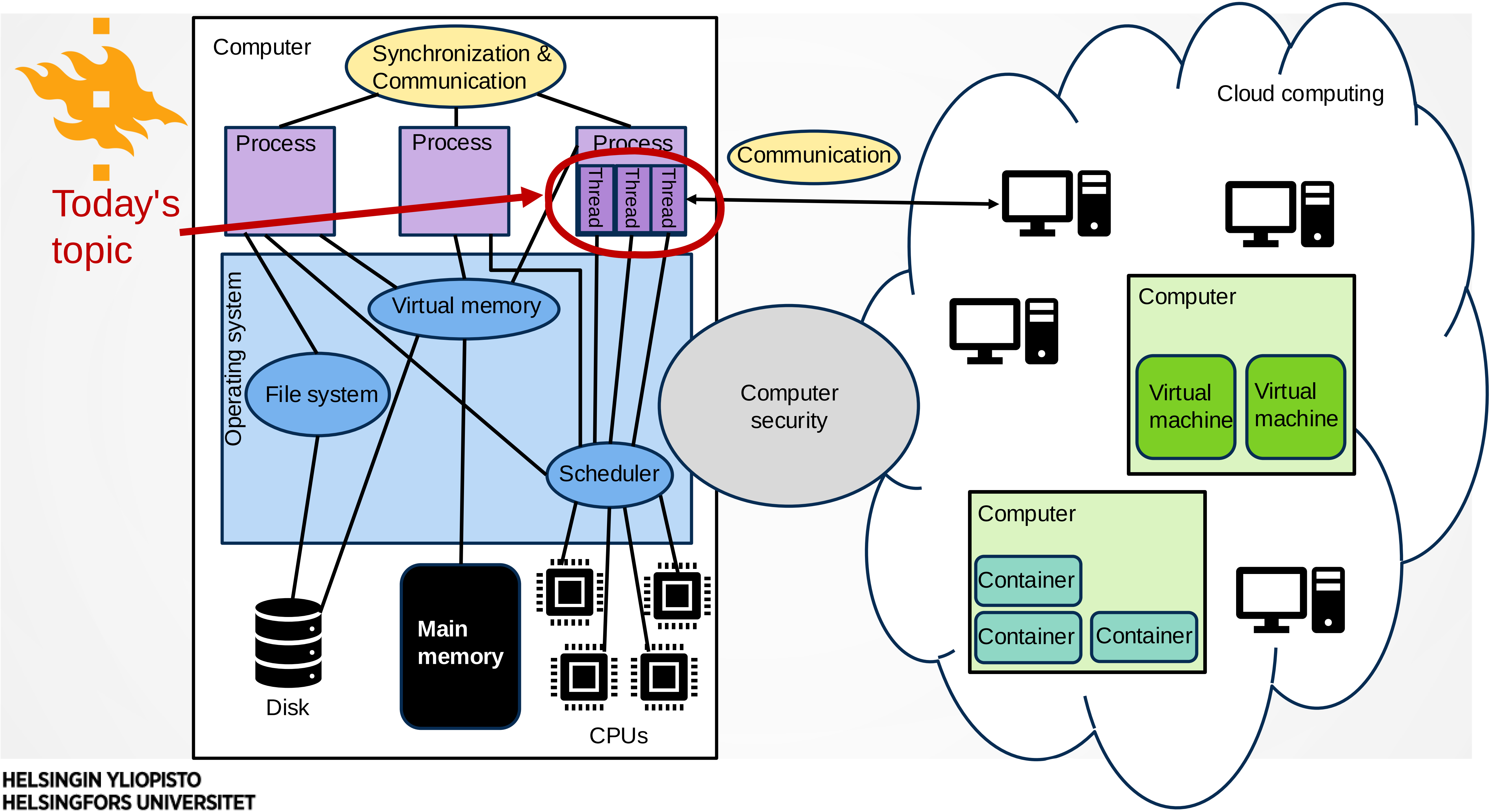
COMPUTING PLATFORMS

Threads


Today's topic
Computer
Synchronization & Communication
Process
Process
Process
Thread
Thread
Thread
Communication
Operating system
Virtual memory
File system
Scheduler
Computer security
Disk
Main memory
CPUs
Cloud computing
Computer
Virtual machine
Virtual machine
Computer
Container
Container
Container

LEARNING OUTCOMES

- After today's lecture, you
  - Understand the relationship and differences between threads and processes
  - Are able to describe how user-level and kernel-level threads differ and provide a comparison of the advantages and disadvantages
  - Know of Amdahl's law and how that relates to multithreading

WHAT ARE THREADS: TWO PROCESS CHARACTERISTICS

- Resource ownership
  - Own virtual address space holding the process image
  - OS protects processes from each other to prevent unwanted interference wrt. resources
- Scheduling / execution
  - Process has an execution state (**running**, **ready**, etc.) and a dispatching priority
  - Entity that is scheduled and dispatched by the OS

WHAT ARE THREADS: TWO PROCESS CHARACTERISTICS

Process / task

- Resource ownership
  - Own virtual address space holding the process image
  - OS protects processes from each other to prevent unwanted interference wrt. resources
- Scheduling / execution

Thread / lightweight process

  - Process has an execution state (**running**, **ready**, etc.) and a dispatching priority
  - Entity that is scheduled and dispatched by the OS

- **Multithreading**: Ability of an operating system to support **multiple, concurrent** paths of execution **within a single process**

WHY THREADS?

- What are advantages of threads over processes?
- What are the uses of threads?

ADVANTAGES OF THREADS

- **Faster to create** a new thread than a new process: Can be ten times faster
- **Faster to terminate** a thread than a process
- **Switching** between two threads **faster** than switching between processes
- Threads **enhance efficiency in communication between programs** (when implemented as threads, not separate processes)
  - Communication between processes in most cases requires the intervention of the kernel (OS protects processes from each other)
  - Threads within a process share memory and files, and thus can communicate with each other **without invoking the kernel**

USES OF THREADS

- **Foreground and background work**
  - E.g., a spreadsheet program: one thread displays user interface, another thread executes user commands and updates the spreadsheet
- **Asynchronous processing**
  - E.g., a word processor may create a backup file of unsaved changes once every minute using a dedicated thread
- **Speed of execution**
  - One thread can be reading data from disk, while another is computing on a previously read batch of data.
  - On a multiprocessor system, multiple threads of the same process can execute simultaneously
- **Modular program structure**

PROCESS MODEL WITH THREADS

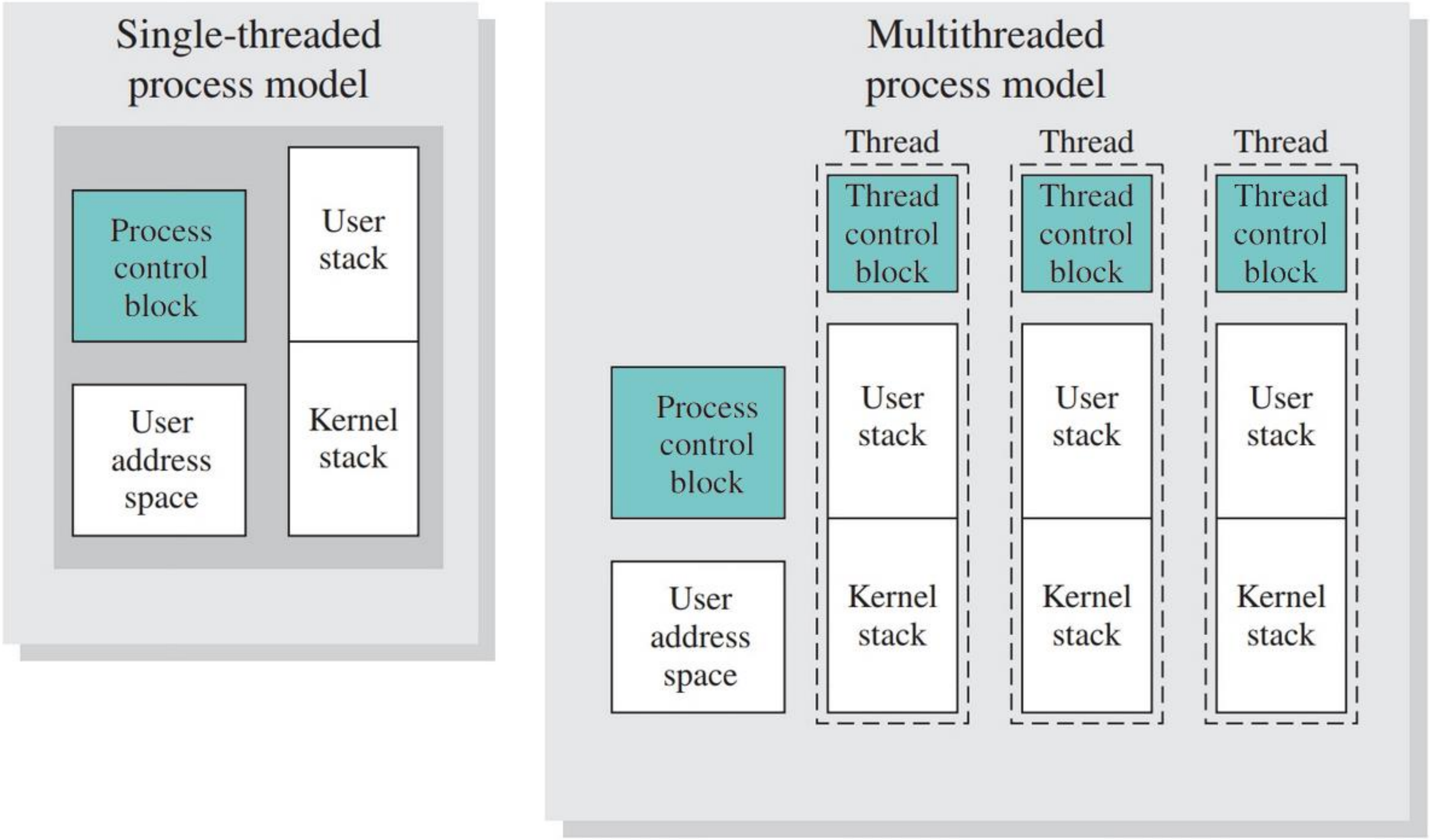


Figure 4.2 Single-Threaded and Multithreaded Process Models

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

EACH THREAD HAS...

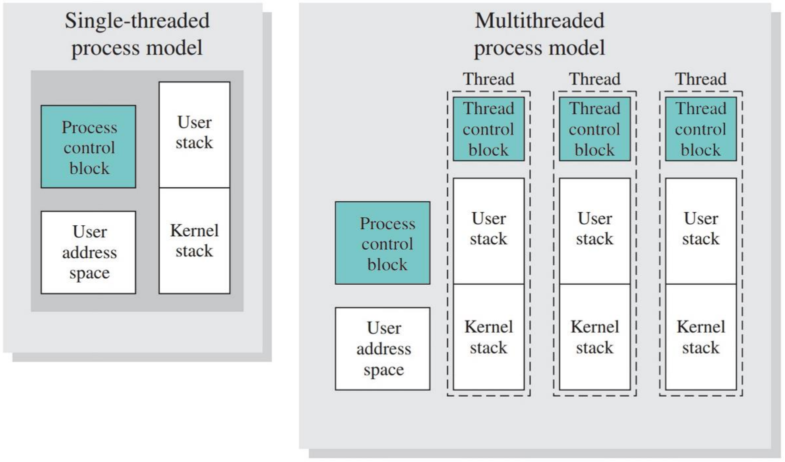


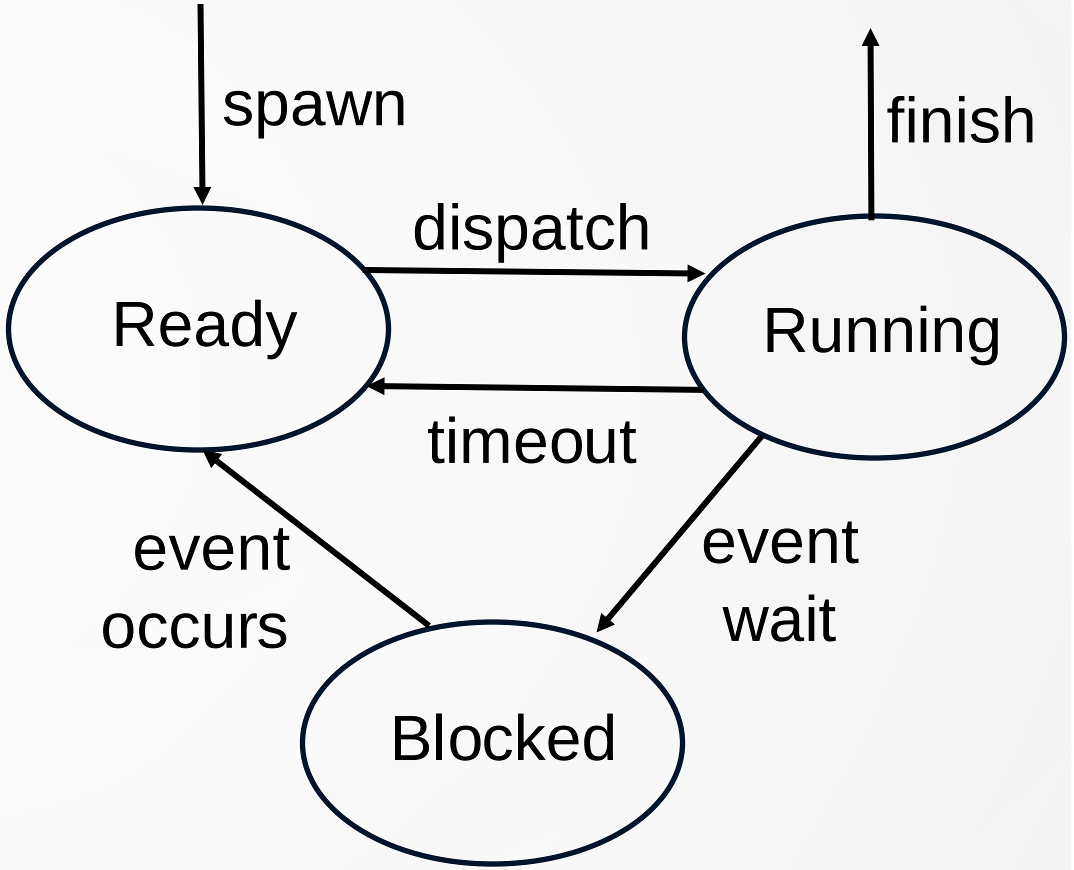
Figure 4.2 **Single-Threaded and Multithreaded Process Models**

- **Private**:
  - Execution state (**running**, **ready**, …)
  - Saved thread context when not running (CPU registers)
  - Execution stack
  - Some static storage for local variables, i.e., "global" data for a thread
- **Shared**:
  - Access to the **memory** and **resources** of its process
  - When one thread alters an item of data in memory, all threads in the process see the results (if they access it).
  - If one thread opens a file with read privileges, all threads (in the same process) can also read from that file. (*What about writing...*)

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

THREAD STATES

- **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 suspends all its threads
  - **Terminating** a process terminates all its threads
- Thread states are thus
  - **Running**
  - **Ready**
  - **Blocked**



EXAMPLE WITH THREADS

- Thread C begins to run after thread A exhausts its time quantum. Note that thread B is also ready to run. This is a **scheduling decision** (a topic covered in the next lecture)

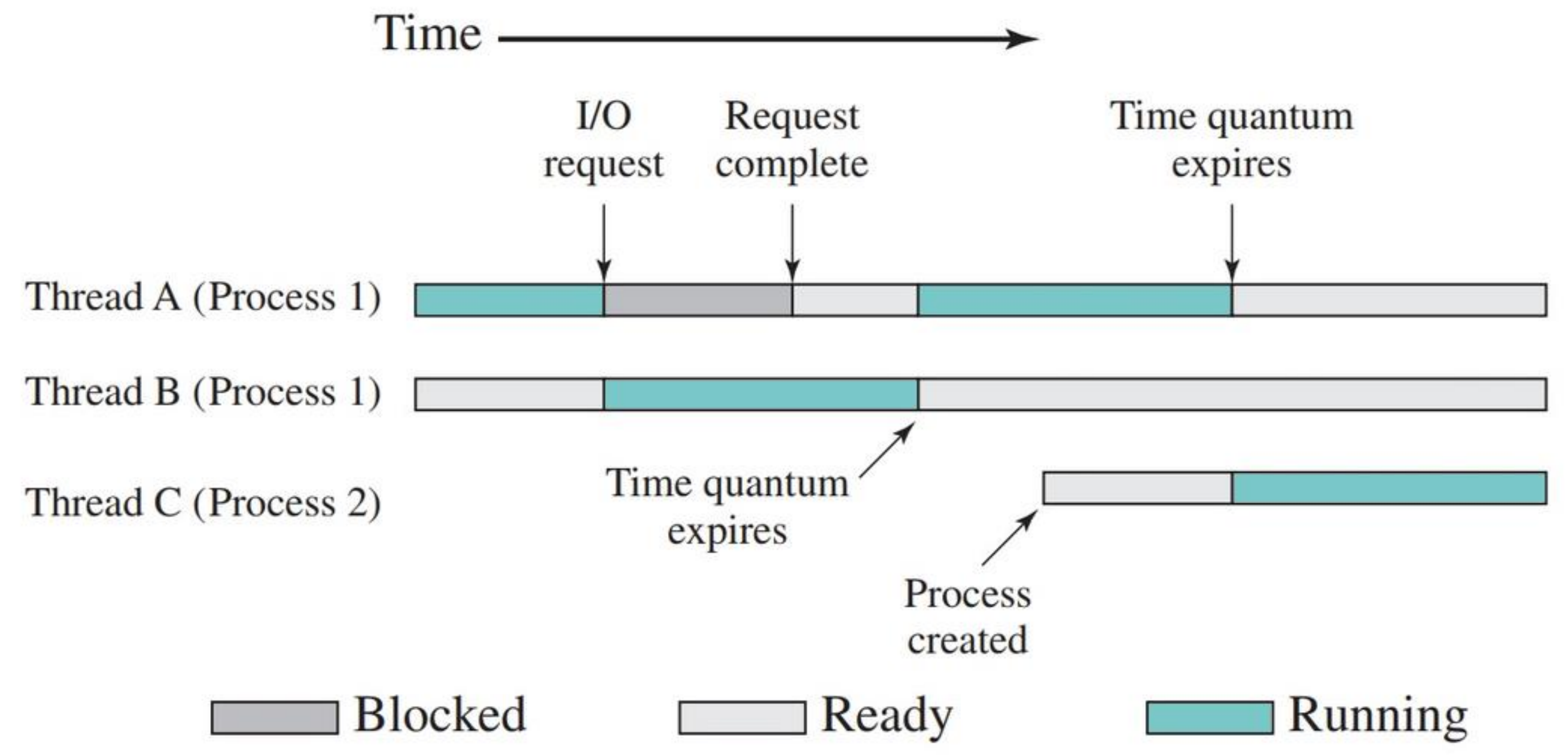


Figure 4.4 Multithreading Example on a Uniprocessor

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

THREAD SYNCHRONIZATION

- All threads of a process **share the same address space** and other **resources** (e.g. open files)
- Any alteration of data in memory or other resources **affects all threads**
- Necessary to **synchronize** the threads so that they do not interfere with each other or corrupt data structures
- Synchronization issues covered later in the course

USER-LEVEL THREADS VS KERNEL-LEVEL THREADS

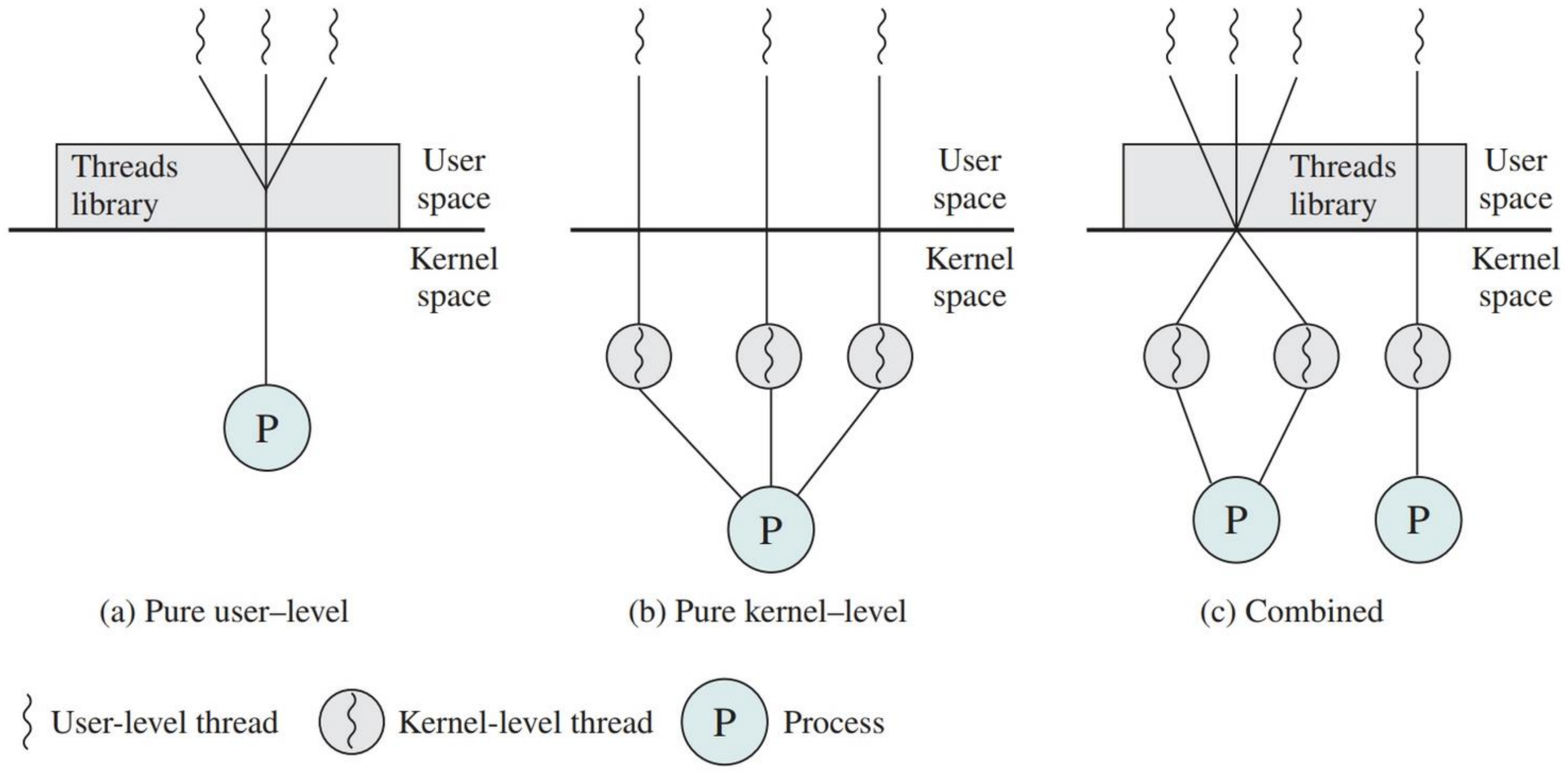
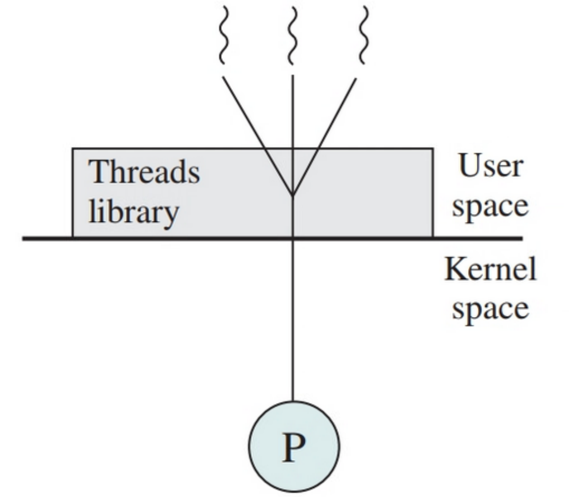


Figure 4.5 **User-Level and Kernel-Level Threads**

Figure from [Stallings, Operating systems: Internals and design principles, 9th ed]

USER-LEVEL THREADS (ULT)



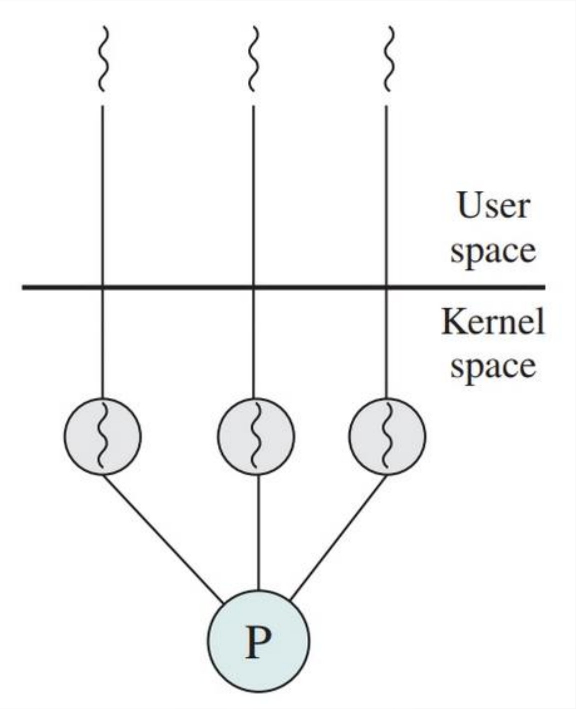
- Thread management is done by the application;
  The **kernel is not aware of the existence of threads**.
- Multithreaded applications are programmed using a **threads library** which contains code for
  - Creating and destroying threads
  - Passing messages and data between threads
  - Scheduling thread execution
  - Saving and restoring thread contexts
- Kernel **schedules the process as a unit** and assigns a single execution state to the process (**Ready, Running, Blocked,…**)

ULT – ADVANTAGES AND DISADVANTAGES

- Thread switching does not require kernel-mode privileges: saves the overhead of two mode switches
- Scheduling can be application specific
- ULTs can run on any OS: The kernel does not even know of the existence of threads
- A **blocking system call** in ULT causes all threads in the process to be blocked
- A multithreaded application cannot take advantage of multiprocessing: Kernel assigns one process to one processor at a time

KERNEL-LEVEL THREADS (KLT)

- All thread management done by the kernel;
  No thread management code in the application level
- Kernel maintains context information for the process as a whole and for each individual thread in the process
- Scheduling in the OS done on a thread basis
- Overcomes two weaknesses of ULT:
  - Kernel can schedule multiple threads from the same process on multiple processors
  - If one thread is blocked, other threads in the process can still be scheduled by the OS
- Kernel can be multithreaded too!
- Drawback: Thread switching now requires two mode switches; much slower than for ULT



MULTITHREADING ON MULTICORE SYSTEMS

- In a multicore system many threads of a single application can run concurrently
- How does this affect performance?
  - Depends how well the application can exploit the parallel resources

AMDAHL'S LAW

$$\text{speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}}$$

- Let *f* be the fraction of code that is (infinitely) parallelizable
- *(1-f)* is then the fraction of code that is inherently serial

$$\text{speedup} = \frac{1}{(1 - f) + \dfrac{f}{N}}$$

AMDAHL'S LAW

$$\text{speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

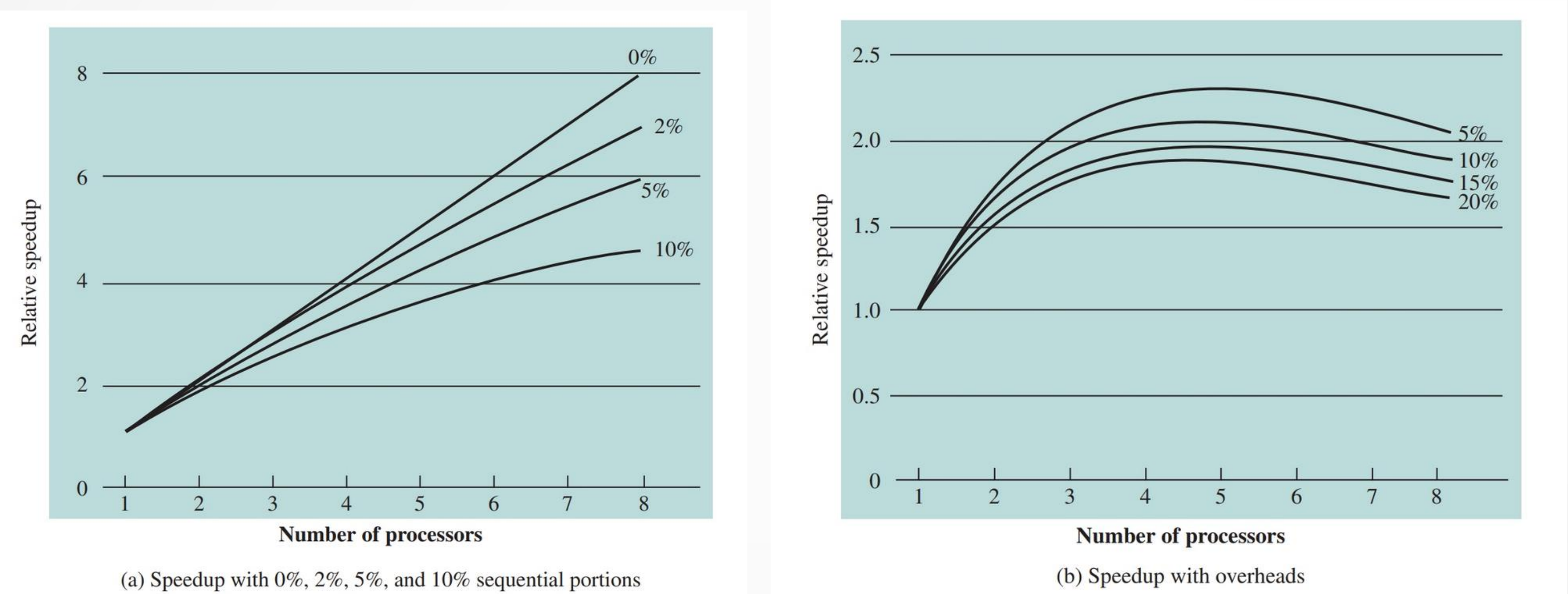


Figure 4.7 **Performance Effect of Multiple Cores**

Figures from [Stallings, Operating systems: Internals and design principles, 9th ed]

SUMMARY

- **Thread** is a scheduling unit
- Two ways of implementing threads: **user-level threads** and **kernel-level threads**
- Benefits of multithreading depend on the fractions of serial and parallelizable code