Chapter 4
Threads

Part **A**: Section 4.**1**

Motivational Example: *Quicksort*

```
Recu_proc Quicks(Arr, lo, hi)
{
 if ( lo < hi ) then
 {
   p := call Parti(Arr, lo, hi);
   call Quicks(Arr, lo, (p-1));
   call Quicks(Arr, (p+1), hi);
  }
 }
```

Those students, who have not yet seen the quicksort algorithm before, are requested to study it in their own time from the CS literature.

- **Remarks:**
- The **Parti**tioning procedure does not only yield the split-value p, but also has a (pre-sorting) *side-effect* on the **Arr**ay
- Since the two recursive calls of **Quicks**ort work on *disjoint* parts of the **Arr**ay, they can at least in principle be carried out **in parallel (concurrently)**, *if* the memory, which holds the **Arr**ay, is allowed to be "shared" by the two recursive invocations.
- By parallelisation **we may *hope* to save some run-time, *if* the coordination of parallelism** (*"management-overhead"*) **does not "eat" the time that was saved** by parallelisation

Overview

- **Processes and threads**
  - Multithreading
  - Thread functionality
- **Types of threads**
  - User level and kernel level threads
- **Multicore and multithreading**
  - Performance of Software on Multicore
- *Windows process and thread management*
  - Management of background tasks and application lifecycles
  - Windows process
  - Process and thread objects
  - Multithreading
  - Thread states
  - Support for OS subsystems
- *Solaris thread and SMP management*
  - Multithreaded architecture
  - Motivation
  - Process structure
  - Thread execution
  - Interrupts as threads
- *Linux process and thread management*
  - Tasks/threads/namespaces
- *Android process and thread management*
  - Android applications
  - Activities
  - Processes and threads
- *Mac OS X grand central dispatch*

Processes and Threads

Resource Ownership

Process includes a virtual address space to hold the process image

- The OS protects processes and resources to prevent unwanted interference

Scheduling/Execution

Follows an execution path that *may be interleaved with other processes*

- A process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS

Processes and *Threads*

- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***
- The unit of resource ownership is referred to as a ***process*** or ***task***
- ***Multithreading -*** The ability of an OS to support multiple, concurrent paths of execution within a single process

Motivational Example: *Concurrent Quicksort*

```
Recu_proc Quicks(Arr, lo, hi)
{
 if ( lo < hi ) then
 {
   p := call Parti(Arr, lo, hi);
   call Quicks(Arr, lo, (p-1));
   call Quicks(Arr, (p+1), hi);
  }
 }
```

In our concurrent quicksort example:

- A **Process** would be the ***owner*** of the **Arr**ay, as well as the ***owner*** of this **program code** (algorithm)
- **Threads**, which can be independently ***scheduled*** (although they also "belong" to that Process), would do the work on the fields of the **Arr**ay.
  - Thereby every (recursive) **call** would "***spawn***" a new thread into existence

Single Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- **MS-DOS** is an example

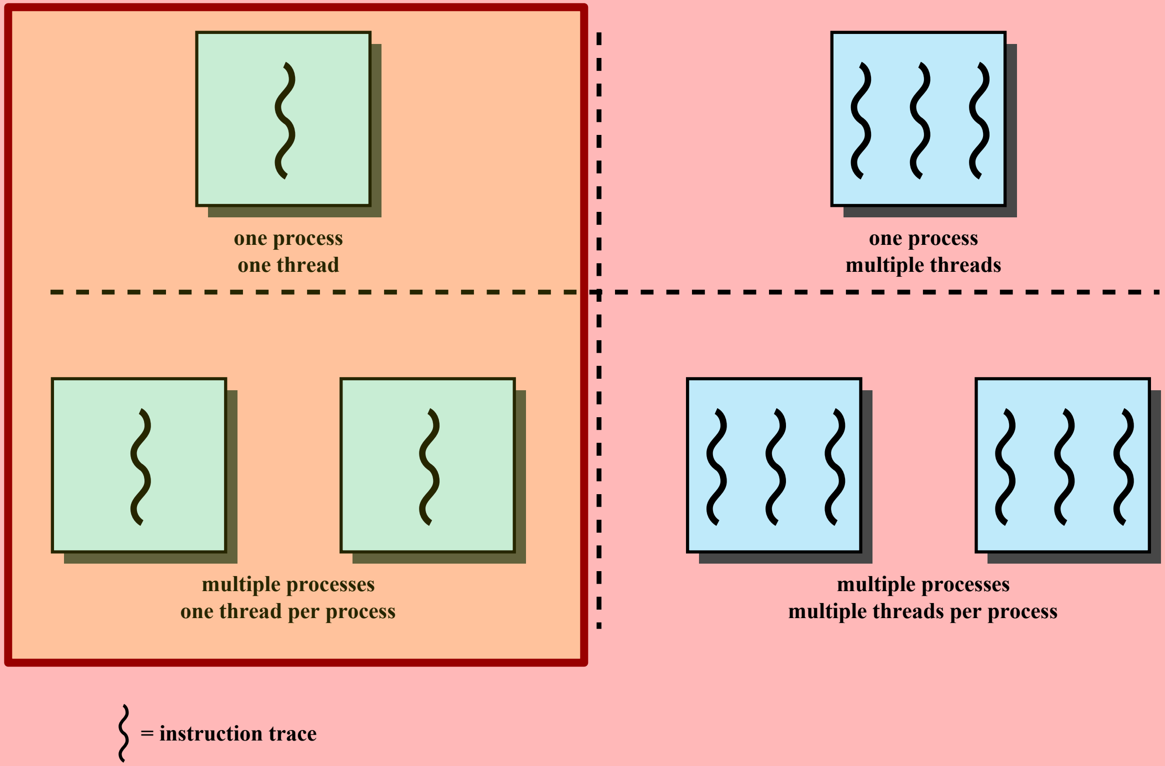


Figure 4.1 Threads and Processes

Multithreaded Approaches

- The right half of Figure 4.1 depicts multithreaded approaches
- A Java run-time environment is an example of a system of one process with multiple threads

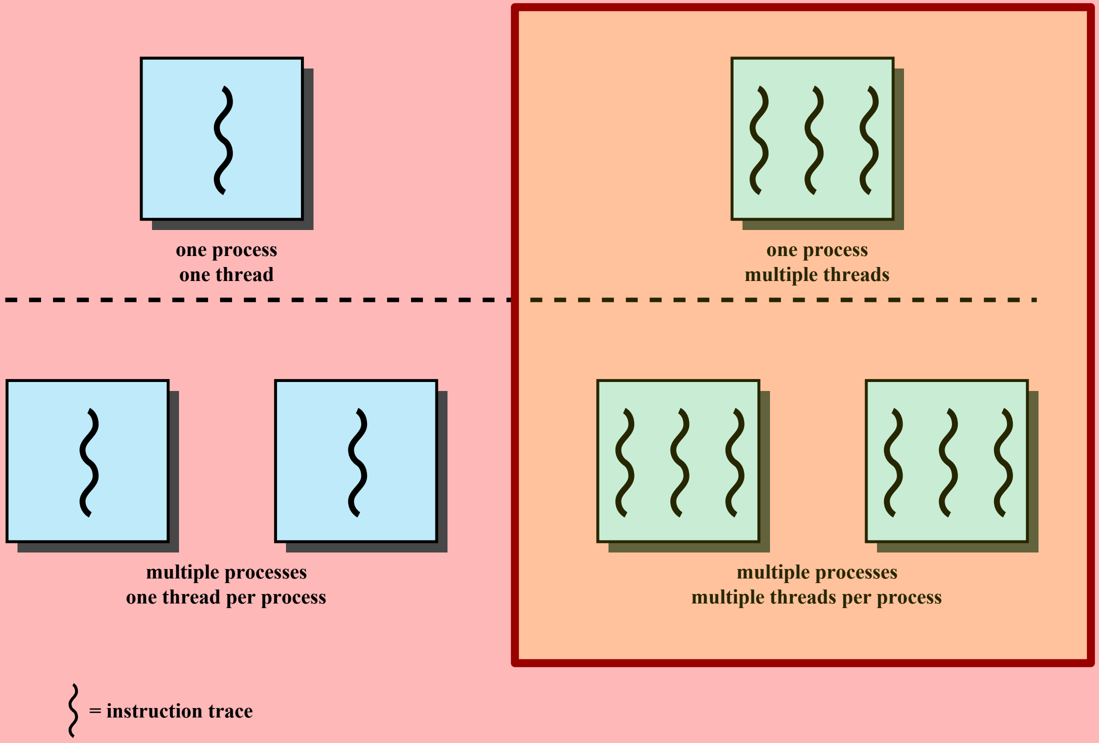


Figure 4.1 Threads and Processes

Process in a multi-threaded environment

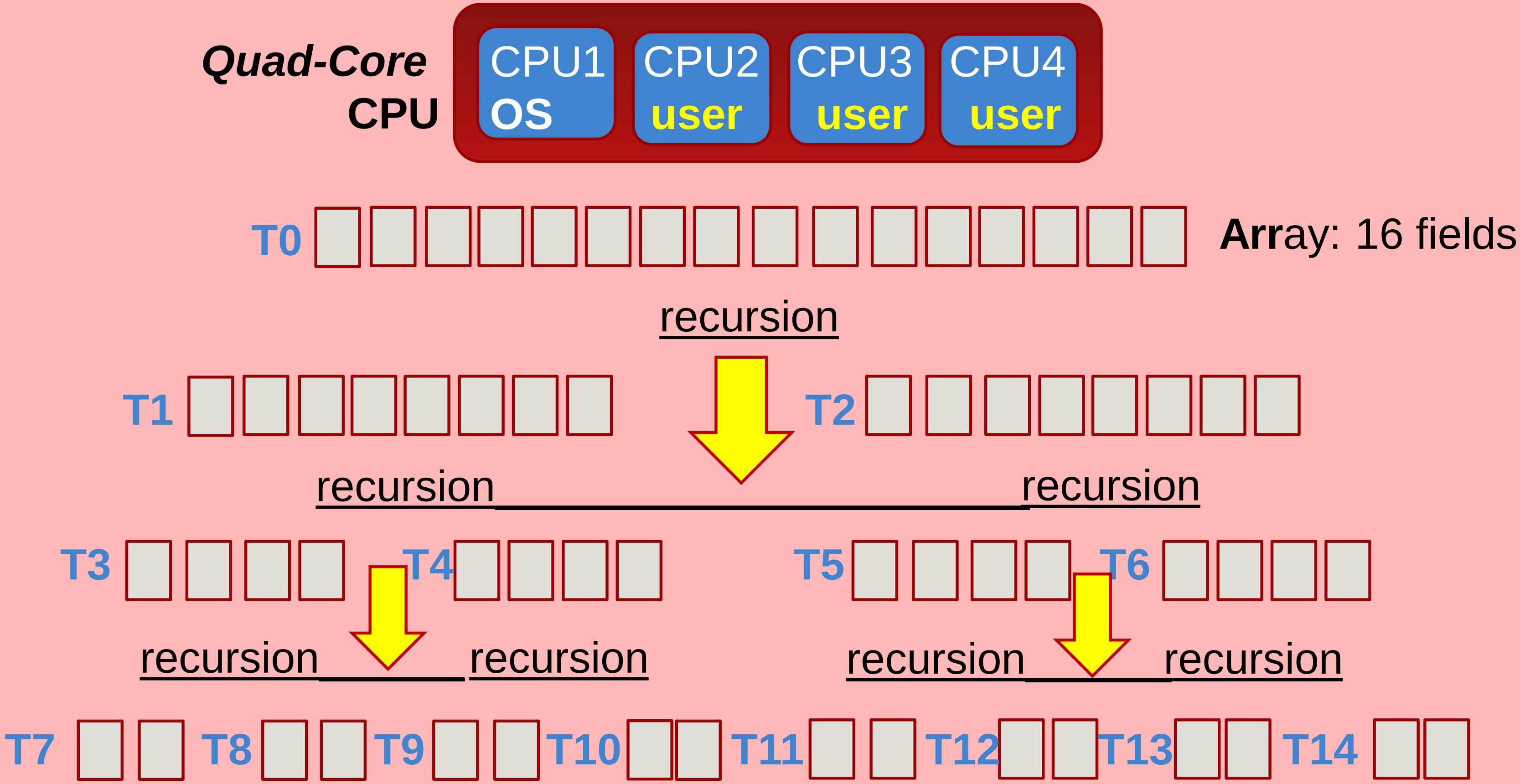
- Defined as "the **unit of resource allocation and** a unit **of protection**"
- **Associated** with processes:
  - A **virtual address space that holds the process image**
  - **Protected access** to:
    - Processors (CPUs)
    - Other processes (for interprocess communication)
    - Files
    - I/O resources (devices and channels)

One or More *Threads* in a Process

Each thread of a process has:

- An execution **state** (Running, Ready, etc.)
- A saved thread context when not running (Thread-Control-Block **TCB**)
- An execution **stack** (run-time stack)
- Some storage for its own local variables
- **Access to the memory and resources of its processes, shared with all other threads in that process**

Our Example again:
Concurrent Threaded Quicksort



After only a few recursions, we have already "*spawned*" far more threads (T...) than we have CPUs, and from that time onwards the thread-management efforts (scheduling, etc.) **begin to "eat"** the advantages of parallel computation !

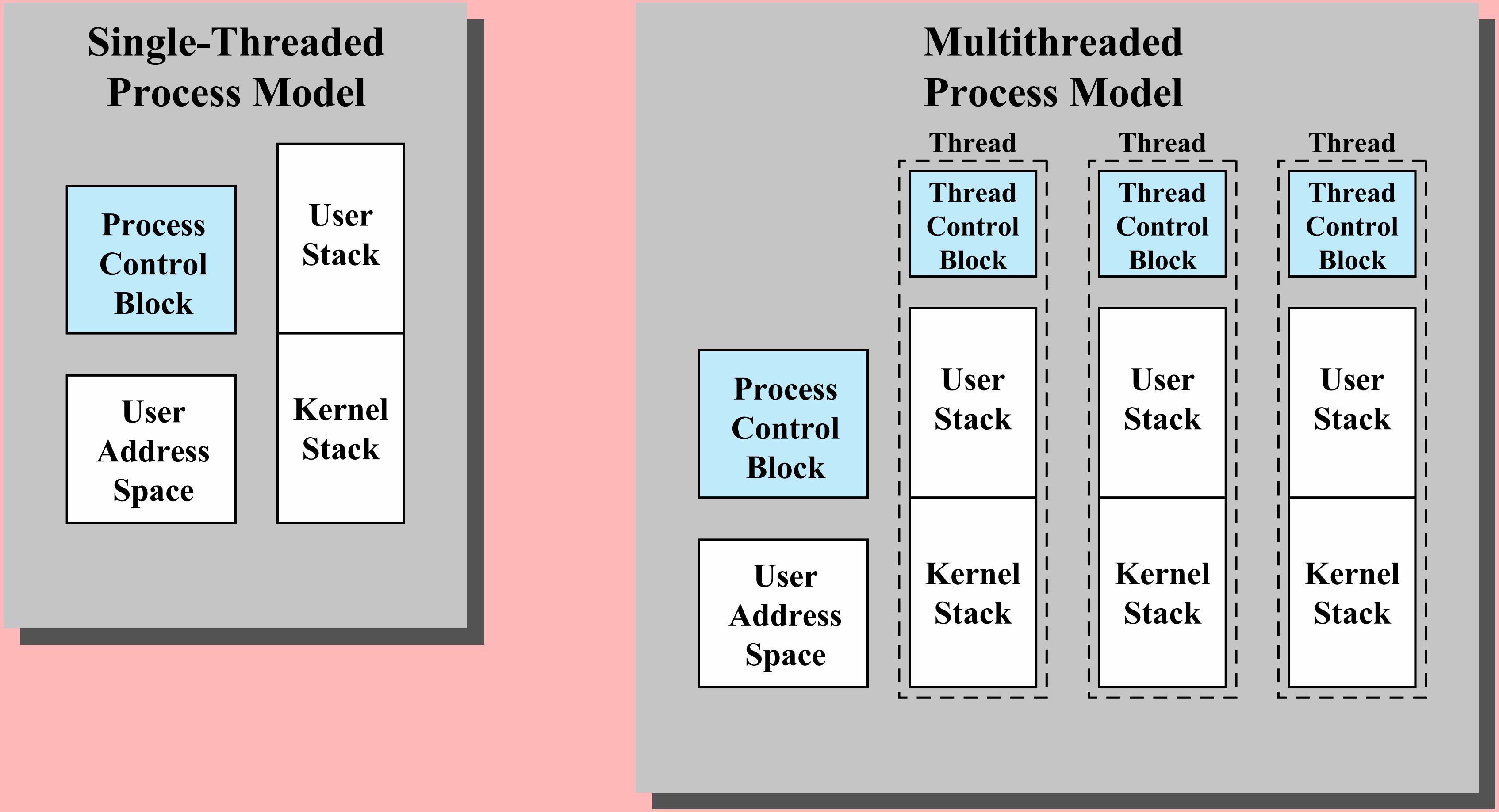


Figure 4.2 Single Threaded and Multithreaded Process Models

Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs

Thread Management

- In an OS that supports threads, scheduling and dispatching is done on a thread basis

Most of the state information dealing with execution is maintained in thread-level data structures

  - Suspending a process involves suspending all threads of the process
  - Termination of a process terminates all threads within the process

Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish

Example: Benefits of Threading

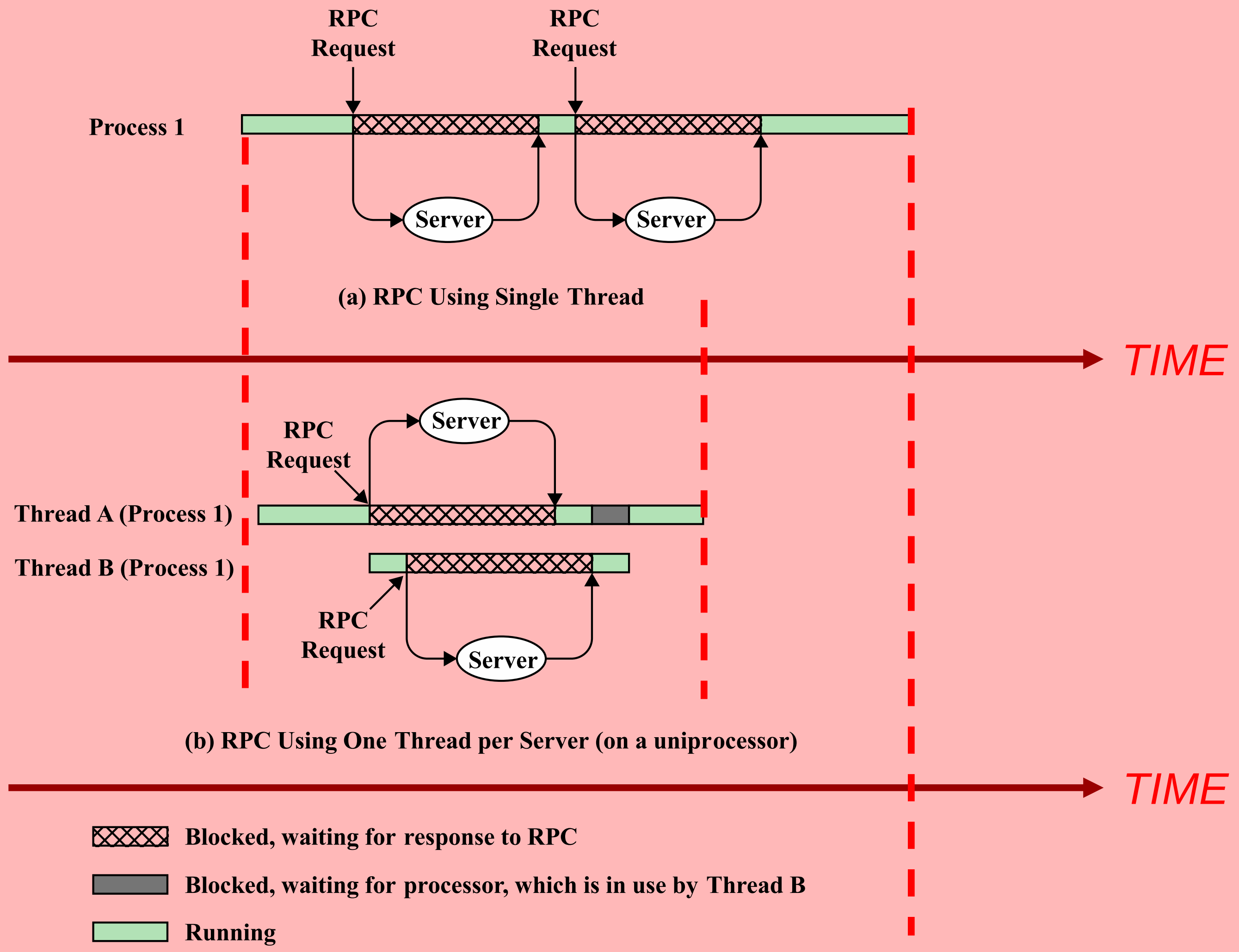


Figure 4.3 Remote Procedure Call (RPC) Using Threads

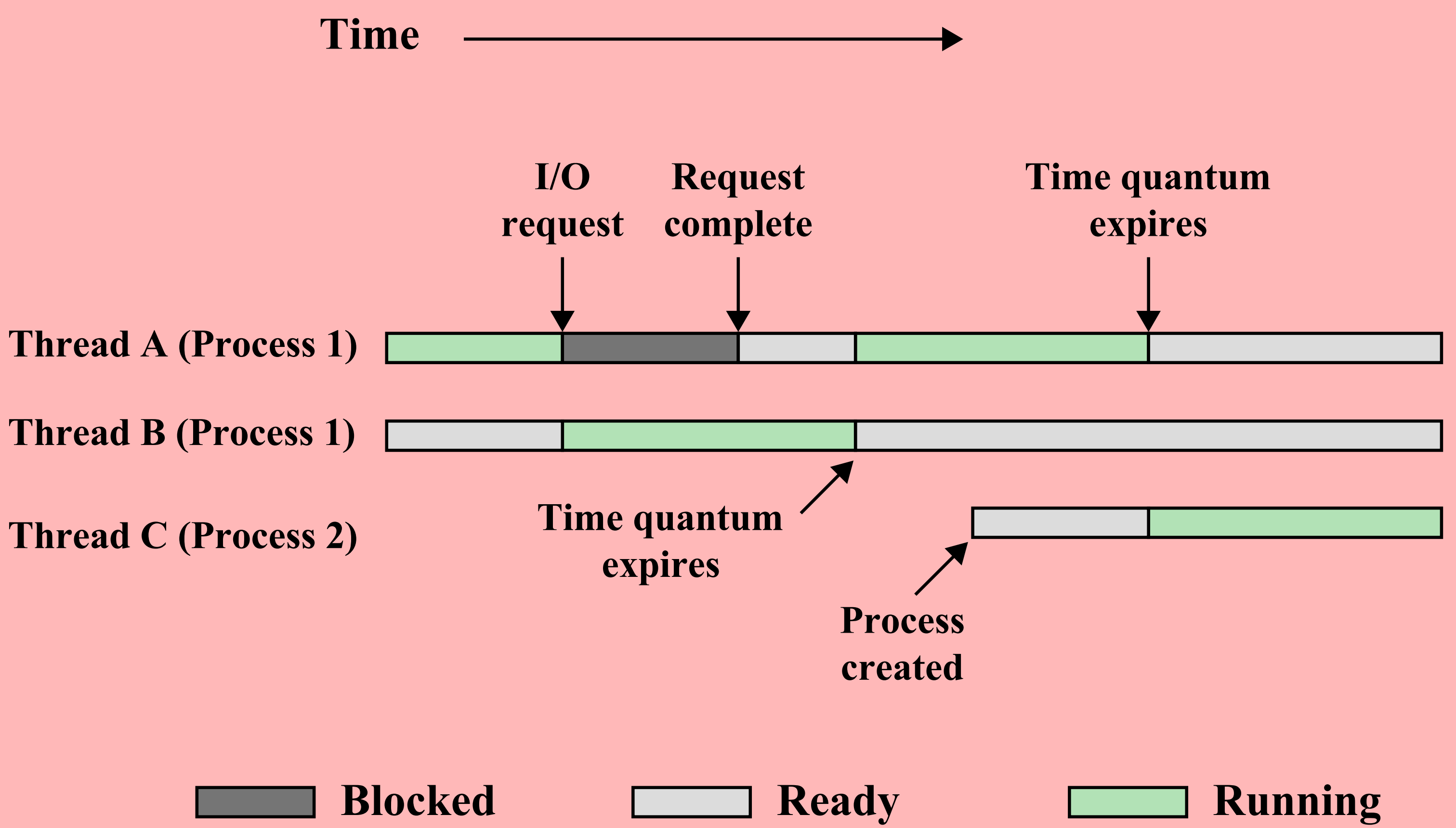


Figure 4.4 Multithreading Example on a Uniprocessor

Thread Synchronization

- It is necessary to synchronize the activities of the various threads
  - All threads of a process share the same address space and other resources
  - Any alteration of a resource by one thread affects the other threads in the same process

→ Chapters 5--6

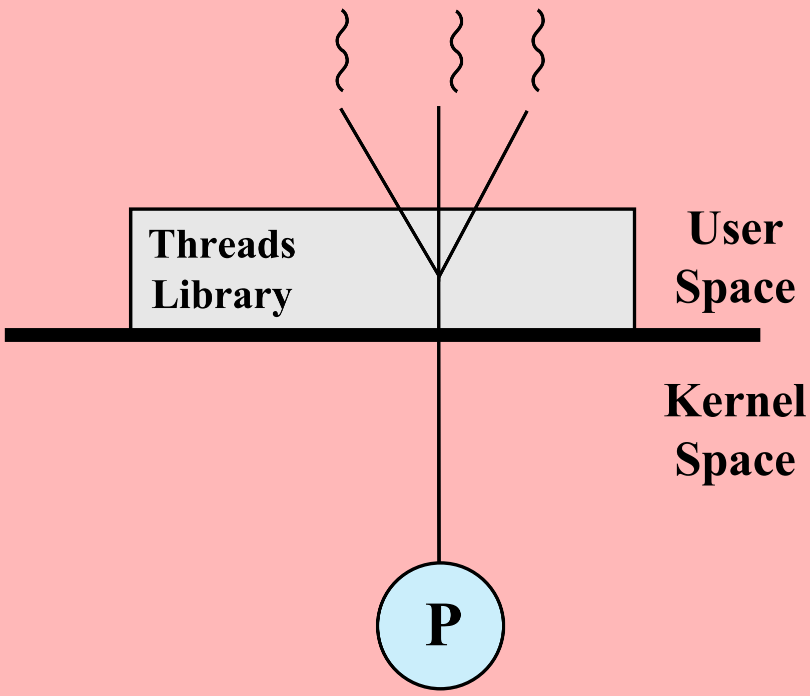
Types of Threads

User Level Thread (ULT)

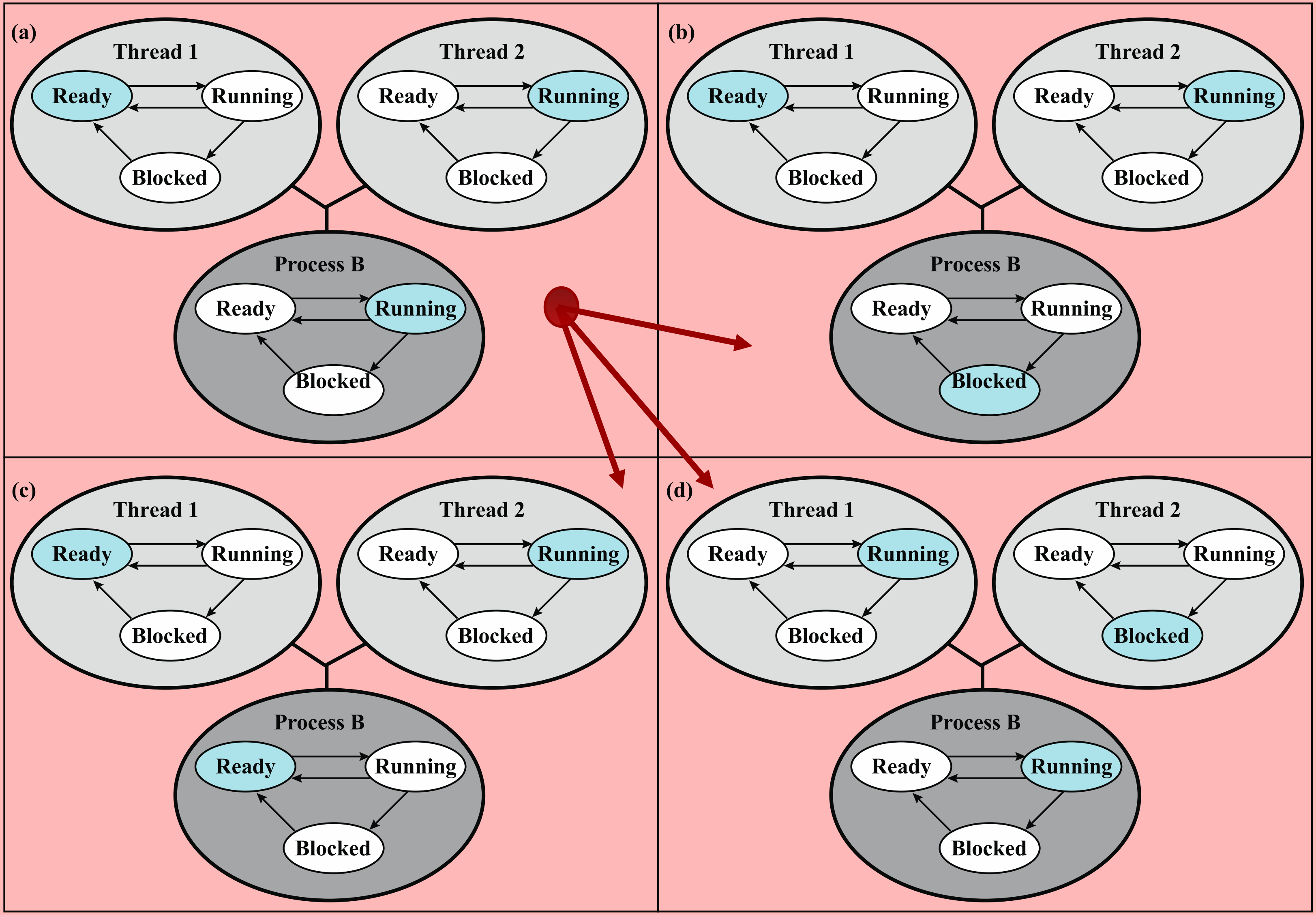
Kernel level Thread (KLT)

User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level



Colored state
is current state

Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

Advantages of ULTs

Thread switching does not require kernel mode privileges

Scheduling can be application-specific

➔ Chapters 9--10

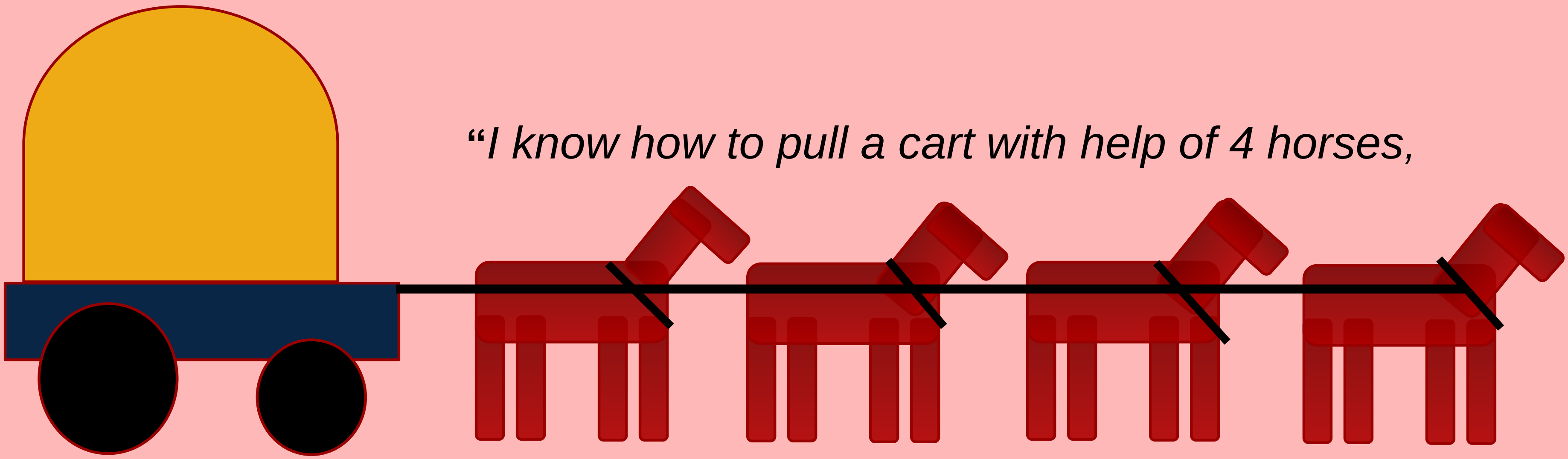
ULTs can run on any OS

Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - As a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
  - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time

Old proverb
in computer science ☺

“I know how to pull a cart with help of 4 horses,



but I do not know how to pull a cart with help of 1024 chickens”