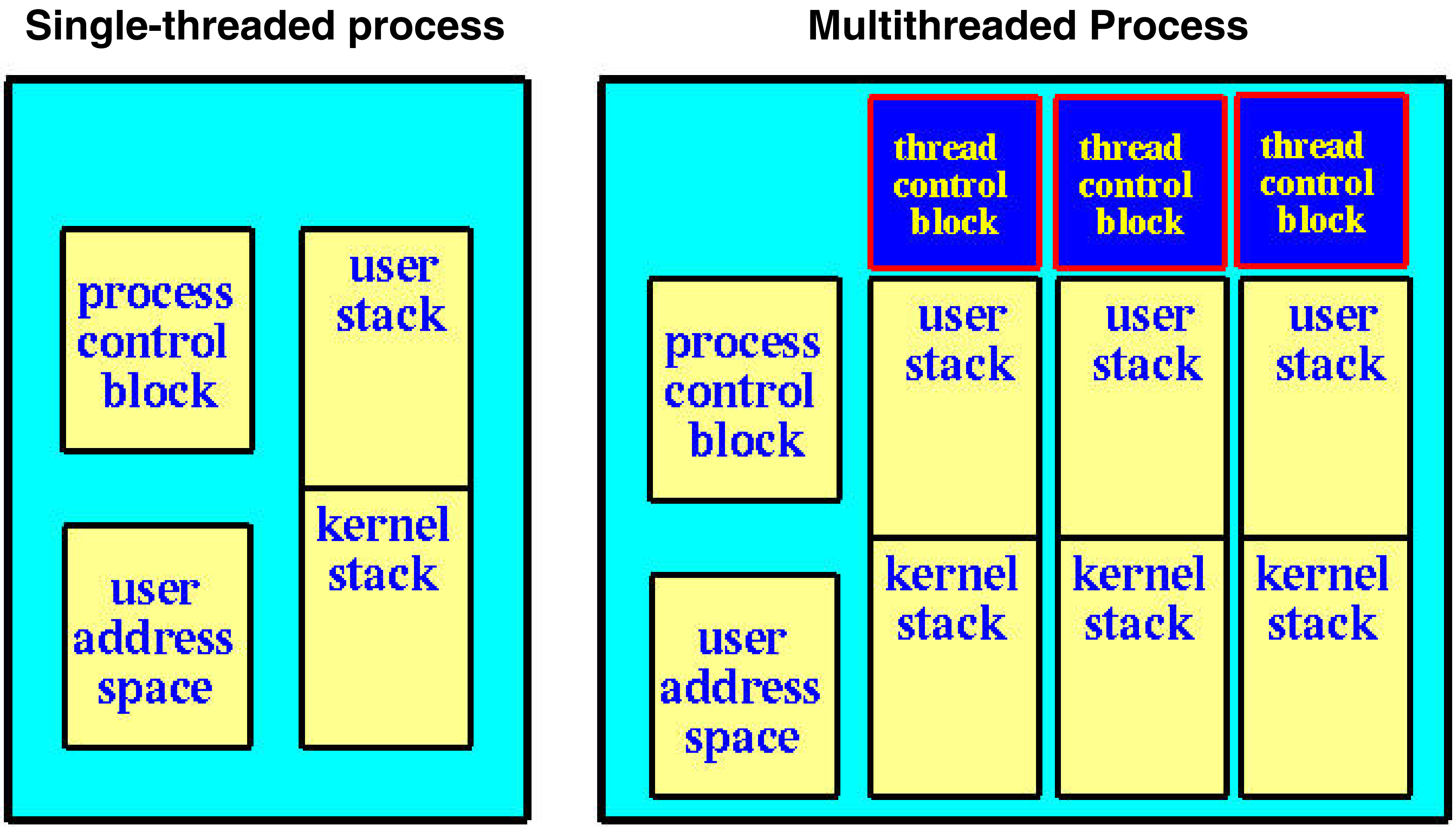
Part II
Process Management

Chapter 4: Threads

What Is a Thread?

- **A *thread*, also known as *lightweight process* (LWP), is a basic unit of CPU execution.**
- **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



Benefits of Using Threads

- **Responsiveness: Other parts (*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 can run multiple threads of the same process. No program change is necessary.**

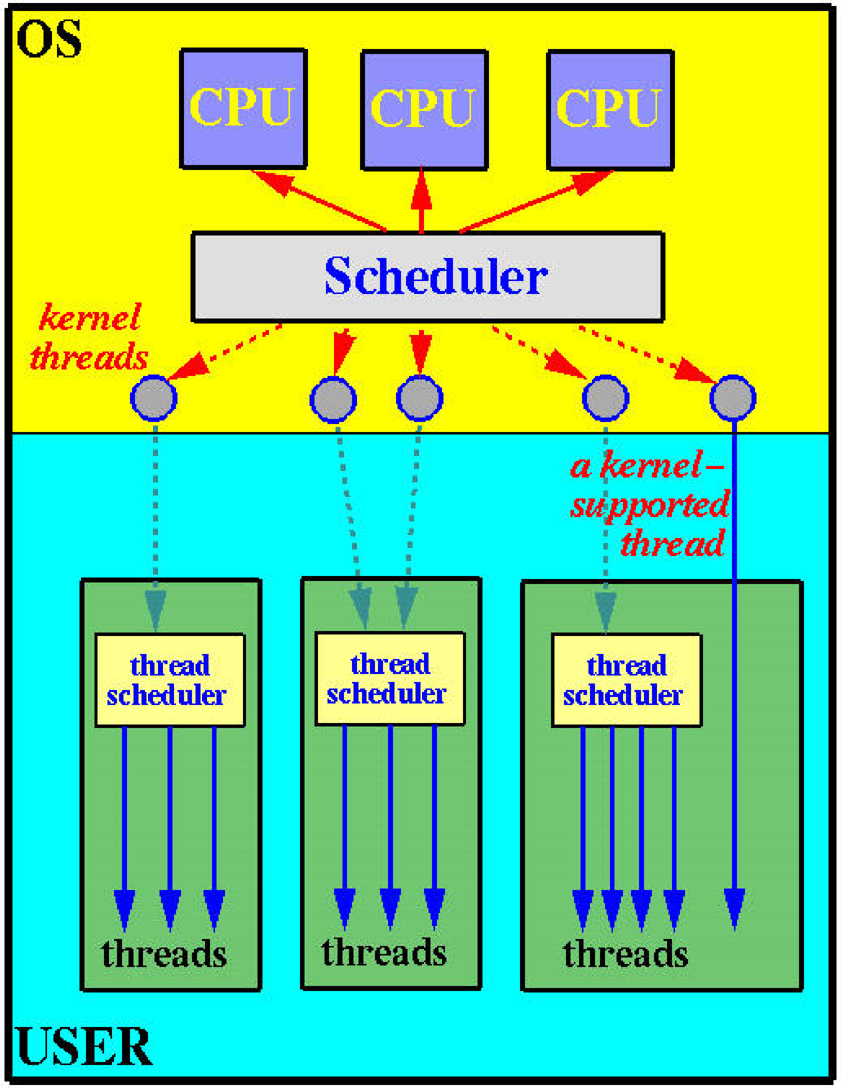
User and Kernel Threads: 1/3

- ***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.**
  - **There is no kernel intervention, and, hence, 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 threads.**
  - **In a multiprocessor environment, the kernel can schedule threads on different processors.**

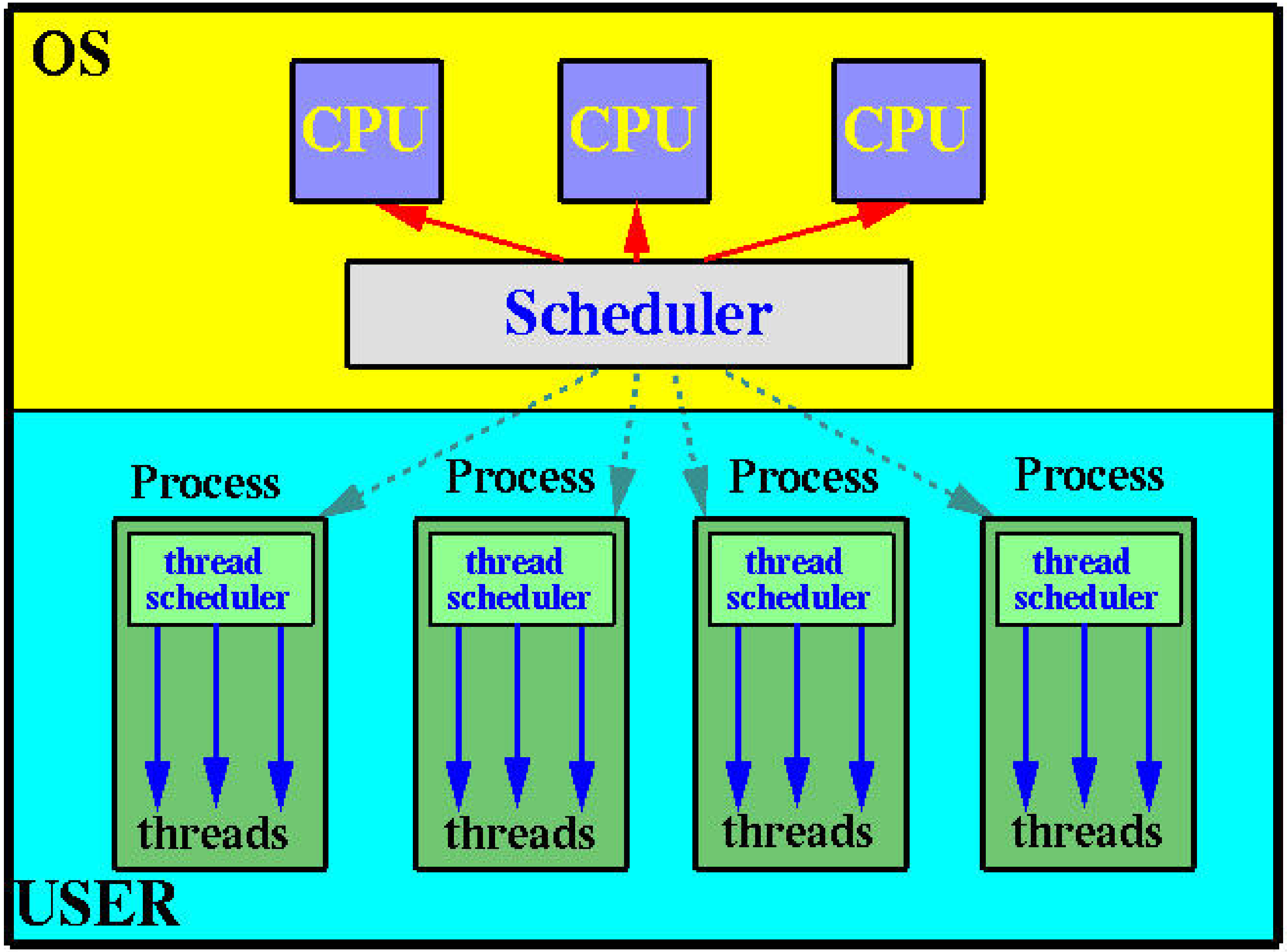
User and Kernel Threads: 3/3



Multithreading Models

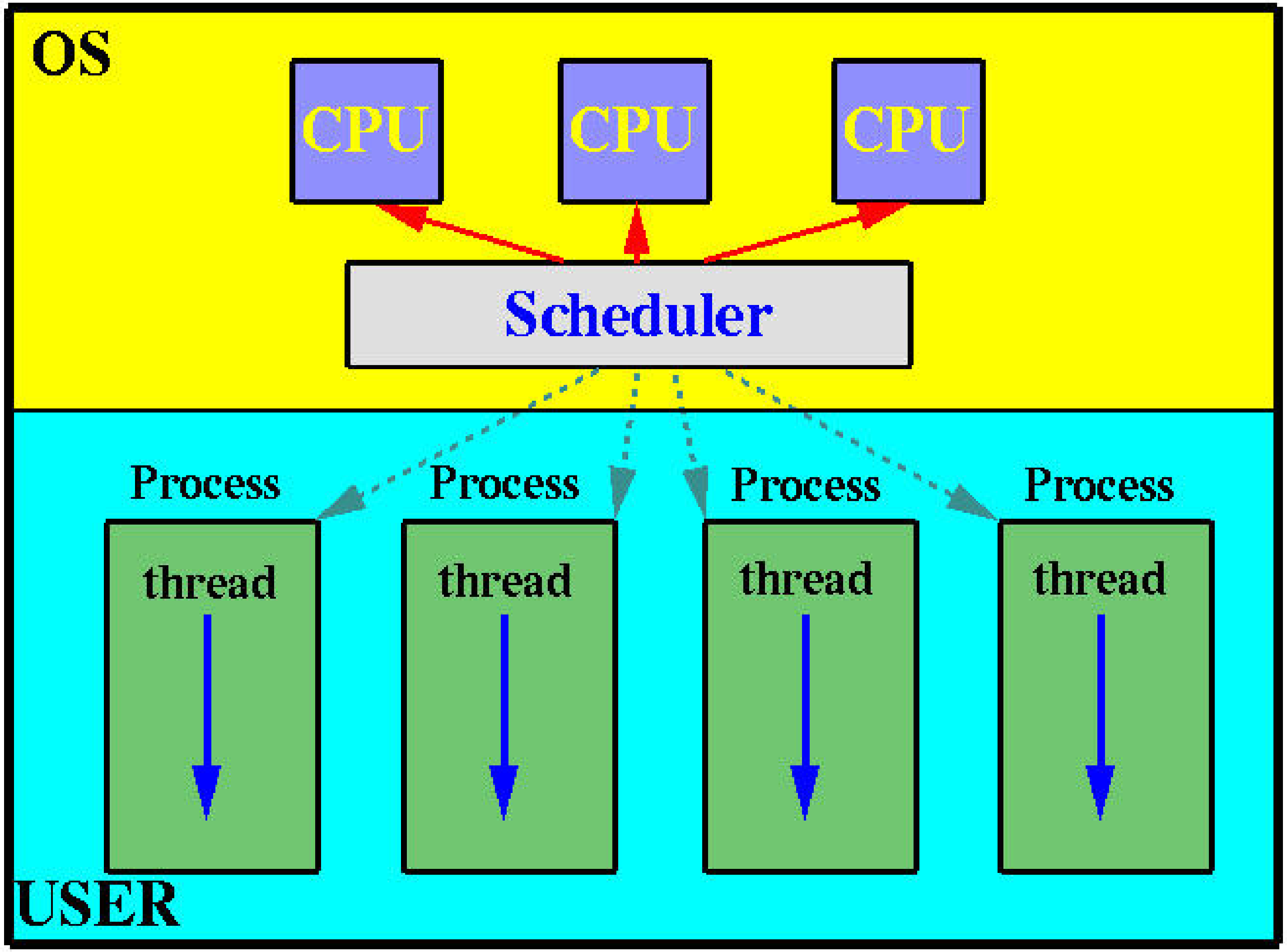
- **Different systems support threads in different ways. Here are three commonly seen thread models:**
  - ***Many-to-One Model*: One kernel thread (or process) has multiple user threads. Thus, this is a user thread model.**
  - ***One-to-One Model*: One user thread maps to one kernel thread (*e.g.*,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

Many-to-One Model

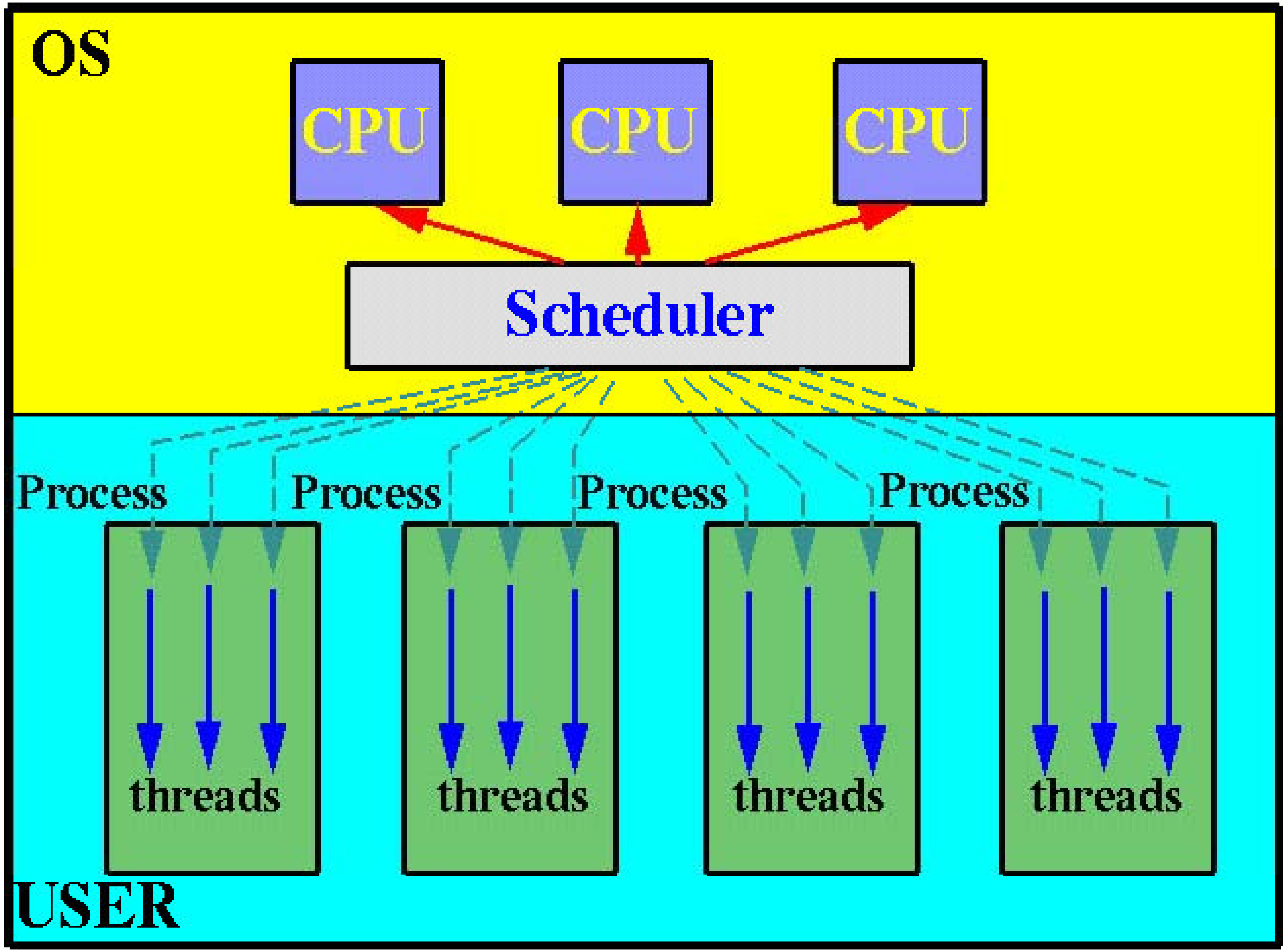


One-to-One Model: 1/2

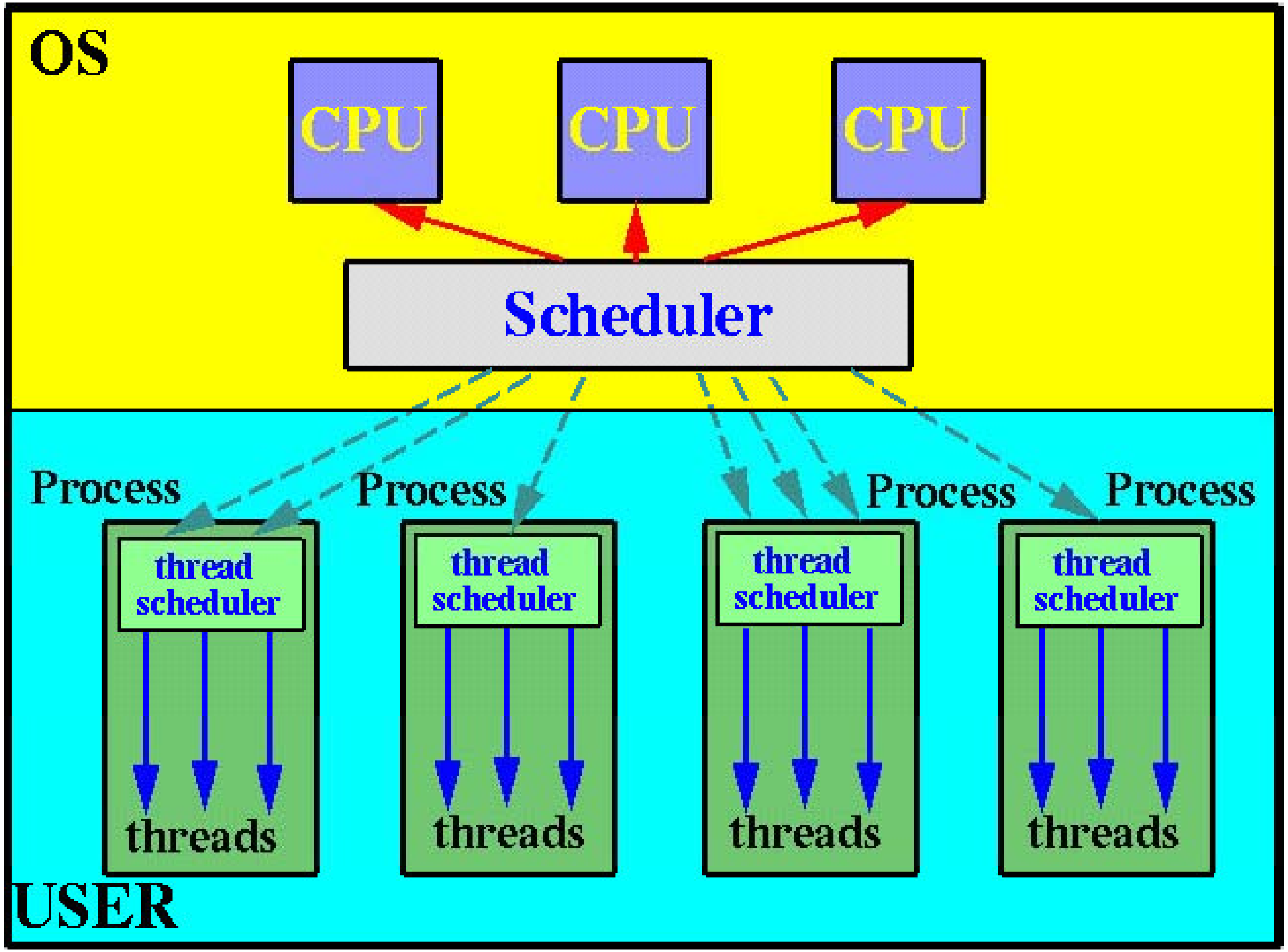
An Extreme Case: Traditional Unix



One-to-One Model: 2/2



Many-to-Many Model

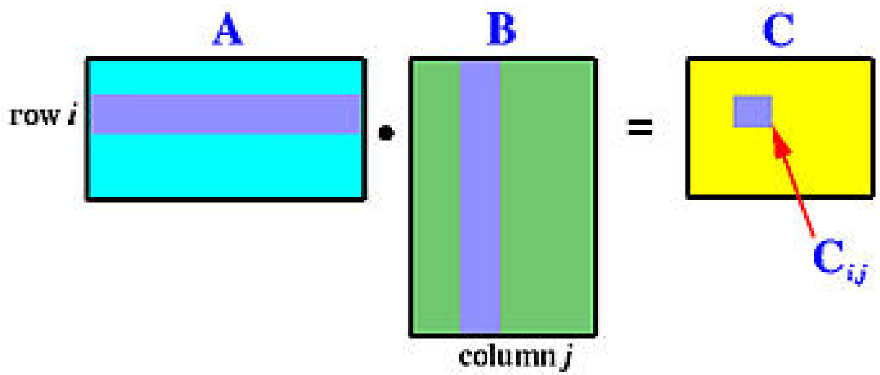


Multicore Programming: 1/6

- **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 can 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).



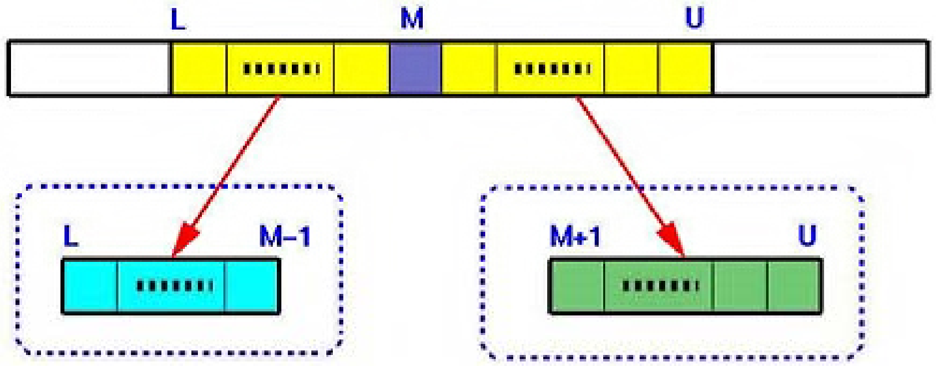
$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j}$$

Multicore Programming: 3/6

- Balance: **Make sure 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 they can be processed separately.
- Matrix multiplication is a good example.
- Quicksort is another. After partitioning, the two sections can be sorted separately.



Multicore Programming: 5/6

- **Data Dependency: Watch for data items that are used by different threads. For example, two threads may update a common counter `count` with `count++`.**
- **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 `count` at any time.**
- **This is the most difficult part in threaded programming. See Chapter 6.**

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 Issues

- **How Does a Thread Fork?**
- **Thread Cancellation**
- **Signal Handling**
- **Thread-Specific Data**
- **What Is Thread-Safe?**

How Does a Thread Fork?

- **If a thread forks, does the new process:**
  - **duplicate all threads?**
  - **contain only the forking thread (*i.e.*, single-threaded)?**
- **Some systems have two `fork` system calls, one for each case.**

Thread Cancellation: 1/2

- ***Thread cancellation*** **means terminating a thread before its completion. The thread that is 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

- **Problem: With asynchronous cancellation, if the target thread owns some system-wide resources, the system may not be able to reclaim all recourses owned by the target thread.**
- **With deferred cancellation, the target thread determines the time to terminate itself. Reclaiming resources is not a problem.**
- **Most systems implement asynchronous cancellation for processes (*e.g.*, system call `kill`) and threads.**
- **POSIX Threads (*i.e.*, Pthreads) supports deferred cancellation.**

Signal Handling

- ***Signals*** **is a way the OS uses to notify a process that some event has happened.**
- **Once a signal occurs, who is going to handle it? The process, or one of the threads?**
- **This is a very complex issue and will be discussed later in this course.**

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 or `new`'s are executed at the same time requesting for memory from the heap? Or, two `printf`'s or `cout`'s are run simultaneously?**
- **If a library can be used by multiple threads properly, it is a *thread-safe* one.**

Thread Pool

- **While we know that managing threads are more efficient than managing processes, creating and terminating threads are still not free.**
- **After a process is created, one can immediately create a number of threads and have them waiting.**
- **When a new task occurs, one can wake up one of the waiting threads and assign it the work. After this thread completes the task, it goes back to wait.**
- **In this way, we save the number of thread creation and termination.**
- **These threads are said in a *thread pool*.**

The End