

# Part II

# Processes and Threads

## Threads Basics

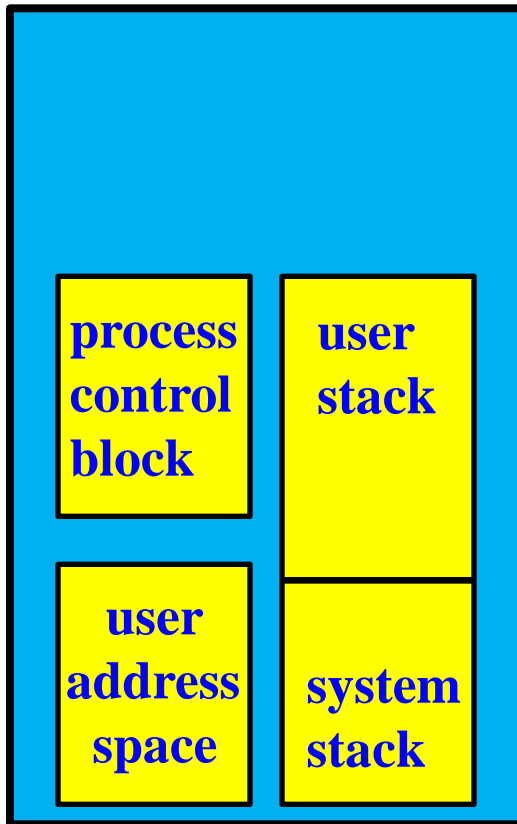
*You think you know when you learn, are more sure when you can write,  
even more when you can teach, but certain when you can program.*

# ***What Is a Thread?***

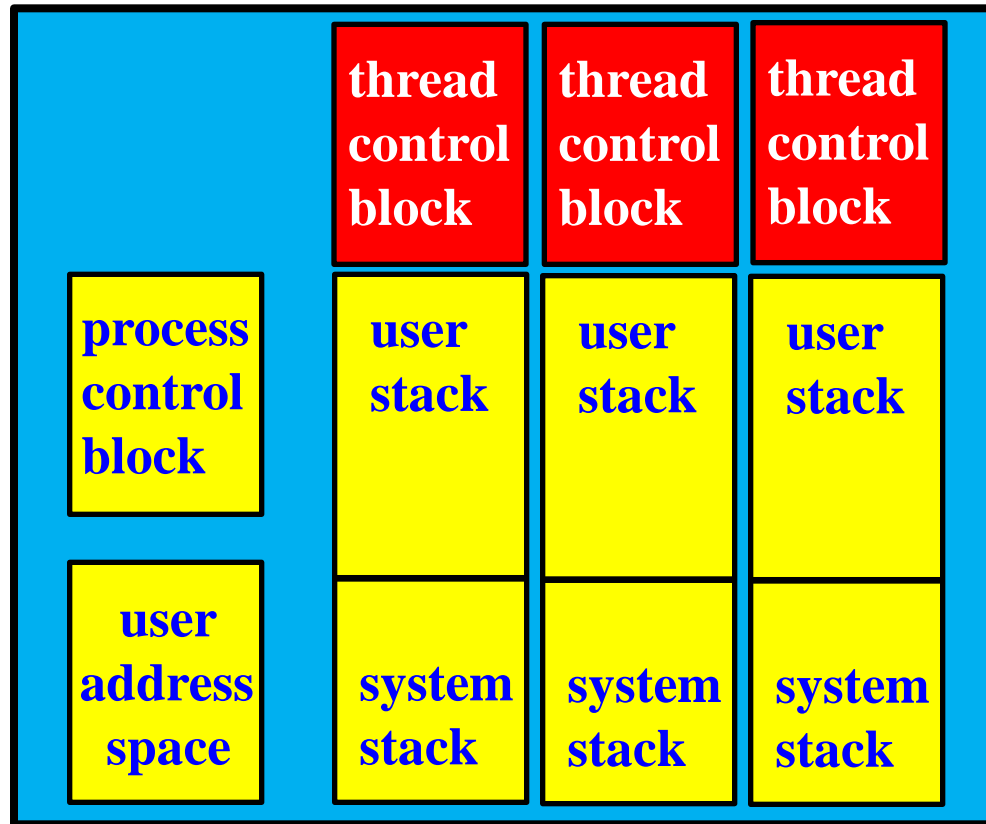
- A ***thread***, also known as ***lightweight process*** (LWP), is a basic unit of CPU execution, and is created by a process.
- A thread has a **thread ID**, a **program counter**, a **register set**, and a **stack**. Thus, it is similar to a process.
- However, a thread ***shares*** with other threads in the ***same*** process its code section, data section, and other OS resources (e.g., files and signals).
- A process, or **heavyweight** process, has a ***single*** thread of control.

# ***Single Threaded and Multithreaded Process***

Single-threaded process



Multithreaded process



# ***Benefits of Using Threads***

- **Responsiveness:** Other part (i.e., threads) of a program may still be running even if one part (e.g., a thread) is blocked.
- **Resource Sharing:** Threads of a process, by default, share many system resources (e.g., files and memory).
- **Economy:** Creating and terminating processes, allocating memory and resources, and context switching processes are very time consuming.
- **Utilization of Multiprocessor Architecture:** Multiple CPUs may run multiple threads of the same process. No program change is necessary.

# ***User and Kernel Threads: 1/3***

## **■ User Threads:**

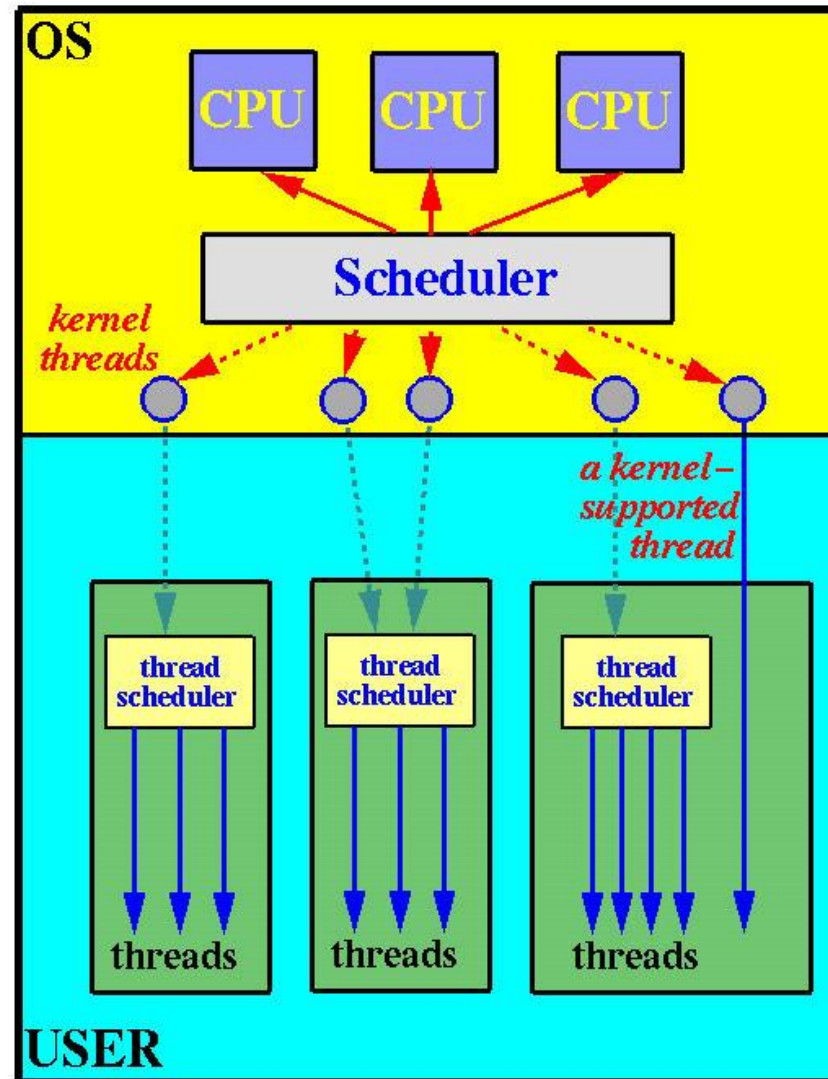
- ❖ User threads are supported at the **user level**. The kernel **is not aware** of user threads.
- ❖ A library provides all support for thread creation, termination, joining, and scheduling.
- ❖ Since there is no kernel intervention, user threads are usually **more efficient**.
- ❖ Unfortunately, since the kernel only recognizes the containing process (of the threads), ***if one thread is blocked, all threads of the same process are also blocked*** because the containing process is blocked.

# ***User and Kernel Threads: 2/3***

## ■ ***Kernel threads:***

- ❖ Kernel threads are supported by the kernel. The kernel does thread creation, termination, joining, and scheduling in kernel space.
- ❖ Kernel threads are usually **slower** than user threads due to system overhead.
- ❖ However, ***blocking one thread will not cause other threads of the same process to block.*** The kernel simply runs other kernel threads.
- ❖ In a multiprocessor environment, the kernel may run threads on different processors.

# User and Kernel Threads: 3/3

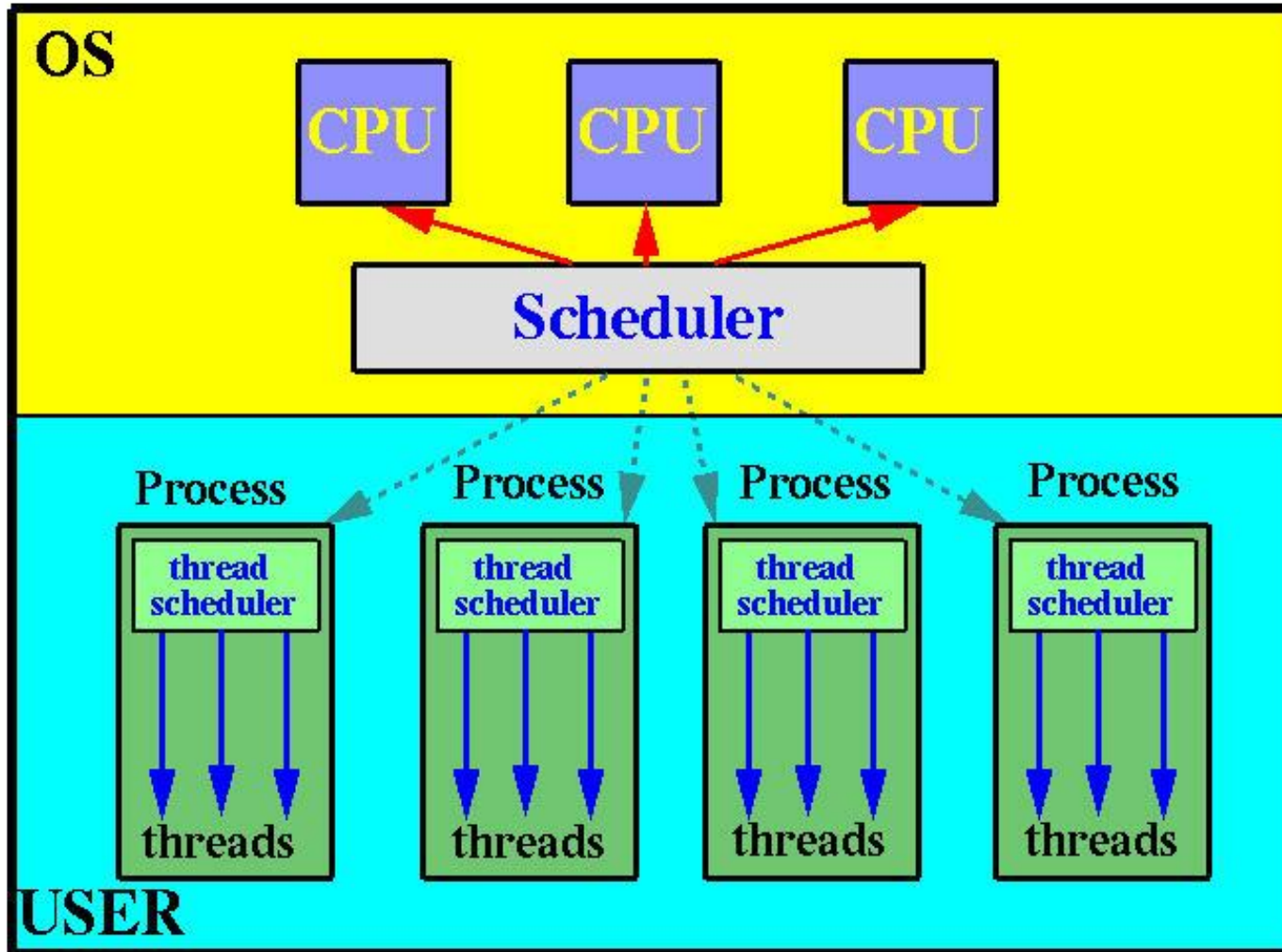


# ***Multithreading Models***

- Different systems support threads in different ways. Here are three commonly seen thread models:
  - ❖ ***Many-to-One Model***: One kernel thread (or process) has multiple user threads. Thus, this is a user thread model.
  - ❖ ***One-to-One Model***: One user thread maps to one kernel thread (e.g., old Unix/Linux and Windows systems).
  - ❖ ***Many-to-Many Model***: Multiple user threads map to a number of kernel threads.



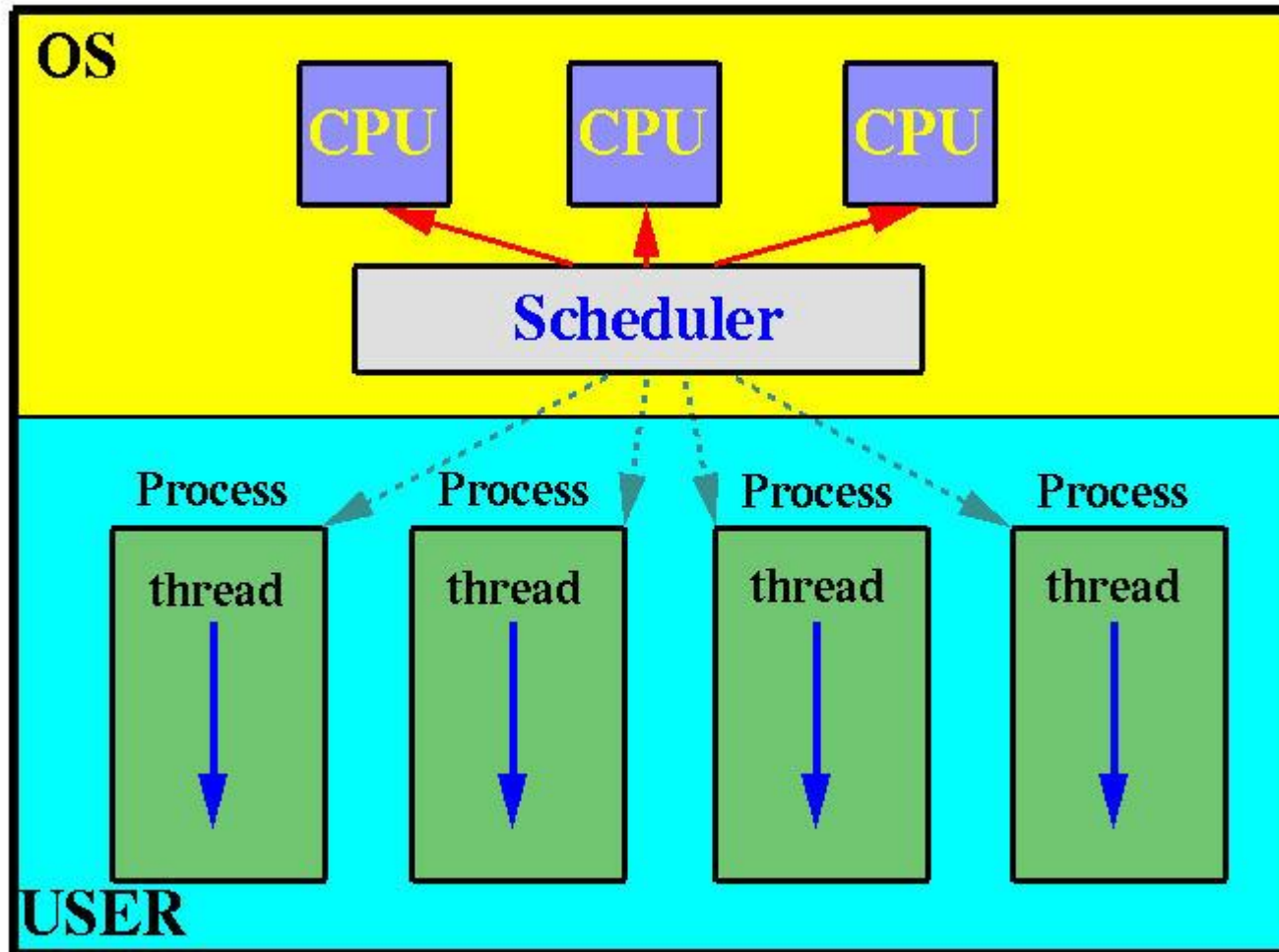
# Many-to-One Model



***Each process has multiple threads that are associated with one kernel threads***

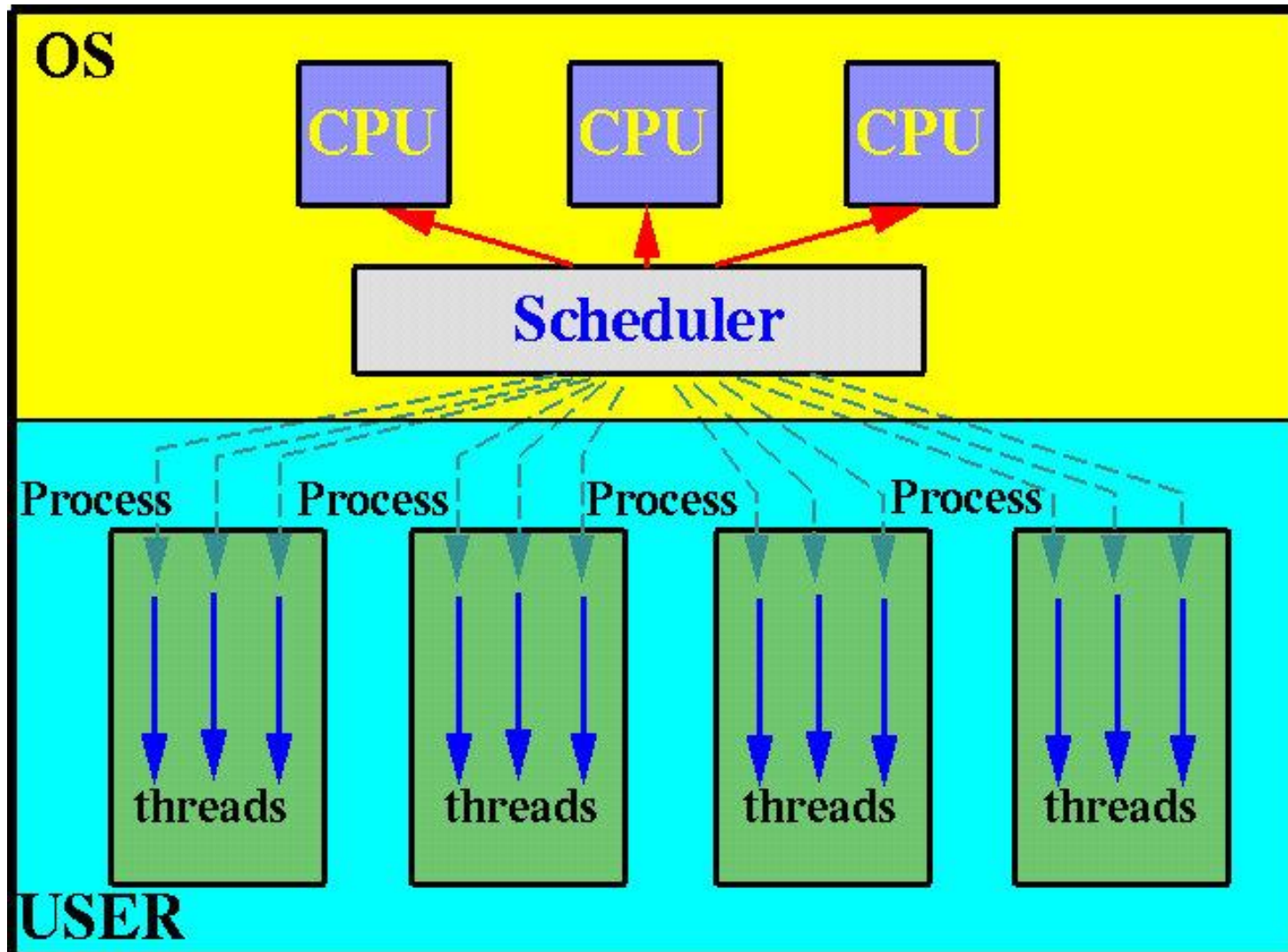
# ***One-to-One Model: 1/2***

## ***An Extreme Case: Traditional Unix***



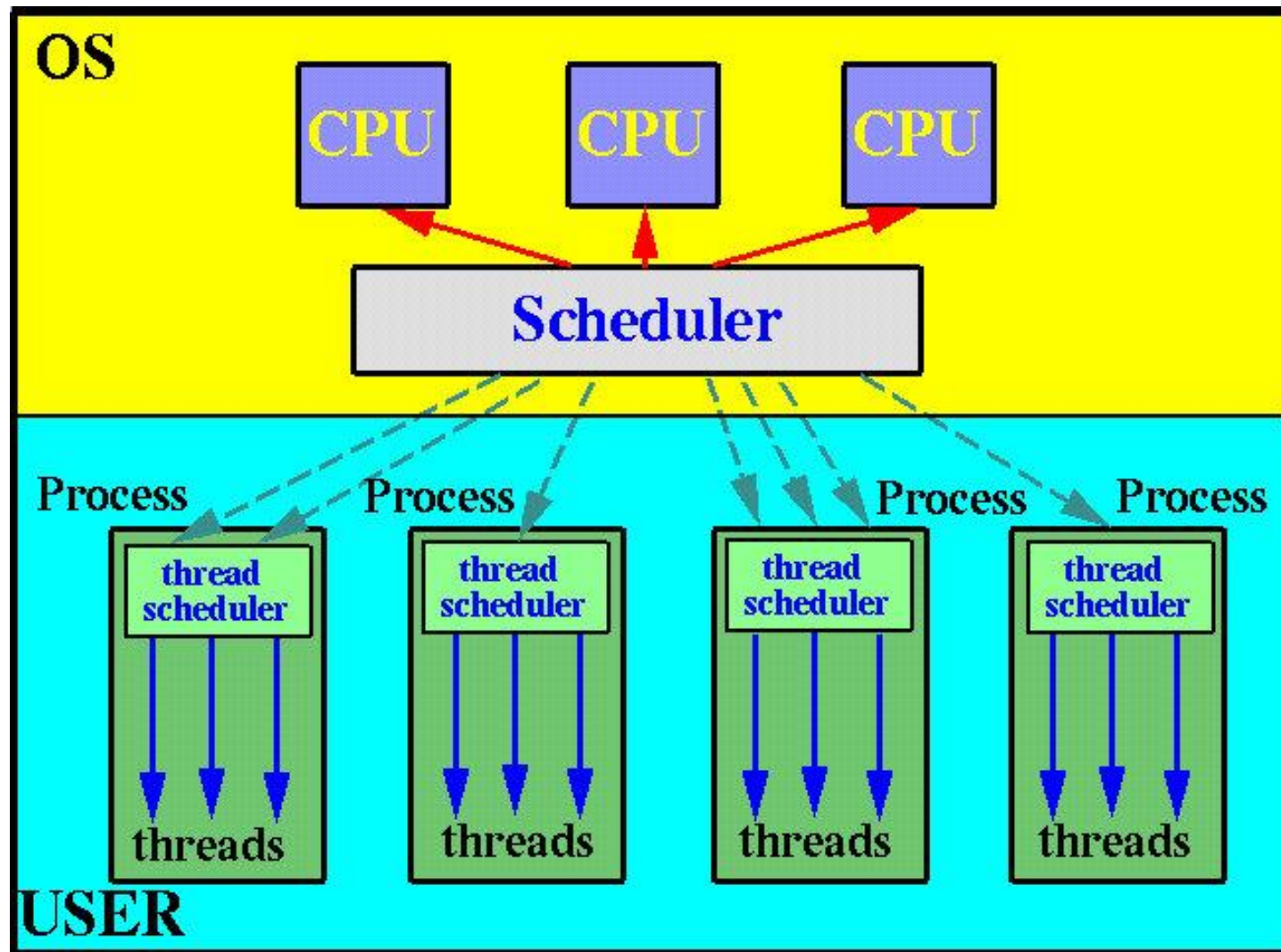
***Each process has only one thread that is associated with exactly one kernel thread***

# One-to-One Model: 2/2



***Each process has multiple threads each of which is associated with one kernel thread***

# ***Many-to-Many Model***



***Each process has multiple threads that are associated with multiple kernel threads***

# ***Multicore Programming: 1/6***

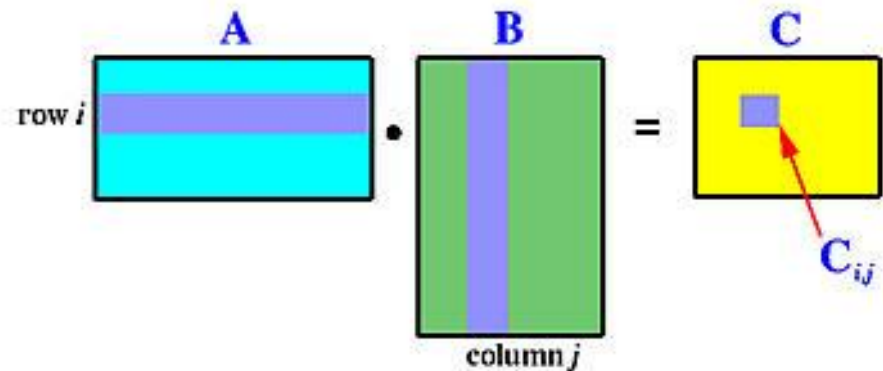
- With a single-core CPU, threads are scheduled by a scheduler and can only run one at a time.
- With a multicore CPU, multiple threads may run at the same time, one on each core.
- Therefore, system design becomes more complex than one may expect.
- Five issues have to be addressed properly: **dividing activities, balance, data splitting, data dependency, and testing and debugging.**

# Multicore Programming: 2/6

- **Dividing Activities:**

Since each thread can run on a core, one must study the problem in hand so that program activities can be divided and run concurrently.

- Matrix multiplication is a good example.
- Unfortunately, some problems are *inherently sequential* (e.g., DFS).



$$C_{i,j} = \sum_{k=1}^n A_{i,k} \times B_{k,j}$$

We may create a thread for each  $C_{ij}$

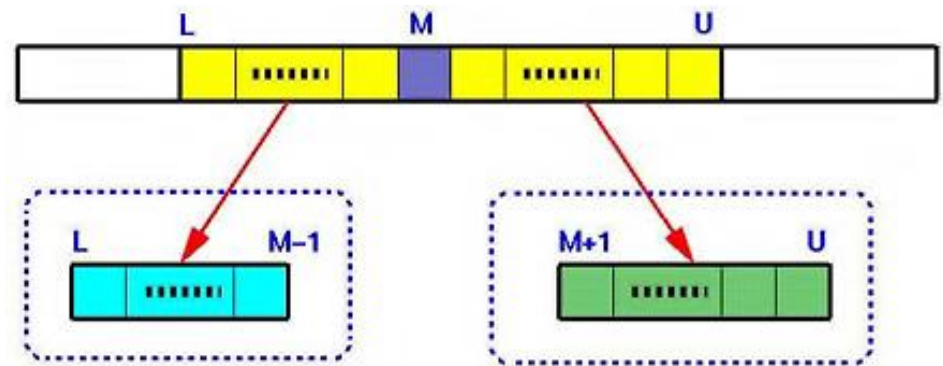
# ***Multicore Programming: 3/6***

- ***Balance***: Make sure that each thread has *equal* contribution, if possible, to the whole computation.
- If an insignificant thread runs frequently, occupying a core, other more useful threads would have less chance to run.



# Multicore Programming: 4/6

- **Data Splitting:** Data may also be split into different sections so that each of which can be processed separately.
- Matrix multiplication is a good example.
- Quicksort is another. After partitioning, the two sections can be sorted separately.



After partitioning a  $[L..U]$  into  $a[L..M-1]$  and  $a[M+1..U]$ , we may create two threads, one for each section. Then, each thread sorts its own section. Threads are created in a binary tree.



# ***Multicore Programming: 5/6***

- ***Data Dependency***: Watch for data items that are used by different threads. For example, two threads may update a common variable at the same time.
- Should this happen, unexpected results may occur. As a result, the execution of threads has to be ***synchronized*** so that only one thread can update a shared variable at any time.
- This is a very difficult issue in threaded programming.

# **Multicore Programming: 6/6**

- **Testing and Debugging:** The behavior of a threaded program is *dynamic*. A bug that appears in this test run may not occur in the next. Some bugs may never surface throughout the life-span of a threaded program, or may appear at an unexpected time.
- Some debugging issues (e.g., race condition – updating a shared resource at the same time, and system deadlock) do not have efficient solutions.
- Thus, testing and debugging is an art, and requires a careful design and planning.

# ***Thread Cancellation: 1/2***

- ***Thread cancellation*** means terminating a thread **before** its completion. The thread to be cancelled is the **target thread**.
- There are two types:
  - ❖ ***Asynchronous Cancellation***: the target thread terminates immediately.
  - ❖ ***Deferred Cancellation***: The target thread can periodically check if it should terminate, allowing the target thread an opportunity to terminate itself in an orderly way. The point a thread can terminate itself is a **cancellation point**.

# ***Thread Cancellation: 2/2***

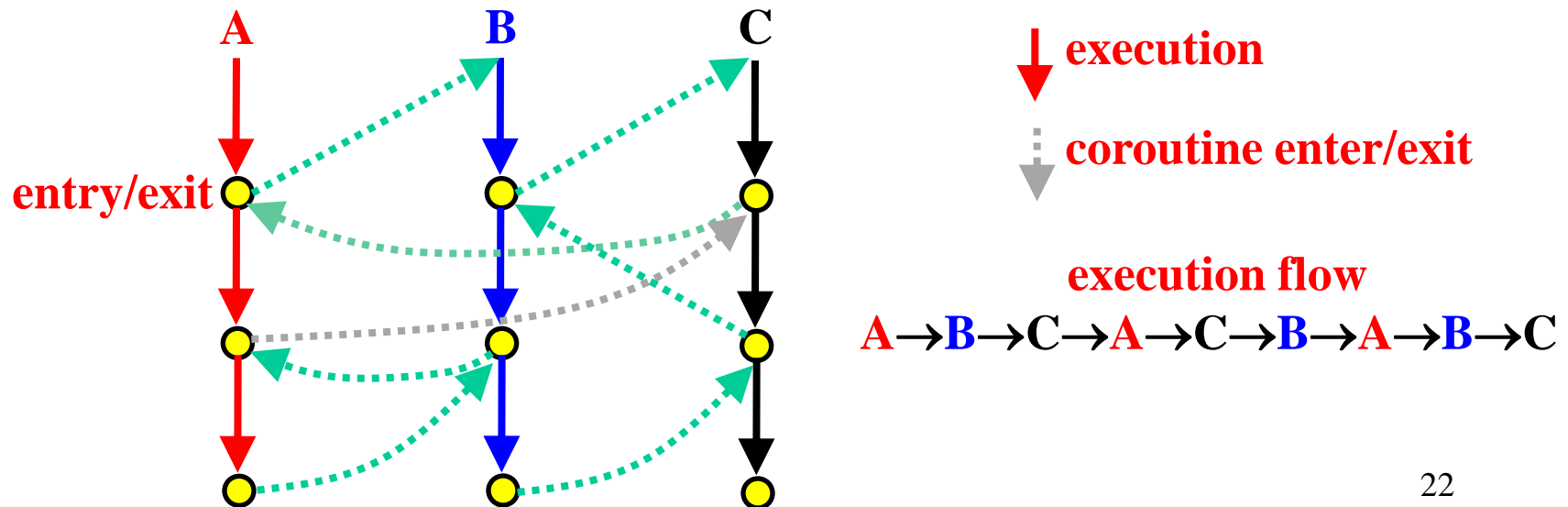
- With asynchronous cancellation, if the target thread owns some system-wide resources, the system may not be able to reclaim these resources because other threads may be using them.
- With deferred cancellation, the target thread determines the time to terminate itself. Reclaiming resources is not a problem.
- Many systems use asynchronous cancellation for processes (e.g., system call `kill`) and threads.
- POSIX Threads (i.e., Pthreads) supports deferred cancellation.

# ***Thread-Specific Data/Thread-Safe***

- Data that a thread needs for its own operation are **thread-specific**.
- Poor support for thread-specific data could cause problems. For example, while threads have their own stacks, they share the heap.
- What if two `malloc()`s are executed at the same time requesting for memory from the heap? Or, two `printf`s are run simultaneously?
- A library that can be used by multiple threads properly is a **thread-safe** one.

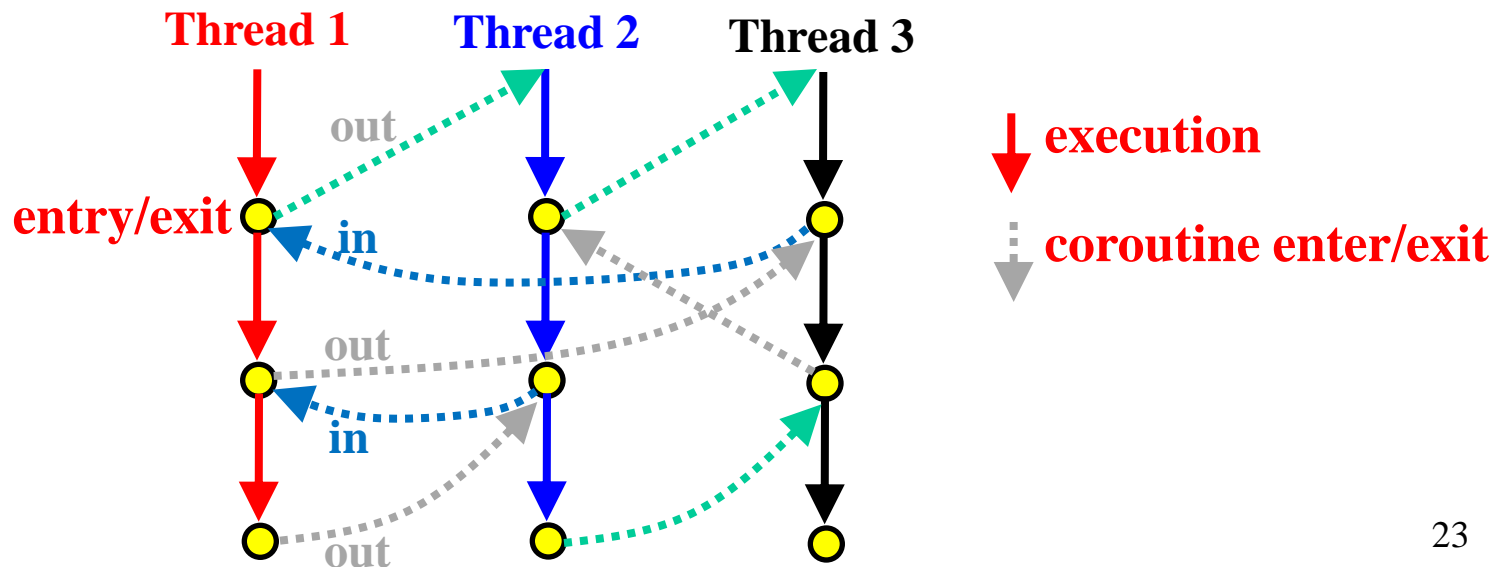
# Coroutines and Fibers: 1/3

- A conventional call to a function always starts from the very beginning of that function.
- A **coroutine** has multiple entry points and exits so that the next “call” to a coroutine resumes its execution from the statement/instruction following the previous exit point.



# Coroutines and Fibers: 2/3

- Do the enter and exit activities look like what a scheduler does?
- Yes, an exit is a switching out, and an enter/re-enter is a switching in.
- Hence, coroutines resemble scheduling activities.



# ***Coroutines and Fibers: 3/3***

- A ***fiber*** is a lightweight thread just like a thread is a lightweight process.
- A fiber is created in a thread and shares resource with other fibers of that thread.
- A fiber has a **stack**, a **subset of registers**, and **data (or local storage)** provided when it is created.
- Fibers are scheduled with **co-operative** scheduling.
- Co-operative scheduling means a fiber voluntarily and explicitly yields its execution to another fiber with a **YIELD** or similar function call.
- Thus, fibers are simpler than threads, and resemble coroutines.



**The End**