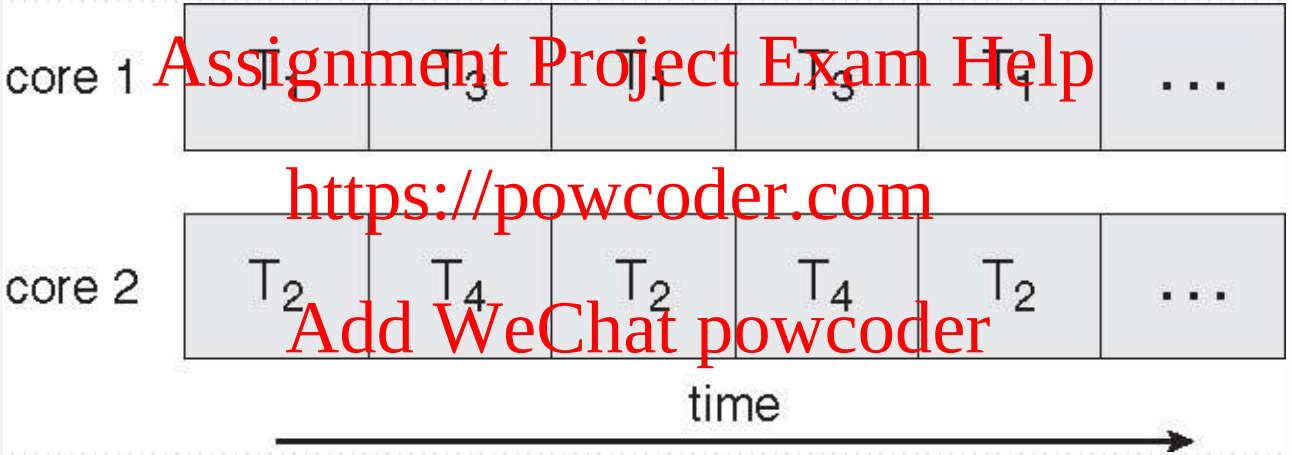
- **Testing and debugging**

  - when program runs in parallel, many different possible execution paths.

Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



Multithreading Models

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

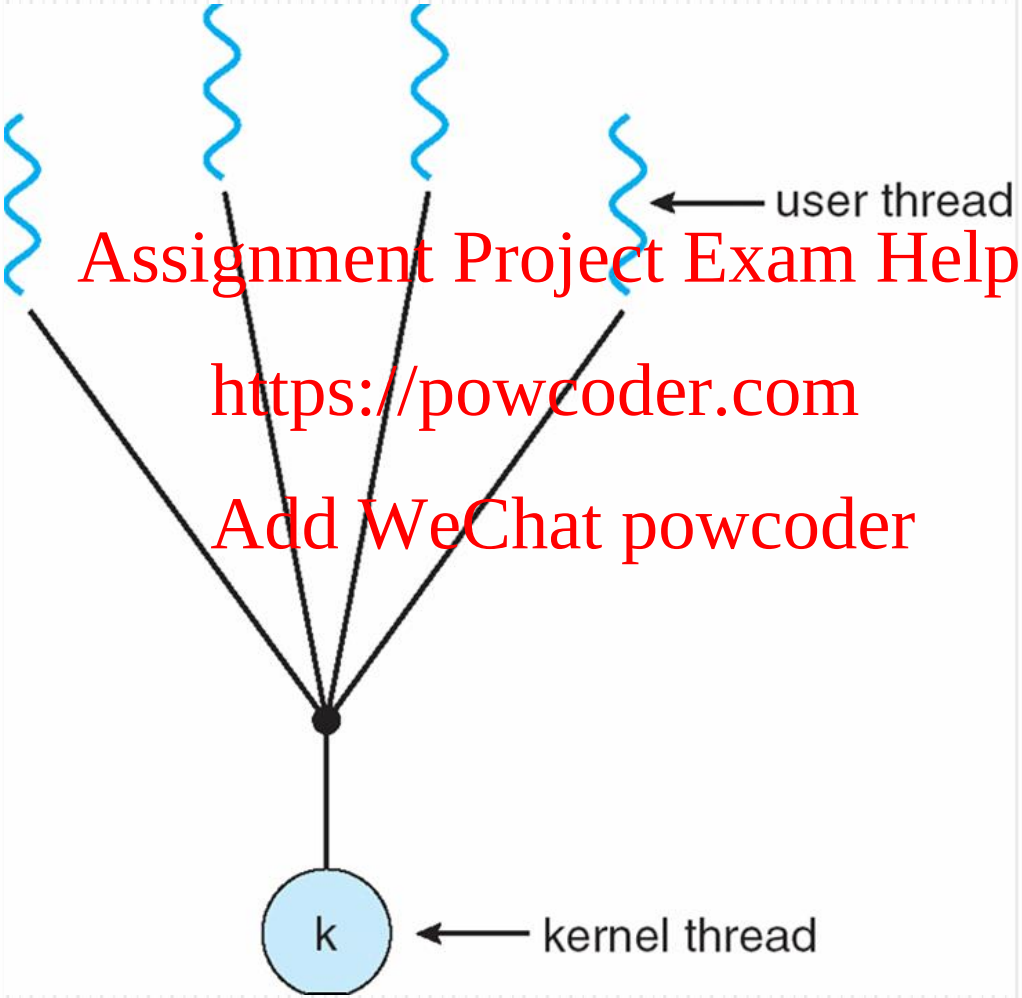
Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
- Advantage: thread management done by thread library in user space, so it is efficient.
- Disadvantages:

  (1) Entire process will block if one thread makes a blocking system call.

  (2) Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- Used on systems that do not support kernel 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
- Advantages

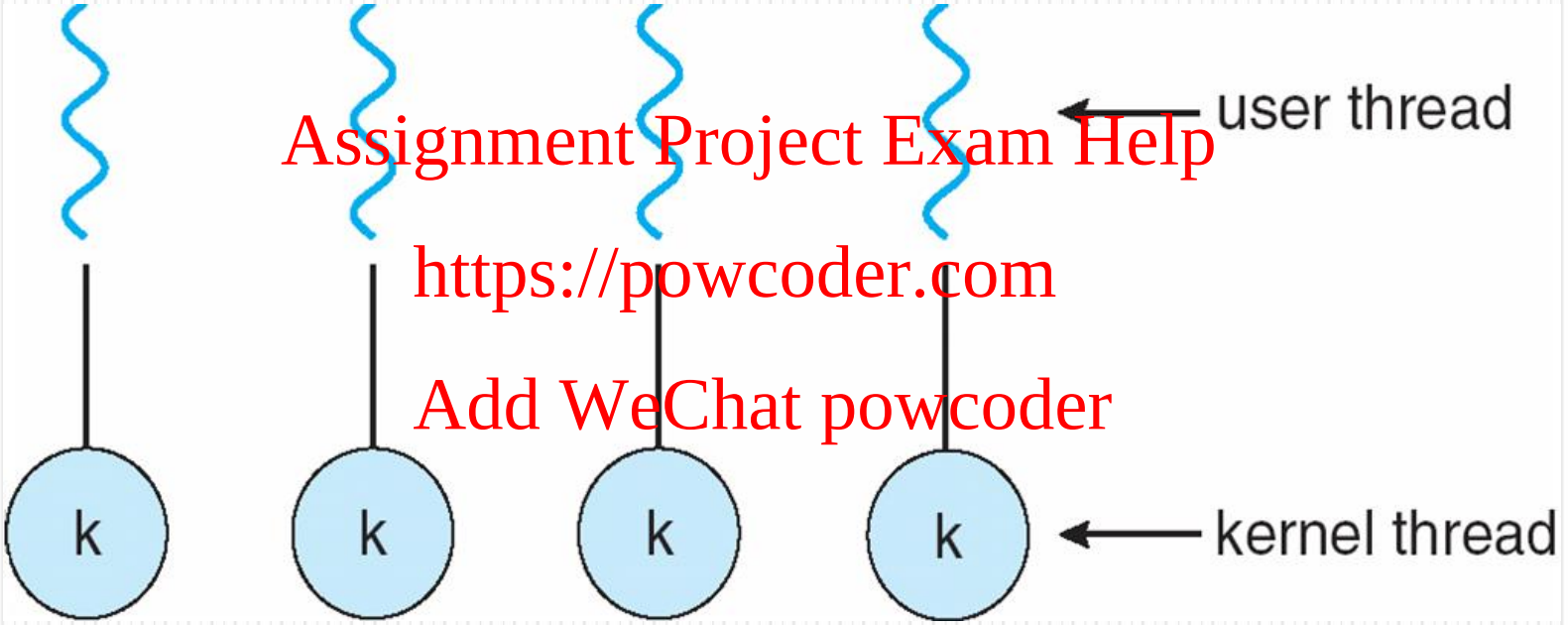
  (1) More concurrency – allows another thread to run when a thread makes a blocking system call.

  (2) Allows multiple threads to run in parallel on multiprocessors.
- Disadvantage:

  (1) Overhead: creating each user thread requires creating a kernel thread.

  (2) Need to restrict the number of threads supported by the system.

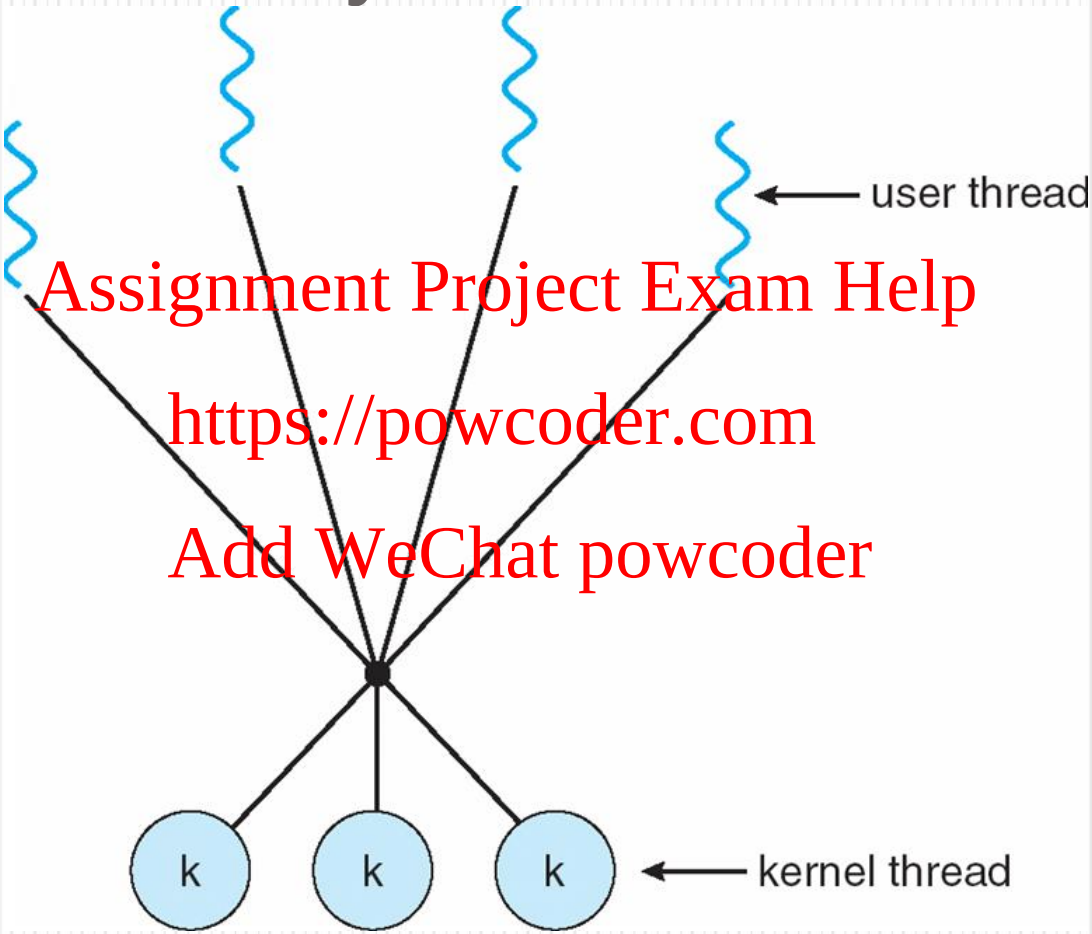
One-to-one Model



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
- Compromise between the previous models.
- More complex to implement.

Many-to-Many 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

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)

Pthread C Program

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

#include <stdlib.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 */

   /* set the default attributes of the thread */
   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 execute 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 4.11 Multithreaded C program using the Pthreads API.

POSIX pthread Mutexes

- Before accessing shared data, call *pthread_mutex_lock(&mutex)*
  - if another thread has locked the mutex, the caller will be blocked until the other thread unlocks the mutex.
- After accessing shared data, call *pthread_mutex_unlock(&mutex)*
- For nonblocking mutex locks, use *pthread_mutex_trylock(&mutex)*
  – this will return an error status (EBUSY) instead of blocking if the mutex is already blocked
- To prevent simultaneous access to shared data

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)

End of Chapter 4