

Chapter 4:

Multithreaded Programming

Chien Chin Chen

Department of Information Management
National Taiwan University

Outline

- Overview.
- Multithreading Models.
- Thread Libraries.
- Threading Issues.

Overview (1/6)

- *Single-threaded* process – *multithreaded* process:
 - A thread is a basic unit of CPU utilization.
 - Traditional process has a single thread of control.
 - Multithreaded process can perform **more than one task at a time**.
- Many applications are multithreaded.
 - A web browser might have one thread display images while another thread retrieves data from the network.
 - A word processor may have a thread for responding to keystrokes from the users, and another thread for performing spelling and grammar checking in the background.

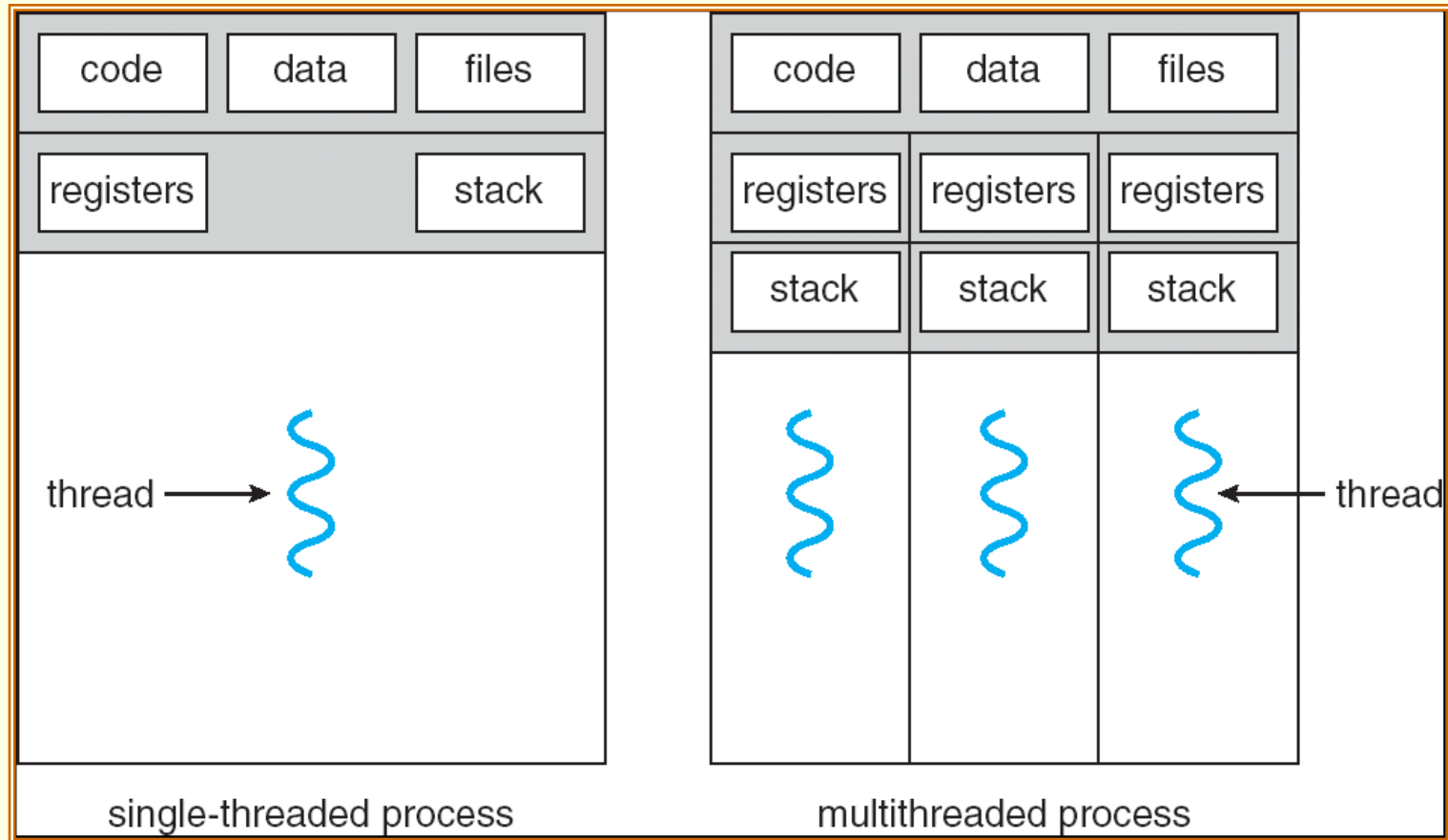
Overview (2/6)

- In many situations, an application may be required to perform several similar tasks.
 - A web server accepts several client requests for web pages.
 - If the web server ran as a traditional single-threaded process ... it would be able to service only one client at a time.
- Solution 1:
 - When the server receives a request, it **creates a separate process to service that request.**
 - E.g., `fork()`.
 - This process-creation method was in common use before threads became popular.
 - **Process creation is time consuming and resource intensive.**
- Solution 2:
 - It is more efficient to use one process that contains multiple threads.
 - When a request was made, the server (a thread) would **create another thread to service the request.**

Overview (3/6)

- A thread comprises:
 - A **thread ID**.
 - A **program counter**.
 - A **register set**.
 - A **stack**.
- It shares with other threads belonging to the same process:
 - Code section.
 - Data section.
 - Other operating-system resources, such as open files.

Overview (4/6)



Overview (5/6)

■ Benefits:

■ Responsiveness:

- Application can continue running even if part of it is blocked or is performing a lengthy operation.
- A multithreaded web browser allow user interaction in one thread while an image was being loaded in another thread.

■ Resource sharing:

- Threads share the memory and the resources of the process to which they belong. **Global data**

■ Economy:

- Allocating memory and resources for process creation is costly.
- In Solaris, creating a process is about thirty times slower than is creating a thread. Context switching is about five times slower.

■ Utilization of multiprocessor architectures:

- Threads may be running in parallel on different processors.
- A single-threaded process can only run on one CPU, no matter how many are available.

Overview (6/6)

- Many operating system kernels are now multithreaded.
 - Several threads operate in the kernel.
 - Each thread performs a specific task.
 - For example, Linux uses a kernel thread for managing the amount of free memory in the system.

Multithreading Models (1/8)

■ User threads – kernel threads:

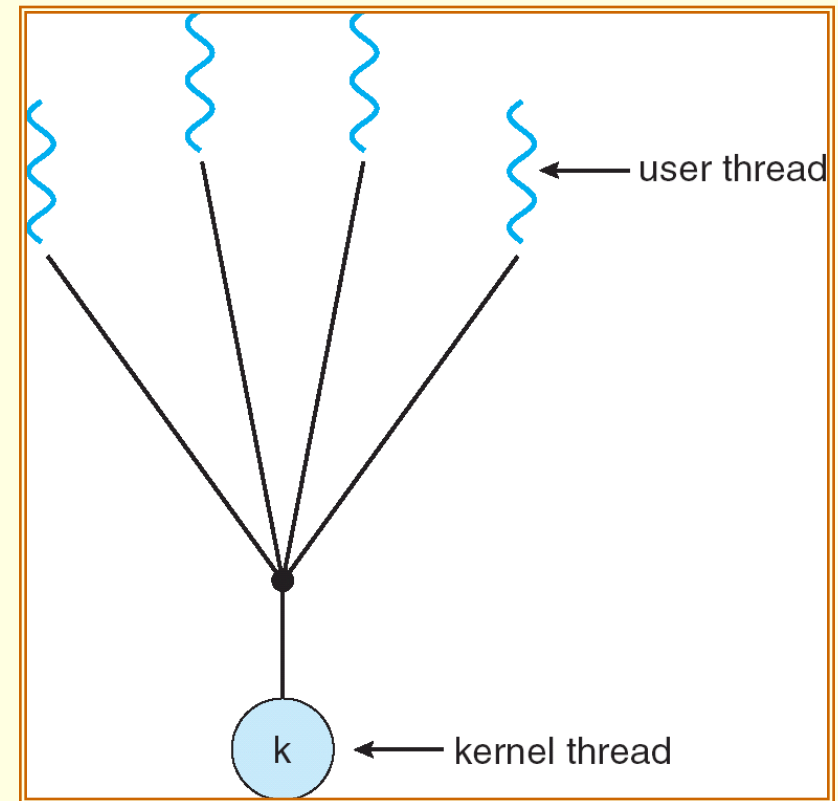
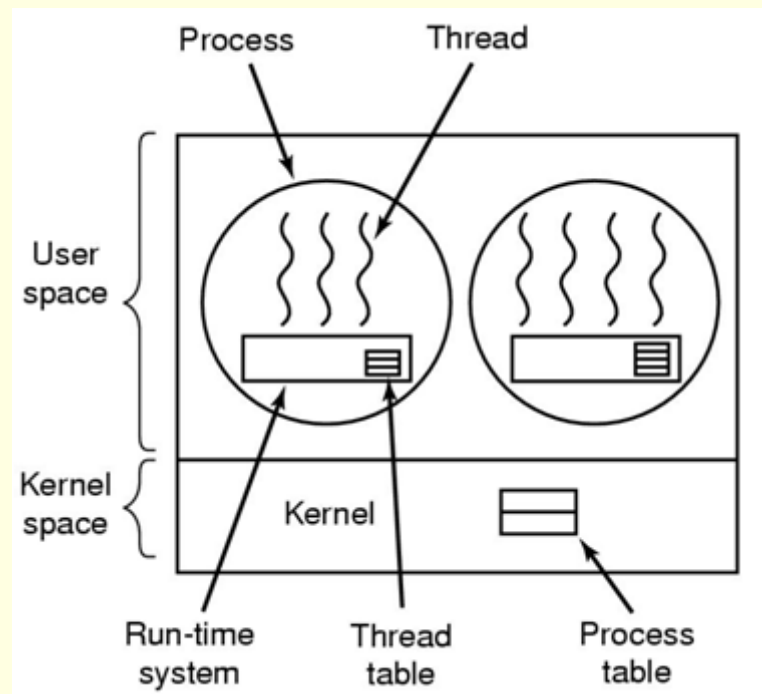
- *User threads* are supported above the kernel and are managed without kernel support.
- *Kernel threads* are supported and managed directly by the operating system.
- All contemporary operating systems support kernel threads.
- There must exist a relationship between user threads and kernel threads.

Multithreading Models (2/8)

■ *Many-to-One model:*

- Map many user-level threads to one kernel thread.
- Thread management is done by the thread library **in user space**.
 - The library provides the supports of thread creation, scheduling ...
- **Is efficient**, because there is no kernel intervention.
- But ... as the kernel is not aware of user threads ...
- The entire process will block if a thread makes a blocking system call.
- Multiple threads are unable to run in parallel on multiprocessors.

Multithreading Models (3/8)

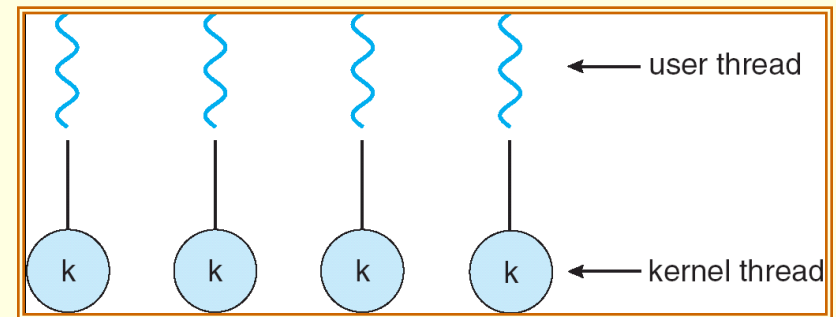
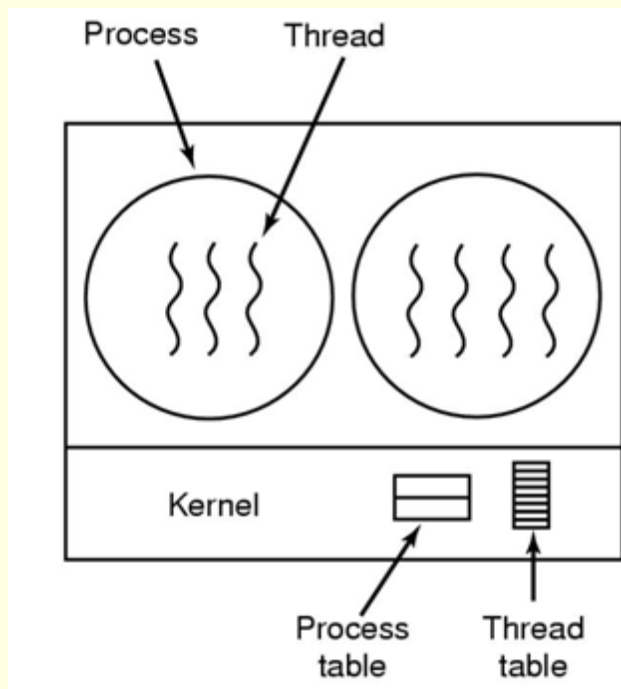


Many-to-One model

Multithreading Models (4/8)

- ***One-to-One model:***
 - Map each user thread to a kernel thread.
 - Allow another thread to run when a thread makes a blocking system call.
 - Allow multiple threads to run in parallel on multiprocessors.
 - But ... creating a user thread requires creating the corresponding kernel thread.
 - The creation overhead can burden the performance of an application.
 - Most implementations of this model restrict the number of threads supported by the system.
 - Supported by Linux and Windows family.

Multithreading Models (5/8)



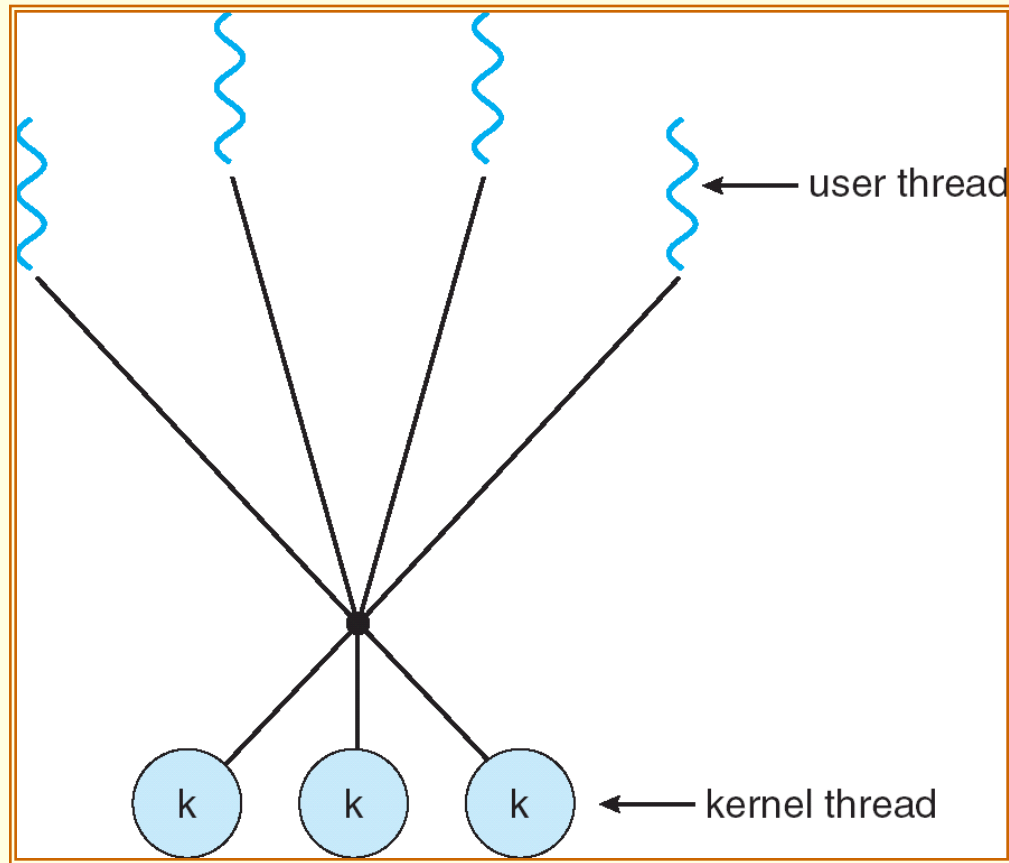
One-to-One model

Multithreading Models (6/8)

■ ***Many-to-Many model:***

- Multiplex many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine.
- The many-to-many model **does not** suffer from the shortcomings of the previous two models.
 - Many-to-One: the kernel can schedule only one thread at a time (on a multiprocessor system).
 - One-to-One: the number of threads is limited.
 - Many-to-many model ¹.allows as many user threads as necessary, and ².the corresponding kernel threads can run in parallel on a multiprocessor. ³.When a thread is blocked, the kernel can schedule another thread for execution.

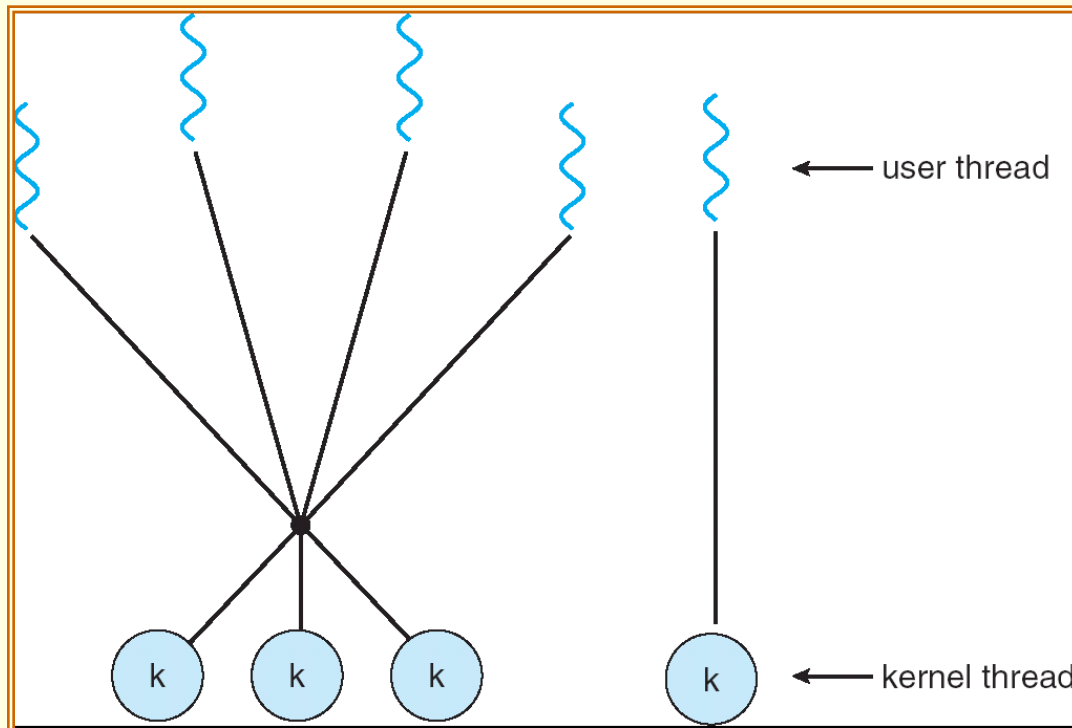
Multithreading Models (7/8)



Many-to-Many model

Multithreading Models (8/8)

- **Two-Level model** (hybrid thread model):
 - Similar to many-to-many model.
 - But allow a user-level thread to be bound to a kernel thread.




Two-Level model

Thread Libraries (1/11)

- A thread library provides the programmer an **API** for creating and managing threads.
- Two ways of implementing a thread library:
 - To provide a library entirely in **user space** with no kernel support.
 - All code and data structures for the library exist in user space.
 - All function calls result in local function calls in user space; and no system calls.
 - To implement a **kernel-level** library:
 - Supported by the operating system.
 - Code and data structures exist in kernel space.
 - A function call in the API for the library results in a system call to the kernel.

Thread Libraries (2/11)

- Three primary thread libraries:
 - **POSIX Pthreads.**
 - May be provided as either a user- or kernel-level library.
 - **Win32 threads.**
 - Kernel-level library, available on Windows systems.
 - **Java threads.**
 - JVM is running on top of a host operating system, the implementation depends on the host system.
 - On Windows systems, Java threads are implemented using the Win32 API;
 - UNIX-based systems often use Pthreads.
- The following multithreaded examples perform the summation of a non-negative integer in a separate thread.

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


A diagram showing a box labeled "User specified" with an arrow pointing to the superscript N in the summation formula $sum = \sum_{i=0}^N i$.

Thread Libraries – **Pthreads** (3/11)

- Pthreads refers to the POSIX standard, defining an API for thread creation and synchronization.
 - **Is a specification** for thread behavior, **not an implementation.**
 - Operating system designers implement the specification in any way they wish.
 - When a program begins, a single thread of control begins in `main()`.
 - `main()` then creates a second thread.
 - In a Pthreads program, separate threads begin execution in a specified function – in this example, the `runner()` function.

Thread Libraries – Pthreads (4/11)

```
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }

    pthread_exit(0);
}
```

A global variable shared with main()

Thread termination

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

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

All Pthreads programs must include this header file.

thrd-posix.c

```
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 attributes for the thread */
```

```
    if (argc != 2) {
```

```
        fprintf(stderr, "usage: a.out <integer value>\n");
```

```
        return -1;
```

```
    }
```

```
    if (atoi(argv[1]) < 0) {
```

```
        fprintf(stderr, "Argument %d must be non-negative\n", atoi(argv[1]));
```

```
        return -1;
```

```
    }
```

```
    /* get the default attributes */
```

```
    pthread_attr_init(&attr);
```

Use default thread attributes.

```
    /* create the thread */
```

```
    pthread_create(&tid, &attr, runner, argv[1]);
```

Create a separate thread

```
    /* now wait for the thread to exit */
```

```
    pthread_join(tid, NULL);
```

The parent thread waits for child to complete.

```
    printf("sum = %d\n", sum);
```

```
}
```

Thread Libraries – **Win32** (6/11)

- A Win32 multithreaded operation will usually be embedded inside a function which returns a `DWORD` and takes a `LPVOID` as a parameter.
 - The `DWORD` data type is an unsigned 32-bit integer.
 - `LPVOID` is a pointer to a `void`.
- Data shared by the separate threads are declared globally.

Thread Libraries – Win32 (7/11)

```
/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD *)Param;

    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;

    return 0;
}
```

Unsigned 32-bit
integer

Is identical to `void *`

A global variable, shared by the thread(s)

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

```
#include <windows.h>
```

Must include this header file

thrd-win32.c

```
DWORD Sum; /* data is shared by the thread(s) */
```

```
int main(int argc, char *argv[])  
{
```

```
    DWORD ThreadId;
```

```
    HANDLE ThreadHandle;
```

```
    int Param; void pointer
```

A windows system resources, such as file and thread, are represented as **kernel object**. Objects are accessed in Windows programs by **handles**

```
    ... // do some basic error checking
```

```
    Param = atoi(argv[1]);
```

```
    if (Param < 0) {
```

```
        fprintf(stderr, "an integer >= 0 is required \n");
```

```
        return -1;
```

```
    }
```

Default security attributes

Default stack size

Creation flags, default values make it eligible to be run by the CPU scheduler

```
    // create the thread
```

```
    ThreadHandle = CreateThread(NULL, 0, Summation, &Param, 0, &ThreadId);
```

Thread function

Parameter to thread function

```
    if (ThreadHandle != NULL) {
```

```
        WaitForSingleObject(ThreadHandle, INFINITE);
```

```
        CloseHandle(ThreadHandle);
```

```
        printf("sum = %d\n", Sum);
```

```
    }
```

```
}
```


Thread Libraries – **Java** (9/11)

- All Java programs comprise at least a single thread of control.
 - E.g., the `main()` method runs as a single thread in the JVM.
- To create Java threads,
 - **First** define a class that implements the `Runnable` interface.
 - The interface has a function `run()` that the class must implement.

```
class Sum
{
    private int sum;

    public int get() {
        return sum; }

    public void set(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.set(sum);
    }
}
```

Thread Libraries – Java (10/11)

- **Then**, create an object instance of the `Thread` class and passing the constructor a `Runnable` object.
- Creating a `Thread` object **does not** create the new thread; it is the `start()` method that actually creates the new thread.
- The `join()` method in Java provides similar functionality to the `pthread_join()` and `WaitForSingleObject()`.

```
public class Driver
{
    public static void main(String[] args) {
        ... // do some basic error checking
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);

        Thread worker = new Thread(new Summation(upper, sumObject));
        worker.start();
        worker.join();
        System.out.println("The sum of " + upper + " is " + sumObject.get());
    }
}
```

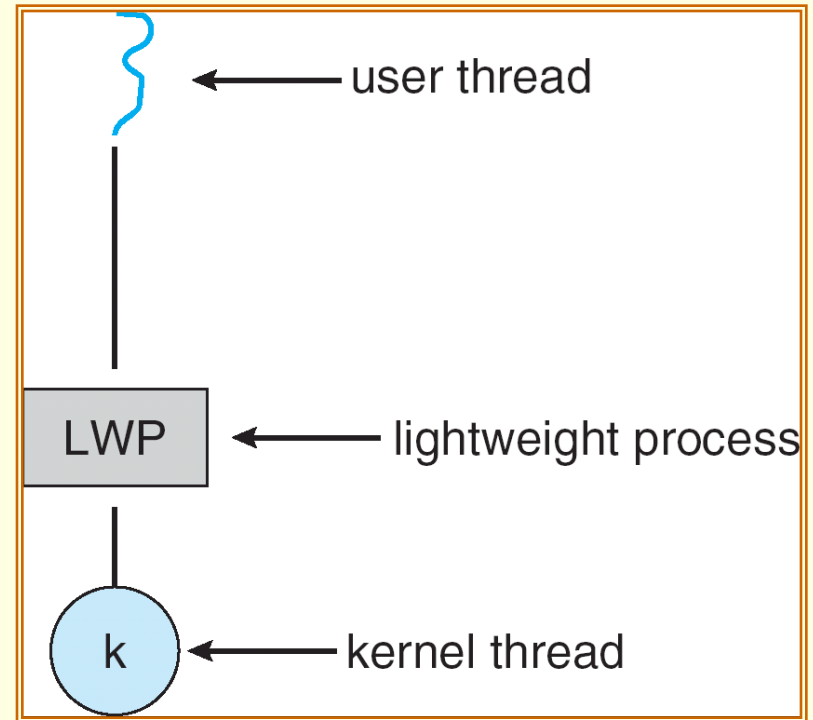
Driver.java

Thread Libraries – Java (11/11)

- Calling the `start()` method does two things:
 - It allocates memory and initializes a new (child) thread in the JVM.
 - The child thread begins execution in the `run()` method.

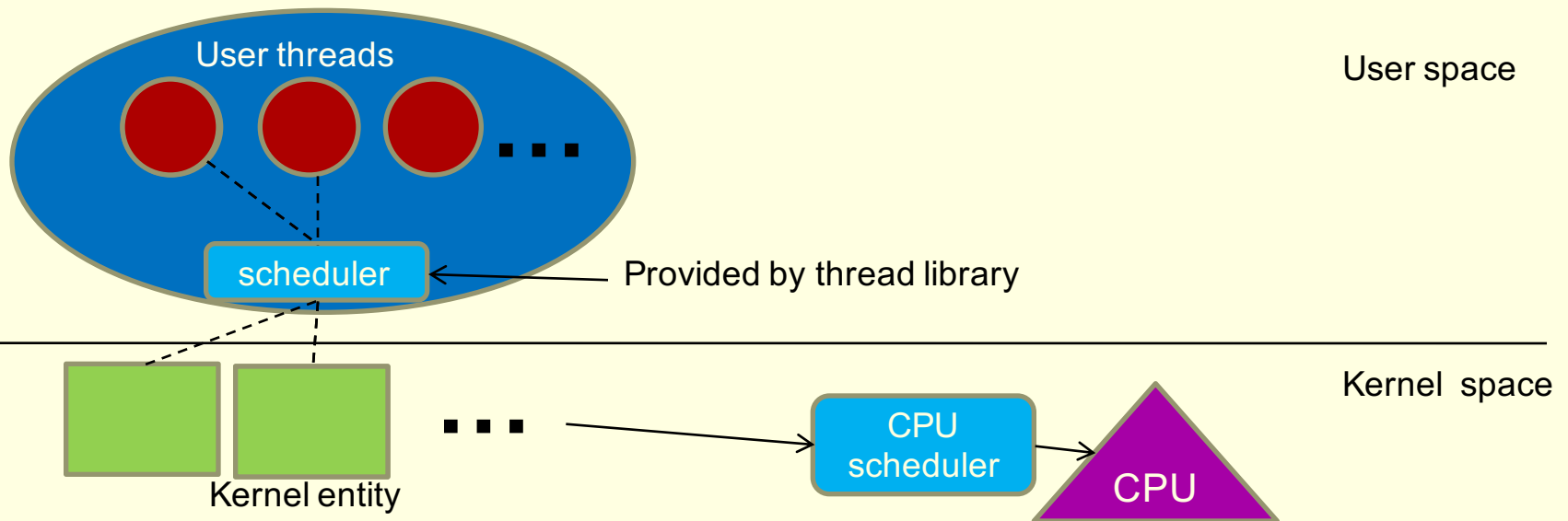
Threading Issues – Scheduler Activations (1/5)

- Many systems implementing the *many-to-many model* place an **intermediate data structure** between the user and kernel threads.
 - **Lightweight process** (or LWP, virtual processor).
- The kernel provides an application with a set of virtual processors.
- Each LWP is attached to a kernel thread, and is scheduled by the operating system to run on physical processors.
- To the user-thread library, a LWP can be used to schedule a user thread (to run).



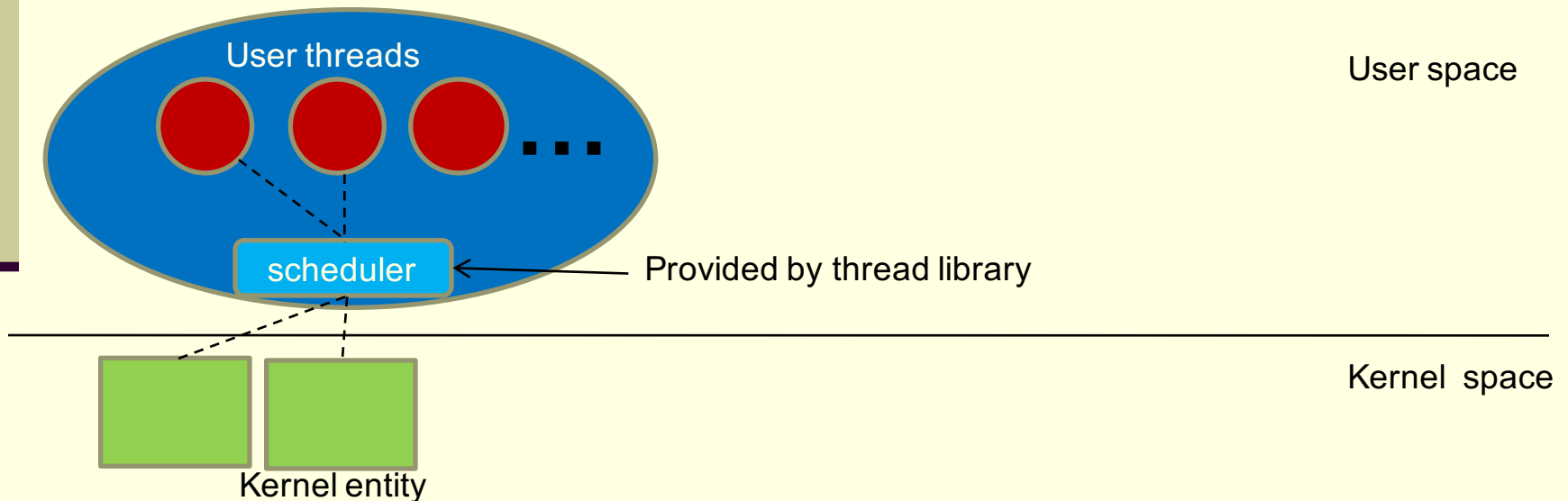
Threading Issues – Scheduler Activations (2/5)

- A Problem of many-to-many model.
 - A thread will block a kernel entity if it makes a blocking system call.



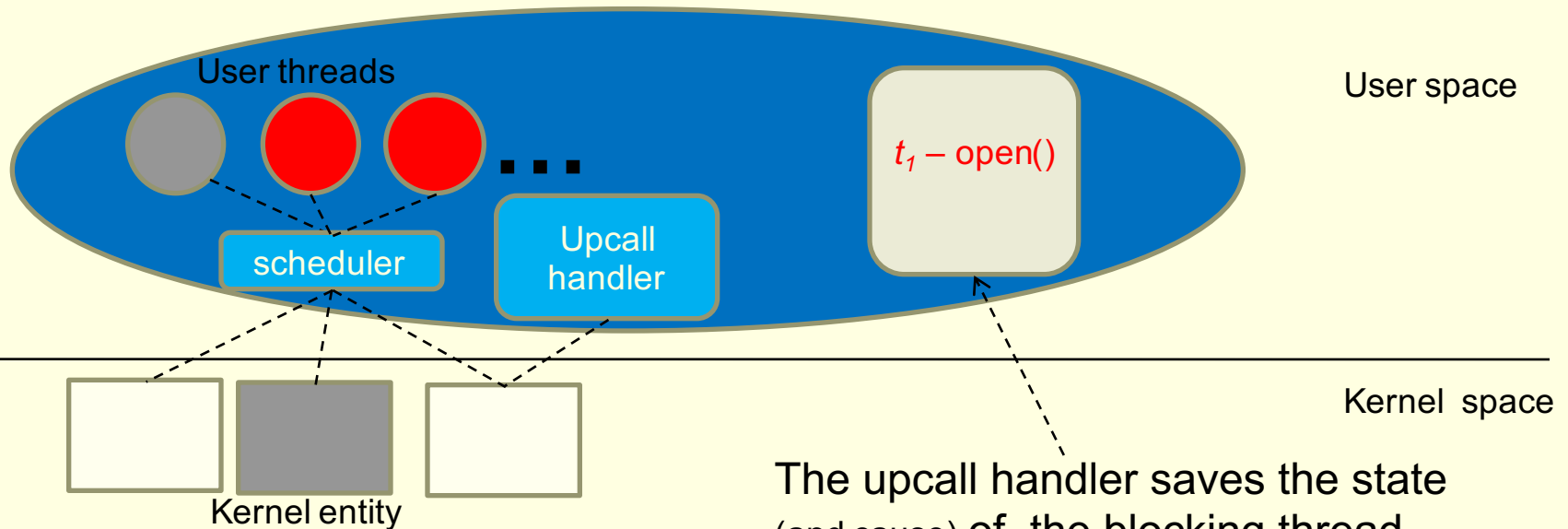
Threading Issues – Scheduler Activations (3/5)

- **Scheduler activation** – an efficient scheme for running many-to-many model.
 - When an user thread is about to block ... kernel **activates** the **scheduler** supplied by thread library to schedule another user thread.



Threading Issues – Scheduler Activations (4/5)

- How ???
- The kernel makes an **upcall** to **inform** an application about certain events.
 - Upcalls are handled by the thread library with an **upcall handler**.



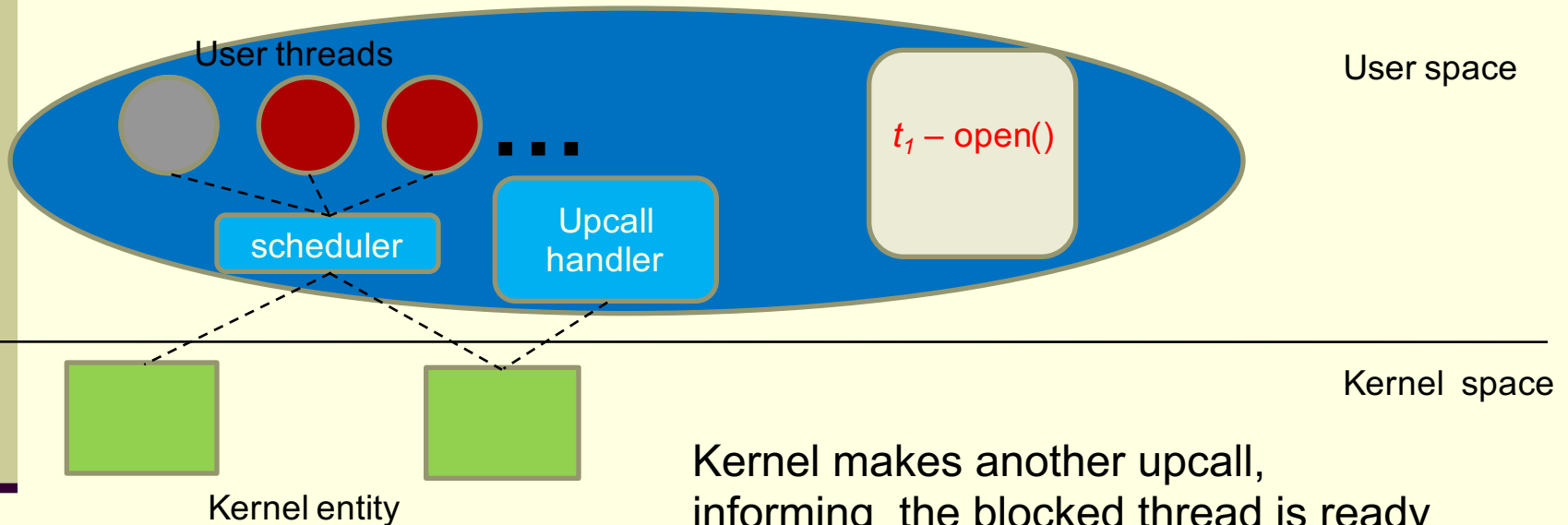
Kernel first allocates a new virtual process (or kernel entity) for executing the upcall handler.

The upcall handler saves the state (and cause) of the blocking thread.

Then it release the blocking kernel entity and **activate** the **scheduler** to run a ready thread

Threading Issues – Scheduler Activations (5/5)

- When the waiting event completes ...



Kernel makes another upcall,
informing the blocked thread is ready

It may preempt one users thread (or allocate
a new virtual processor) to run the upcall handler.

Then **active** the **scheduler** again to
schedule a ready thread.

End of Chapter 4

Homework 2:

Exercise:

Due date:

Reference: **Programming with POSIX(R) Threads (Addison-Wesley Professional Computing Series)**

by David R. Butenhof... **400 pages**