1 System & Activities

Gerd Liefländer

28. Oktober 2008

System Architecture Group

Roadmap for Today & Next Week

- System Structure
  - System Calls
  - (Java) Virtual Machine
- Basic System Abstractions
  - Address Space
  - Activities
    - Procedures
    - Process, Task
    - Threads
      - Kernel Level Threads
      - User Level Threads
- Assignment Hints
- OS Kernels
  - Monolithic
  - Micro

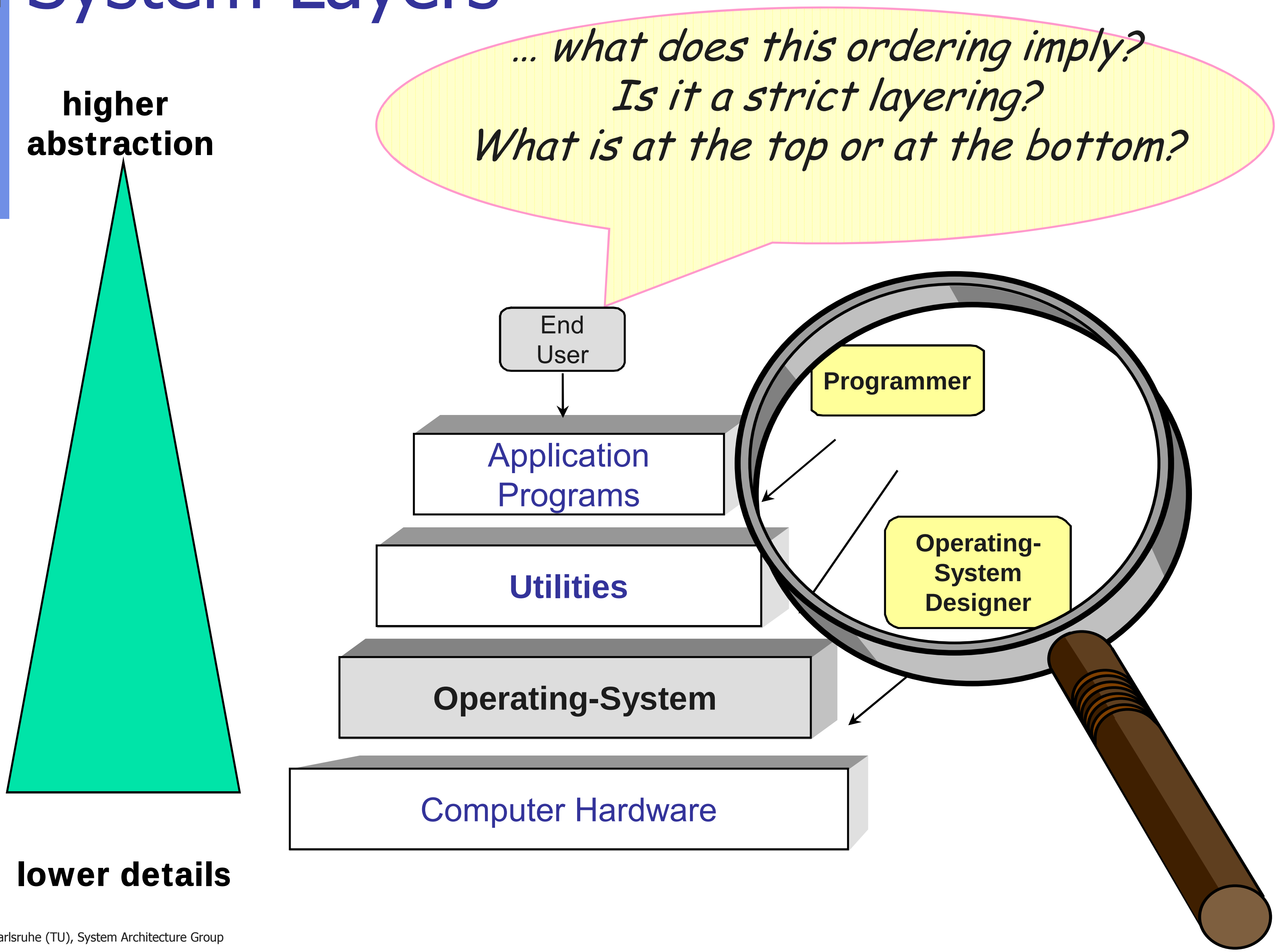
System Structure

Layered Systems

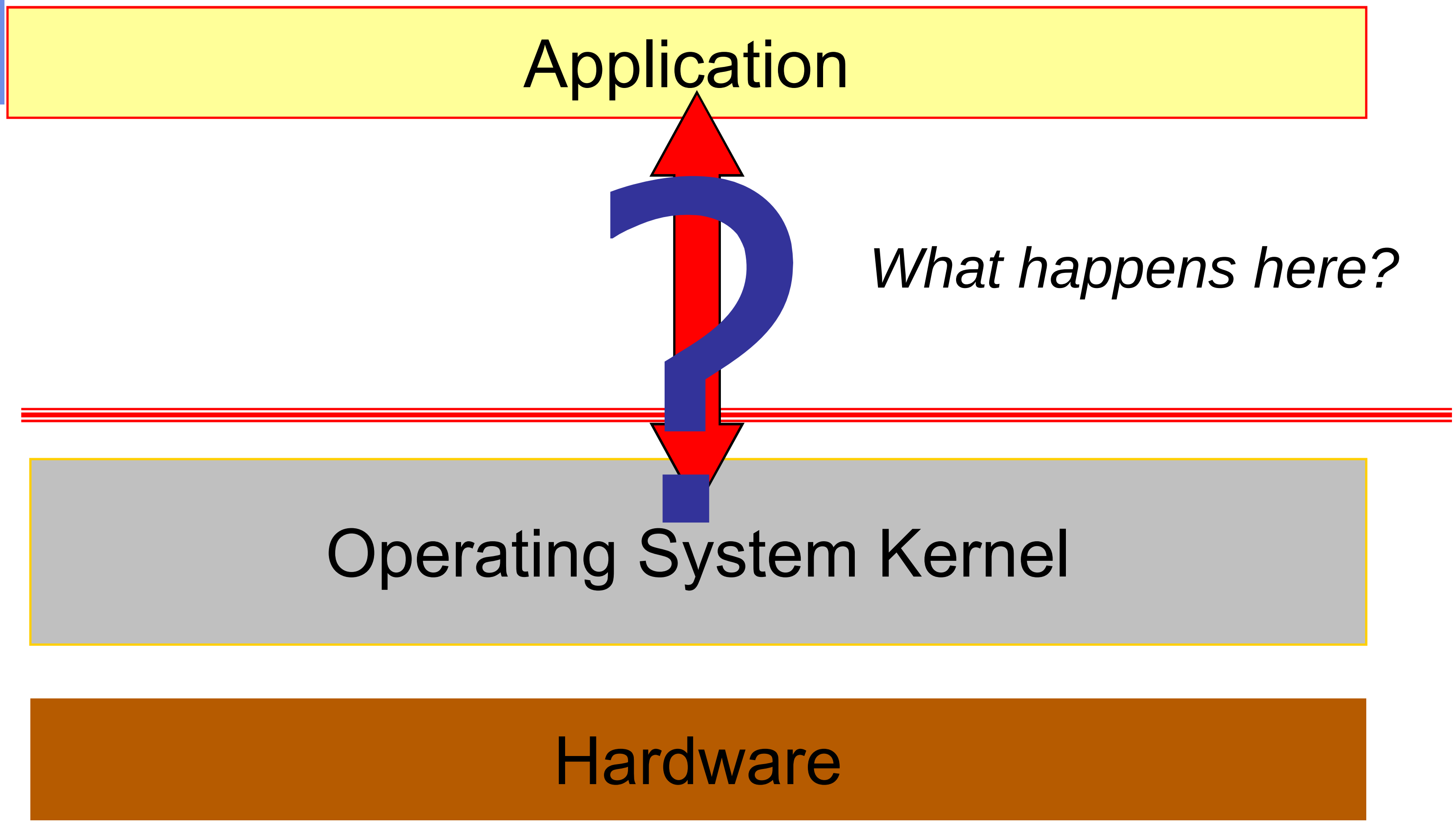
Privileged OS Kernel

System Interface

System Layers



Major System Components



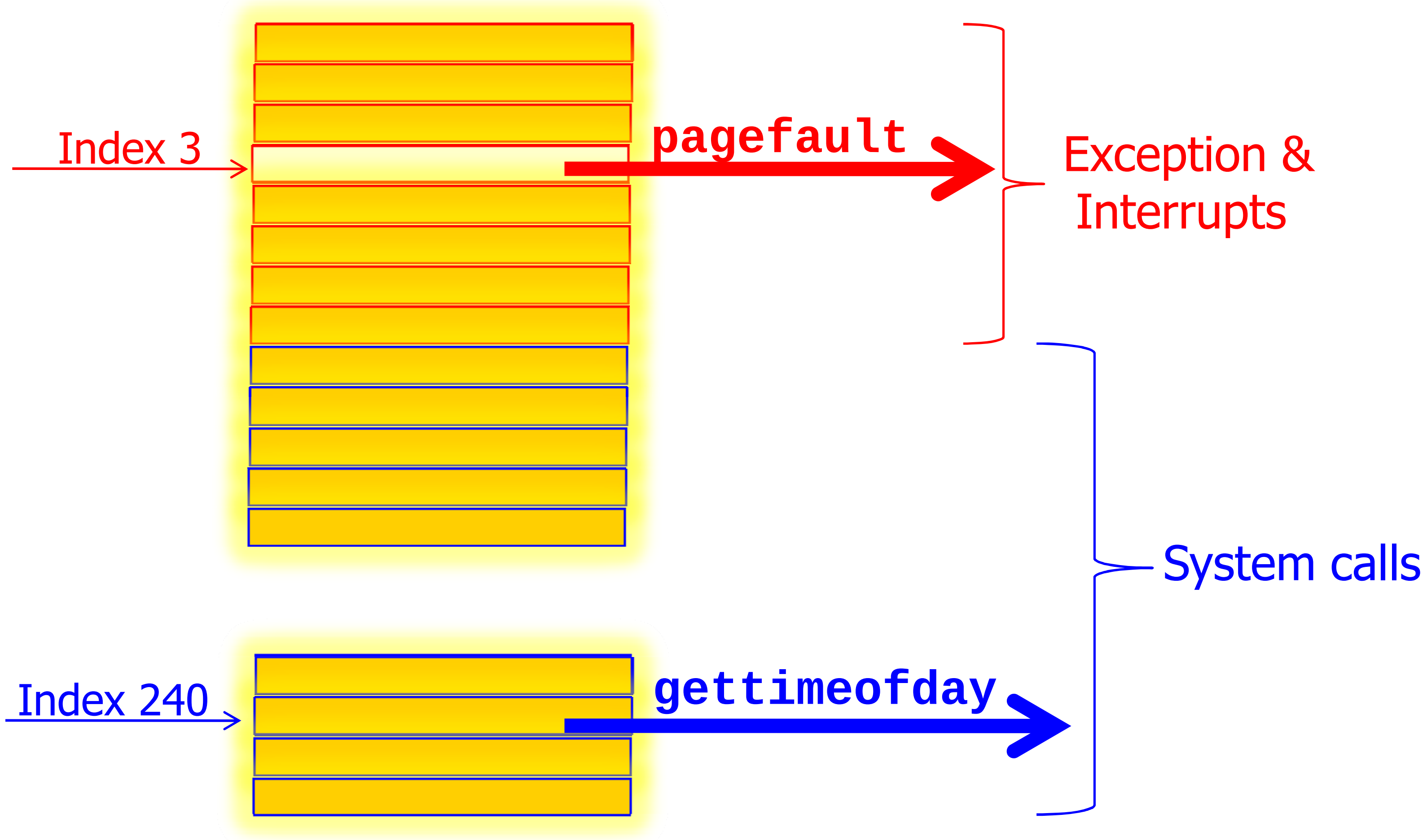
The Privileged OS (Kernel)

- Applications should not be able to bypass the OS (apart from the *non-privileged CPU instructions*)
  - OS can enforce the extended machine
  - OS can enforce its resource management
  - OS prevents applications from interfering with each other
- Some embedded OSes (e.g. PalmOS) do not have privileged components

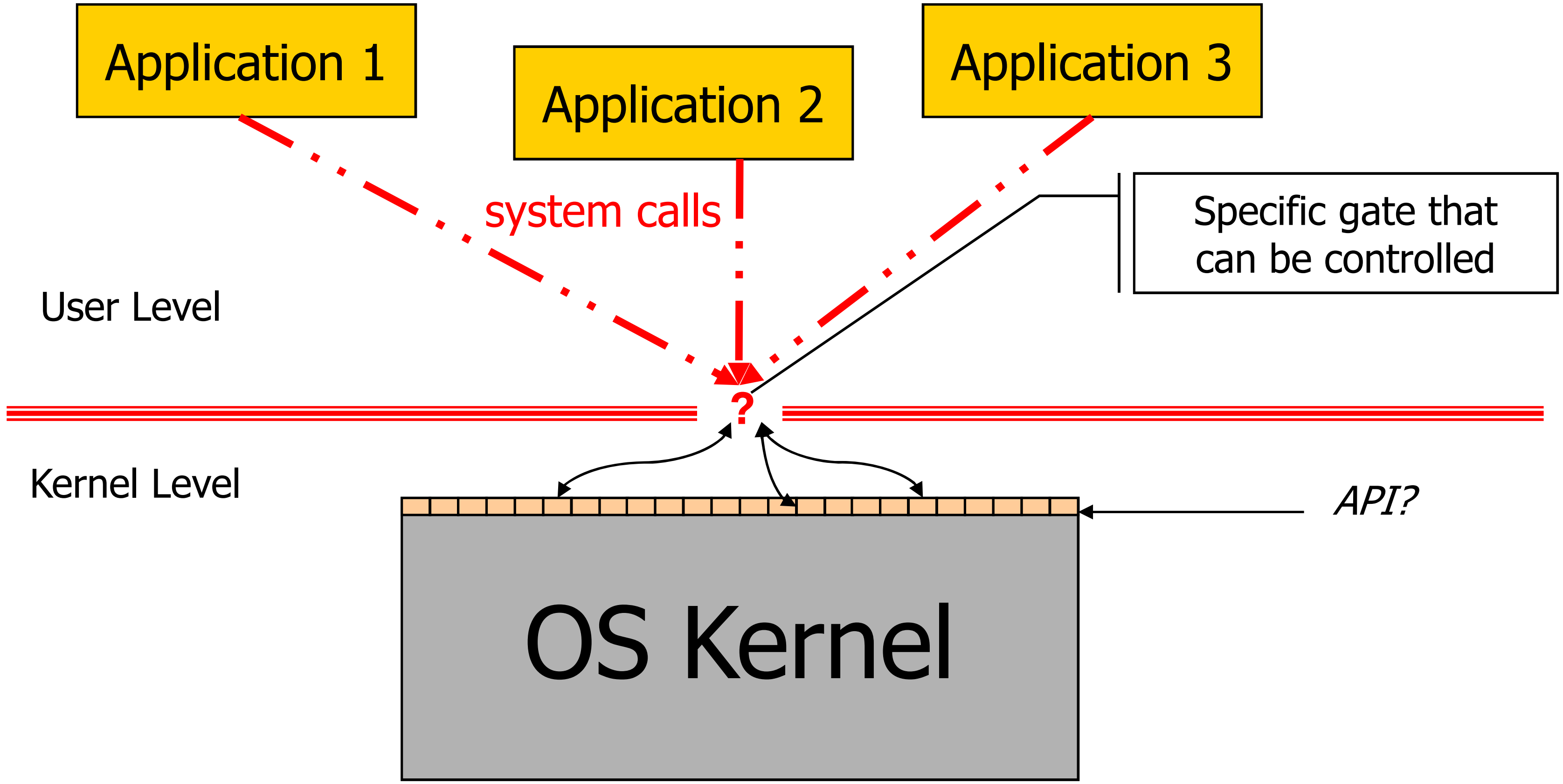
System Calls

- OS supplies its functionality via *system calls*
- System calls form a well defined interface (API) between applications and OS
  - Applications only need to know these system calls in order to get the requested service from the kernel
  - *How is a system call implemented?*
  - Via a specific, but non privileged instruction:
    - **`trap`**
    - **`int`**
- The trap instruction needs a specific parameter indicating the target IP within the kernel
  - To enable some control this parameter must be transferred within a predefined register

Interrupt Vector Table



OS as a Privileged Component



The System API is often hidden within a user level library, e.g. the Java API

Typical system calls?

Linux System Calls for Processes

Process Management	
Call	**Description**
`pid = fork()`	Create child process
`pid=waitpid(pid, &statloc, options)`	Wait for a child to terminate
`s = execve(name, argv, environp)`	Replace a process' core image
`exit(status)`	Terminate execution and return status

Linux System Calls for Files

File Management	
Call	**Description**
fd = open(file, how, …)	Open file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get the file's status information

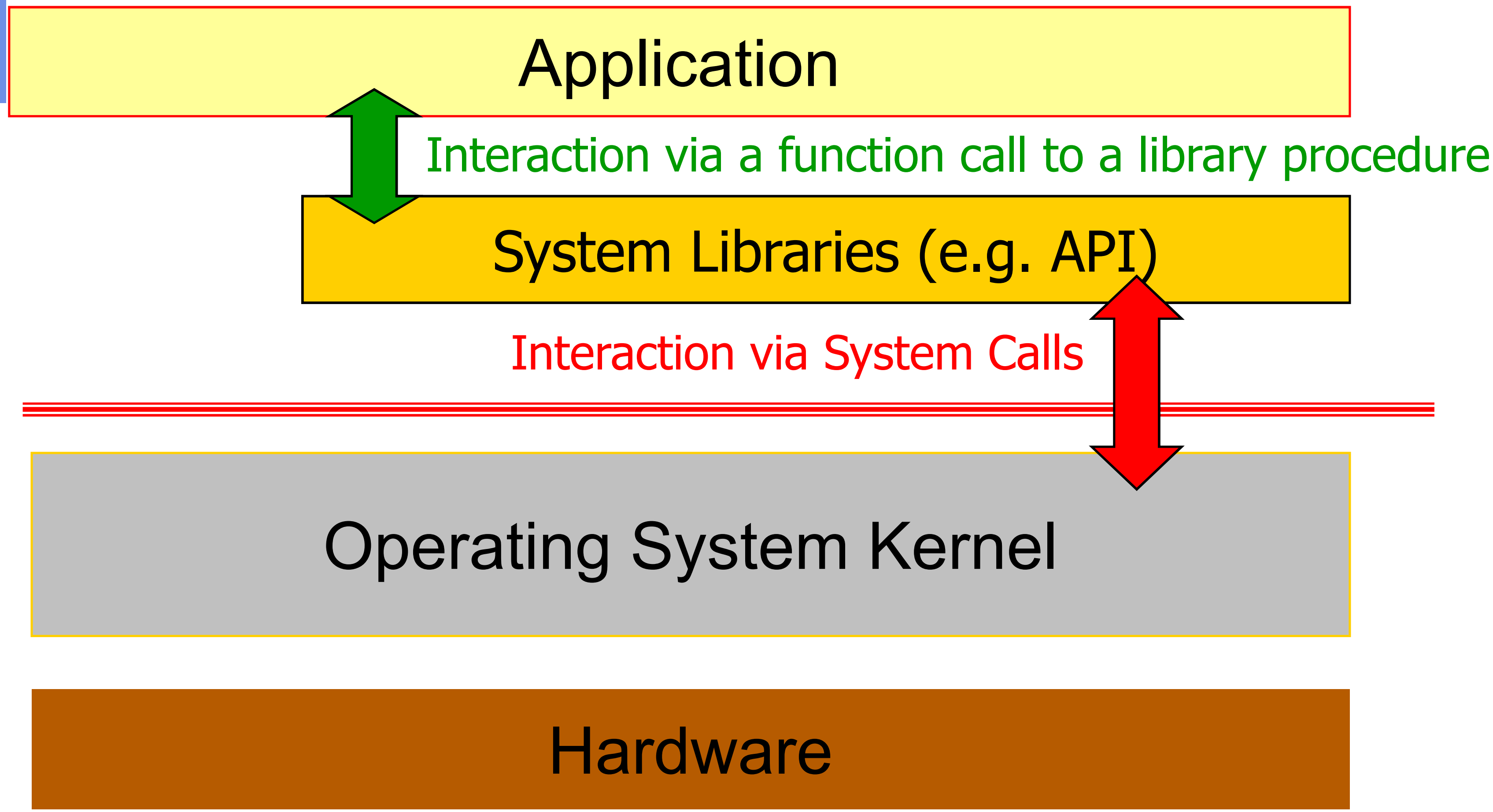
Linux System Calls for Directories

Directory Management	
Call	**Description**
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create new entry name2 → name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

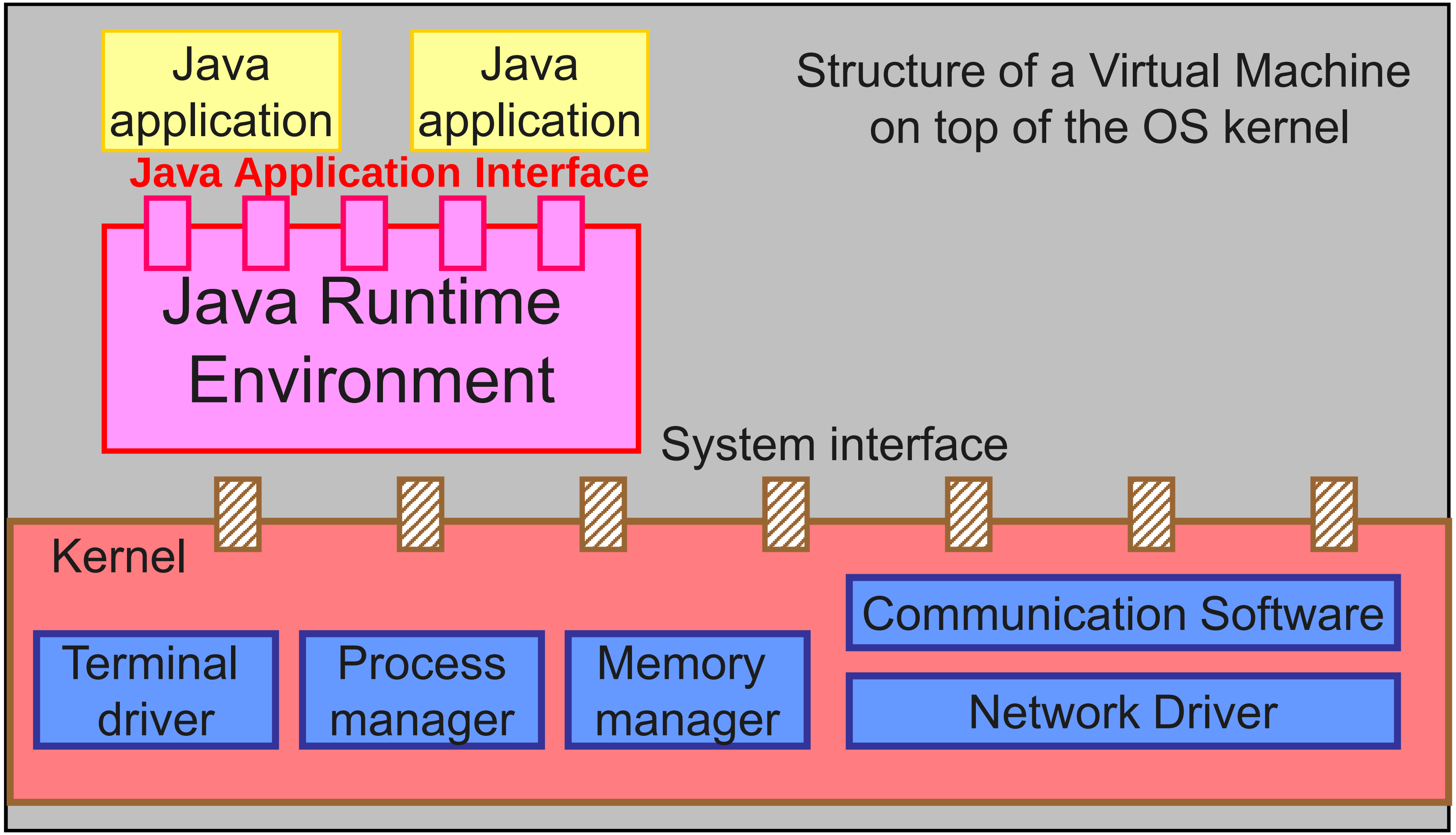
System Calls for Miscellaneous Tasks

Miscellaneous Management	
Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
`s = kill(pid, signal)`	Send a signal to a process
seconds = time(&seconds)	Get elapsed time since Jan. 1, 1970

Interdependencies



Nested Layered System Structure



Basic System Terms

Address Space,

Process, Thread, Task,

Thread Types

2 Main Abstractions within Systems

1. How to install „information processing“, i.e. activity “when“ to execute “what” code

   ⇒ activity , e.g. thread (process*)

2. How to install „protected code and data depositories“, i.e. ”where“ to store ”what” software entities

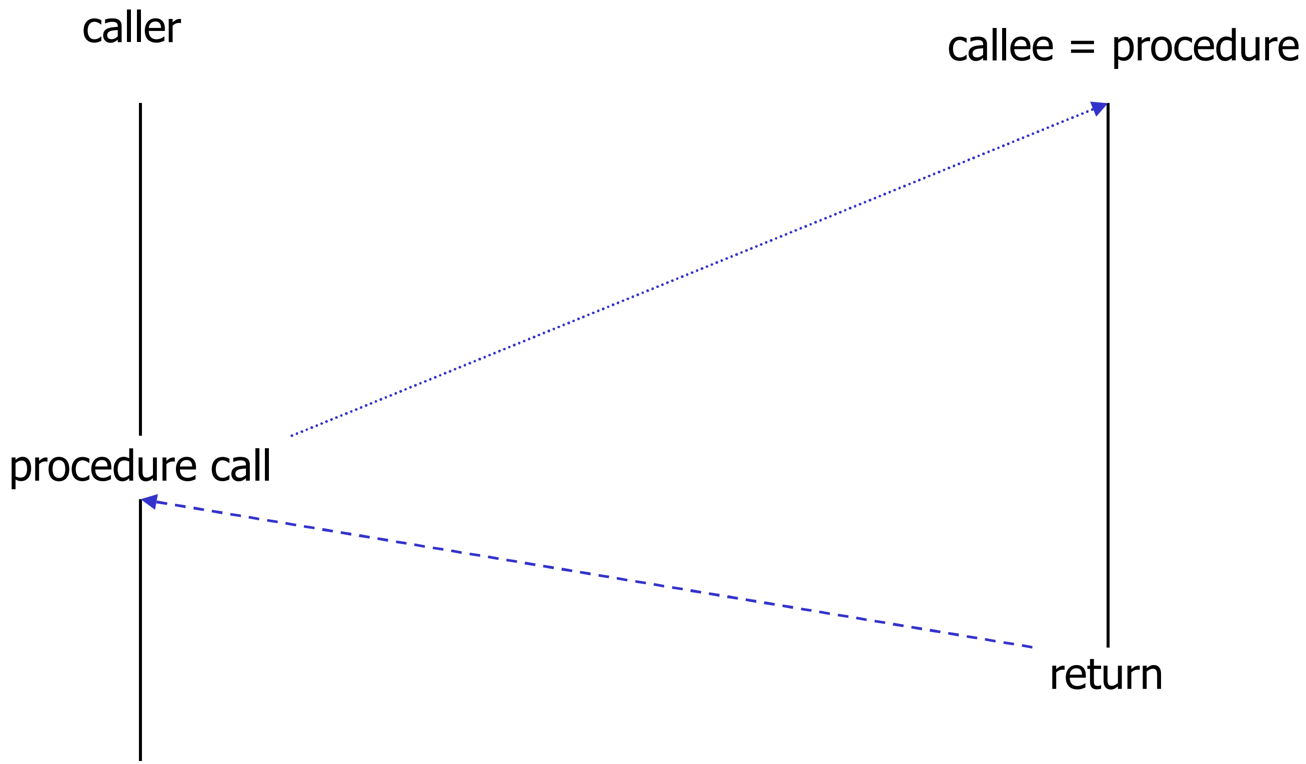
   ⇒ address space

*Note: Notion “process” ← “procedere” = “*voranschreiten*”
Notion “thread” ~ “*Faden abwickeln*”

Design Parameters for Address Spaces

- Number of data entities
- Boundary checks
- Types of buffers (stack, heap, file, …)
- Security of data entity (object, protection domain)
- Duration of data entity (volatile/temporary/persistent)
- An address space (AS) provides a *protected domain* for an activity, i.e. an executing program

Procedure



1. In most cases caller & callee belong to same AS

2. Either caller or callee are running

Why Processes/ Threads?

- Suppose your system offers a software tool, enhancing the way how you can edit, compile, and test your programs
- If this tool allows concurrent editing, compiling, and testing, ⇒ this tool could reduce your work a great deal

⇒ Processes/threads help to manage

concurrent activities

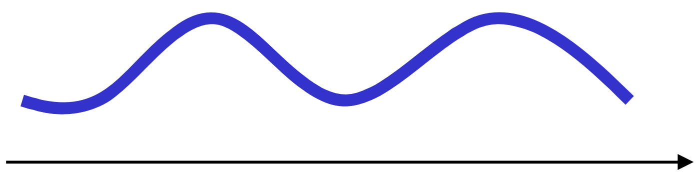
Design Parameters for Activities

- Number of activities
  - Static
  - Dynamic
- Types of activities
  - Foreground
  - Background
- Urgency of activities
  - Real time
    - Hard real time
    - Soft real time
  - Interactive
  - Batch
- Degree of interdependency
  - Isolated
  - Dependant
- ...

Dependant on these design parameters different activity models have been used

Thread

- Basic entity of pure activity
- Object of scheduling
  - Internal scheduling in the kernel
  - External scheduling in a runtime system



Basic characteristics of a thread?

Characteristics of Threads

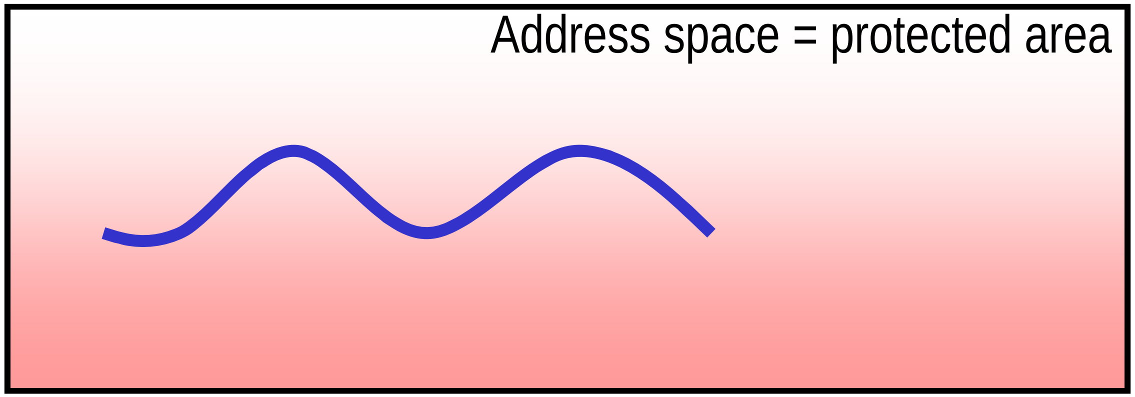
- Protected domain
  - The kernel address space is domain for **all** kernel threads
  - A user address space is domain for **all** threads of this **application,** i.e. each application has its own user address space
- Code
- Instruction pointer
- Stack
- Stack pointer
- Thread control block TCB

Additional Attributes of Threads

- Internal state (*context)*
- External *state* (running, ready, waiting, …)
- Priority
- Creation time
- Start time
- Deadline
- Waiting time
- Exit time

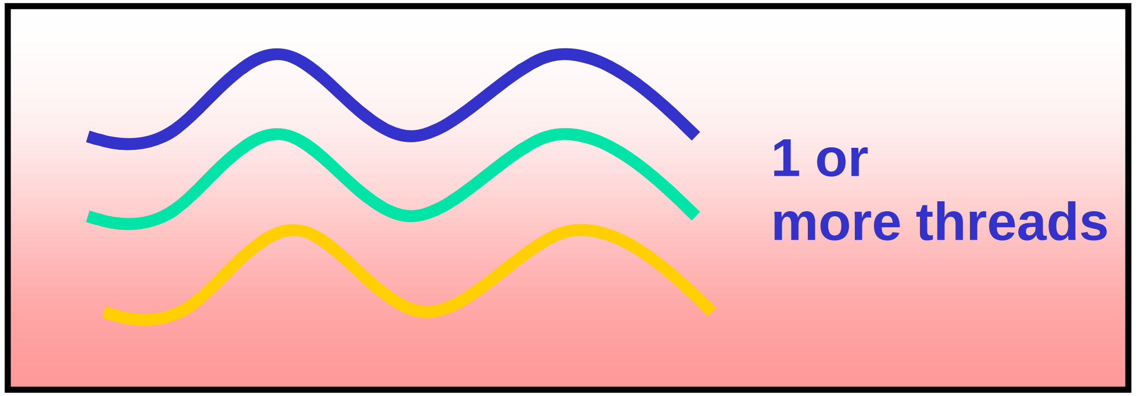
Process

- Single threaded
- Address space (Unix terminology)
- Additional resources



Task

- Entity of an "application" consisting of
  - t ≥ 1 thread(s)
  - Address space
  - Resources



The Activity Models

Process Mode

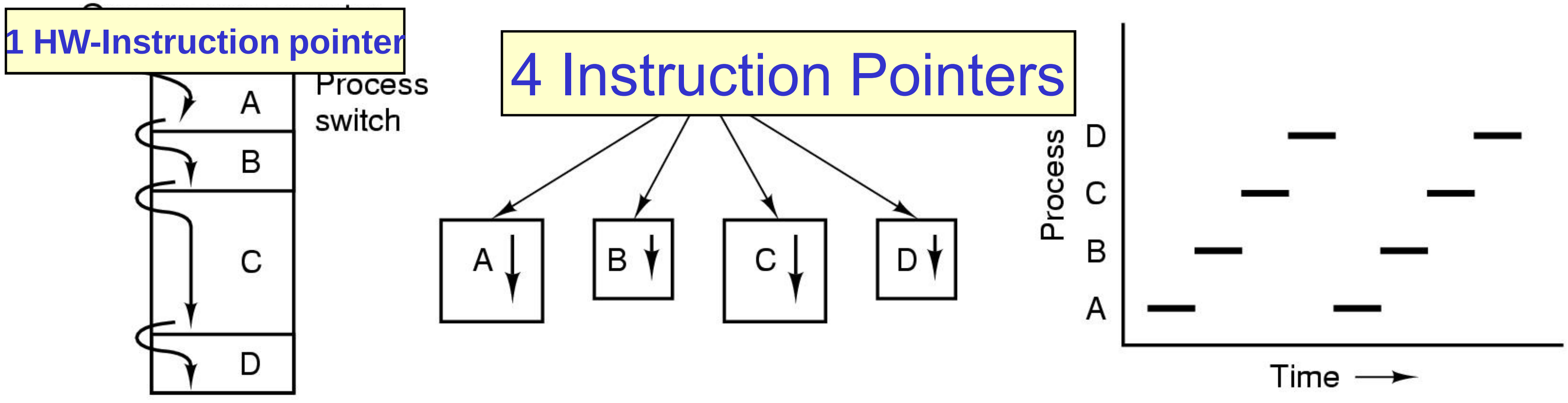
Procedure versus Thread

Process versus Task

Shared Memory

Java Threads

Process Model

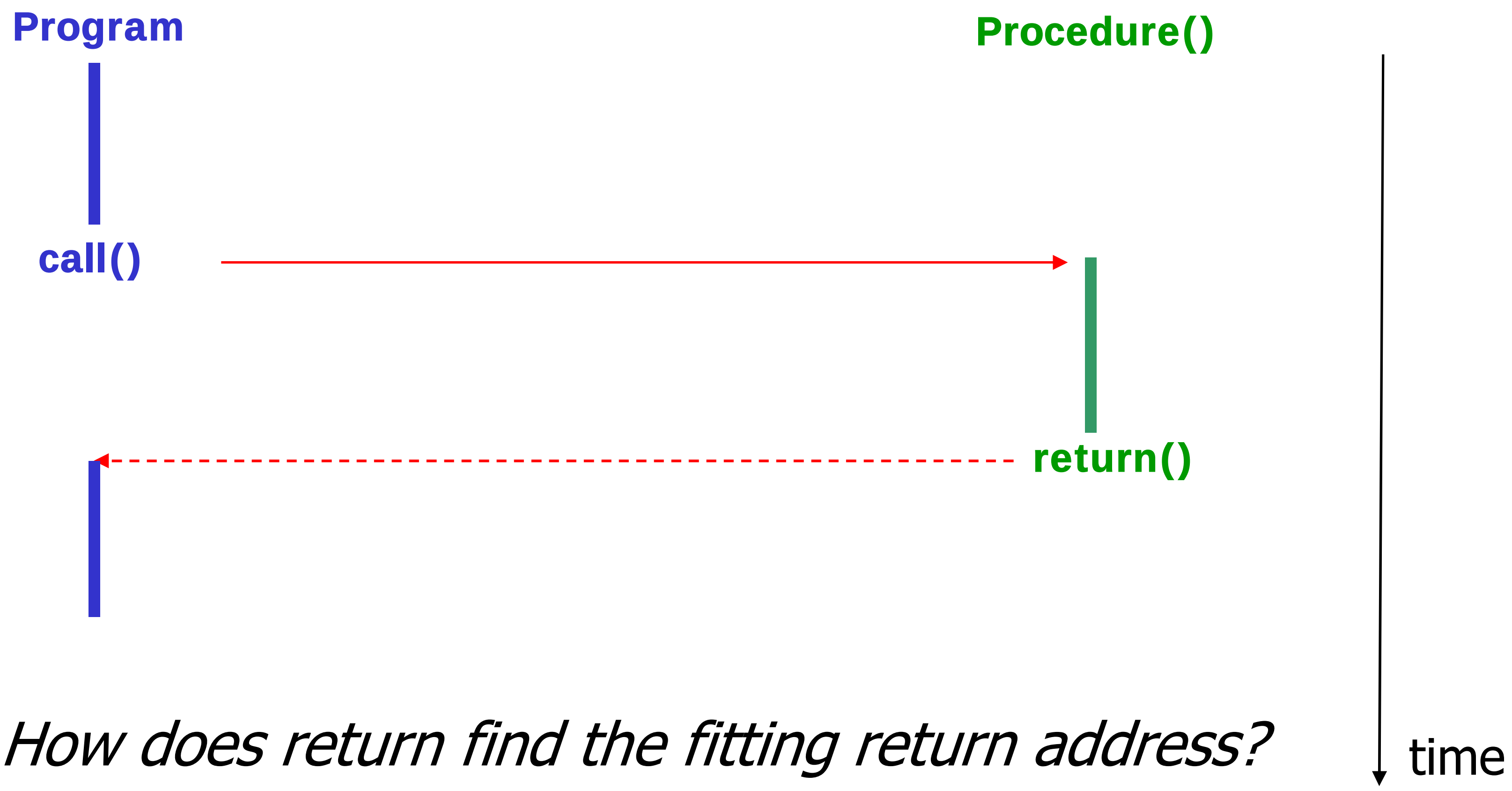


(a) (b) (c)

- Multiprogramming of 4 programs, each program is located in an extra address space
- Conceptually 4 independent, *sequential* processes
- However, on a single processor only one process is running at any instant

Procedure vs. Thread

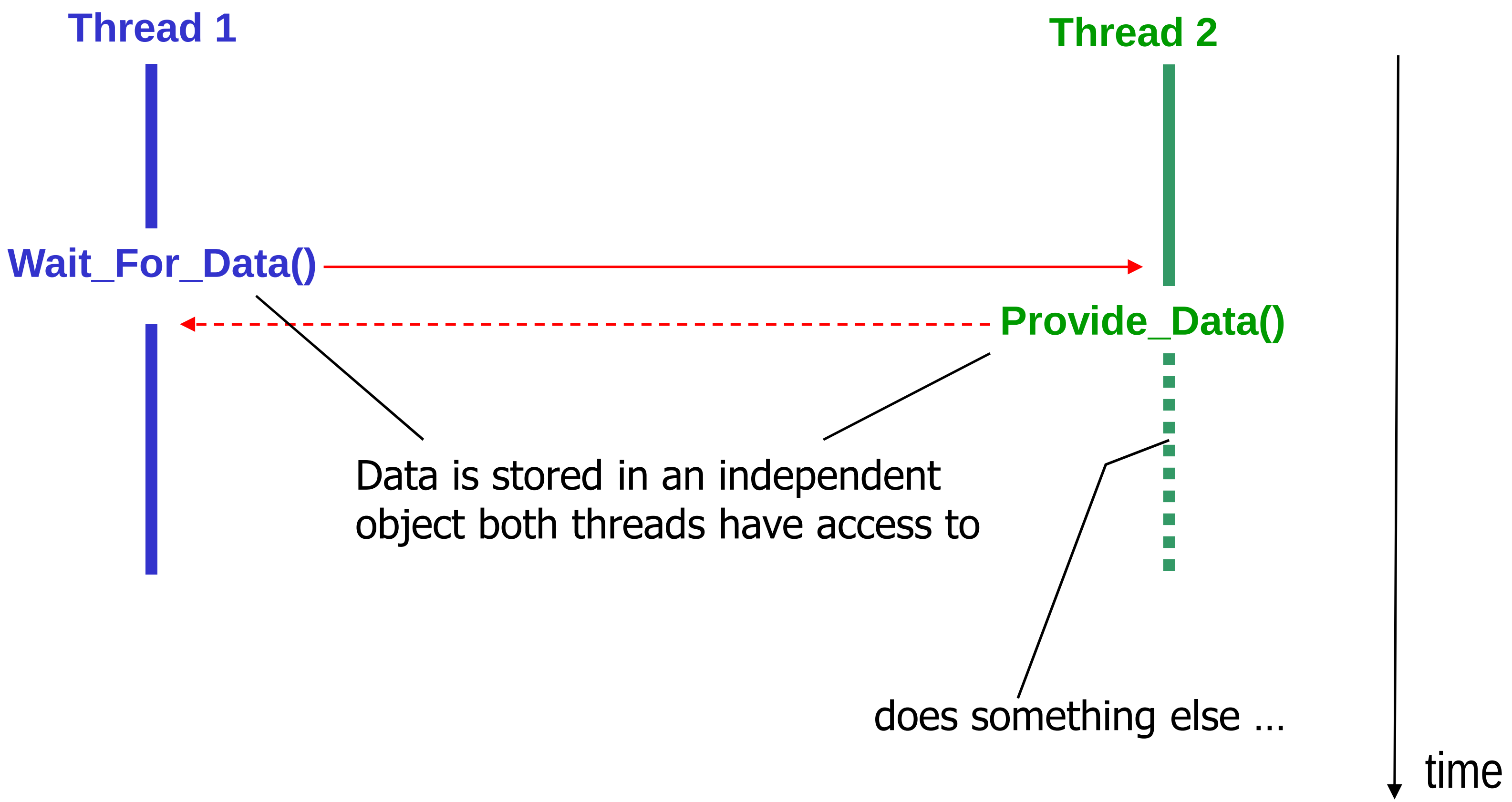
<u>Assumption:</u> Given program with a simple procedure call to compute data needed for the program to progress



How does return find the fitting return address?

Procedure vs. Thread

<u>Assumption:</u> Given program with two threads, one computes data that the other thread needs for its progress



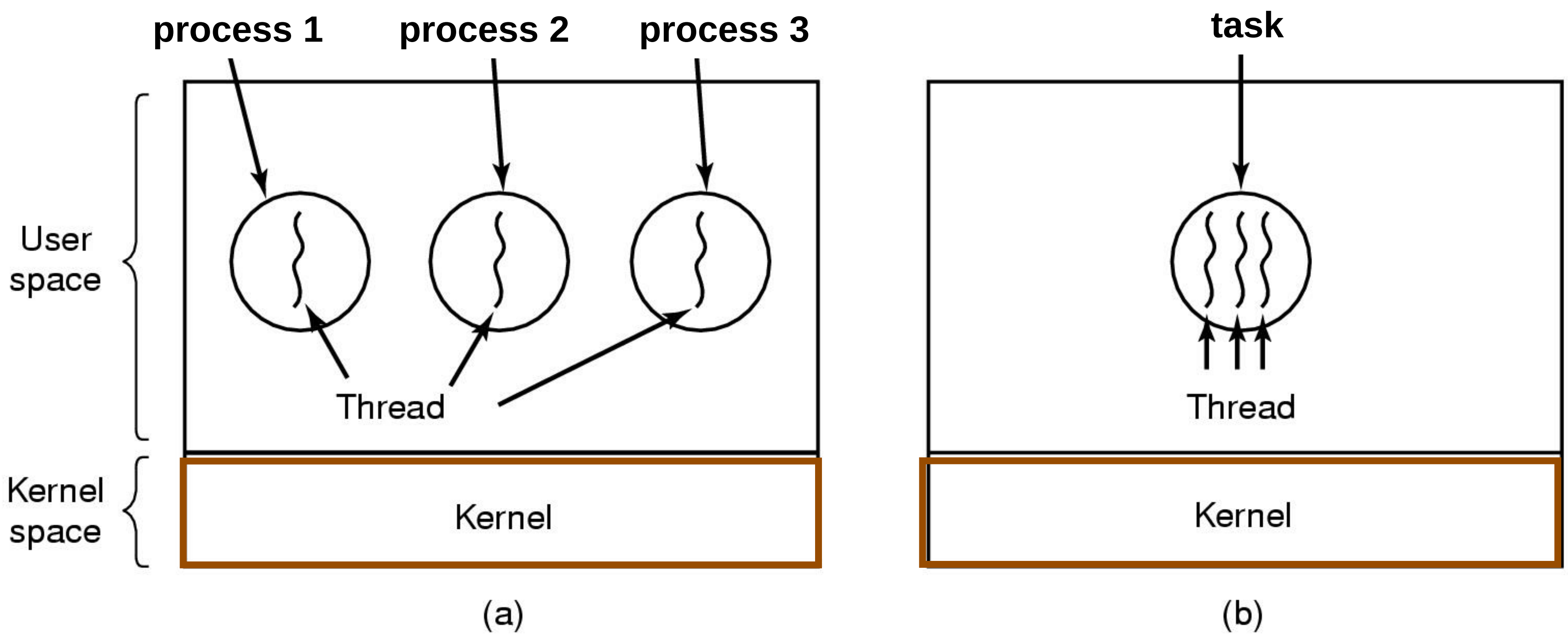
Thread

- Thread = abstraction for a pure activity
  (e.g. being executed on a CPU) ⇒
- Thread includes code and private data (e.g. a stack)
- A thread may also need some *environment*
  - Address space
  - Files, I/O-devices and other resources
  - It may even share this environment with other threads

Example: A file server may consist of ***t*** identical threads,
each thread serving only one client's request.

Process versus Task Model

Compare both models!
Pros and cons?



(a) Three processes (each task with only one thread)

(b) One task with three threads

Process versus Task

Process model

– create and delete need more
  – time
  – space, e.g.
    new address space

– Cooperation via IPC or shared memory (→)

+ well-separated from each other

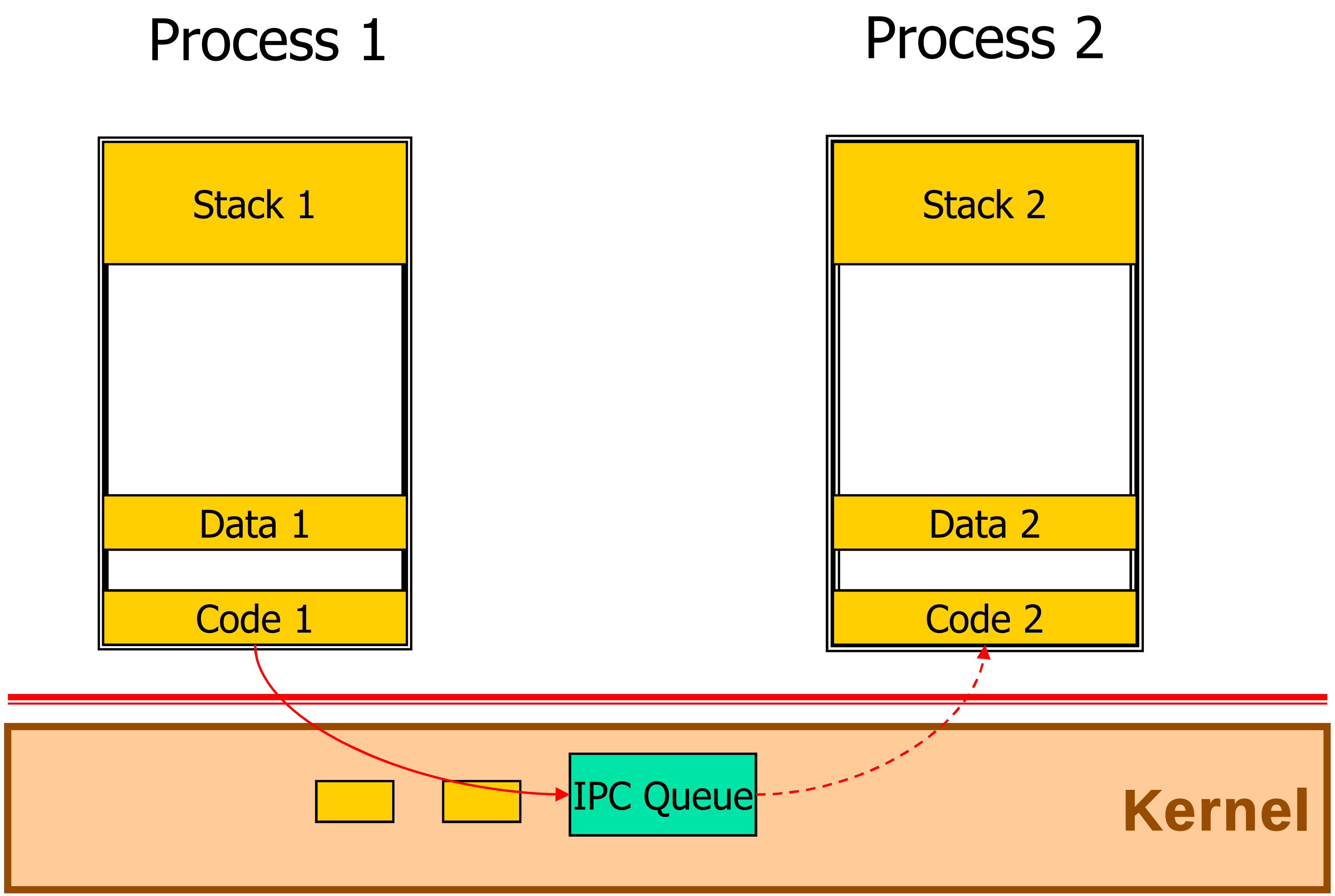
Thread model

– Might destroy each others data

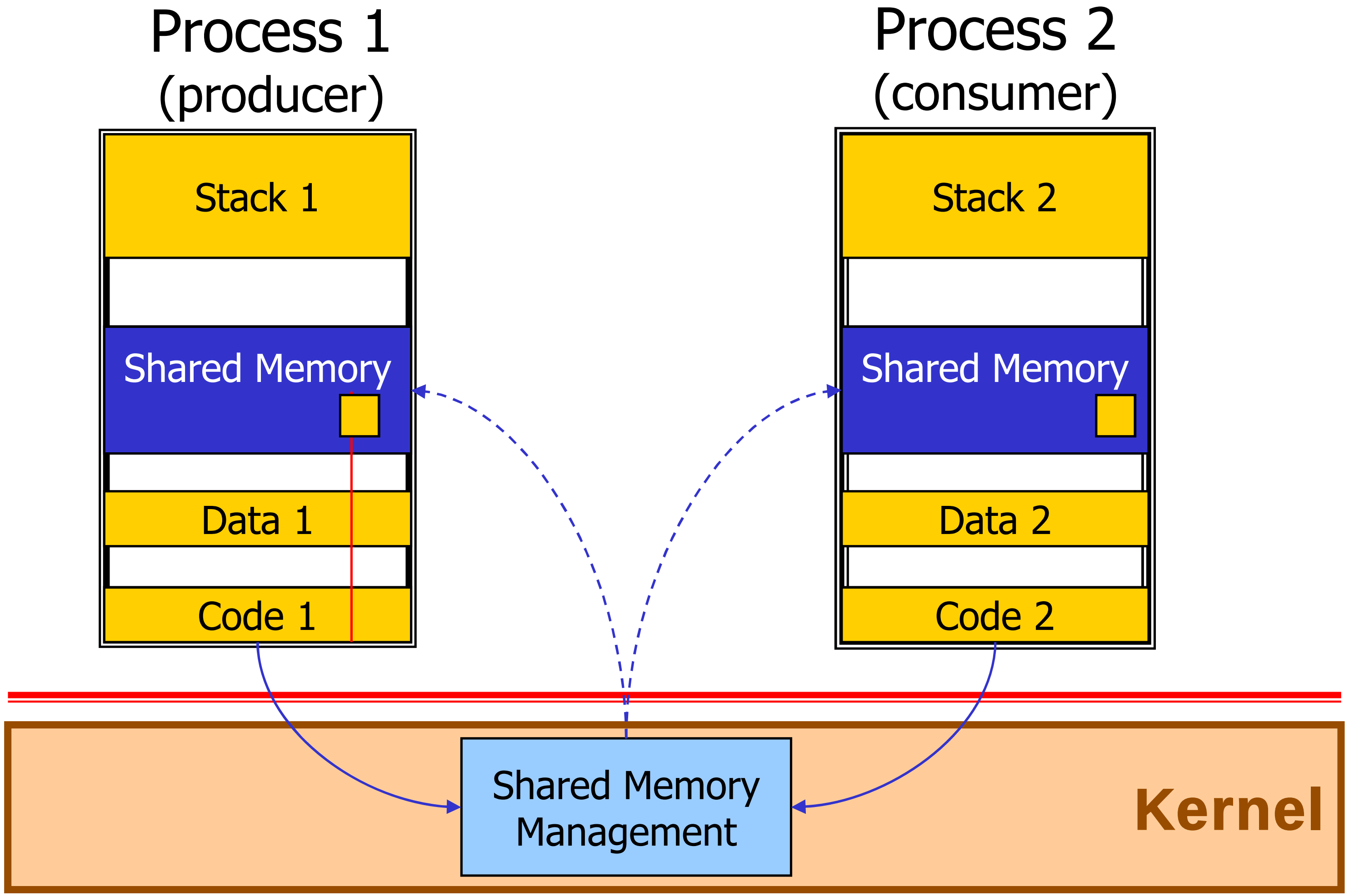
+ create and delete need less
  - time
  - space, e.g. only new
    - stack and TCB

+ easier to work together on common data

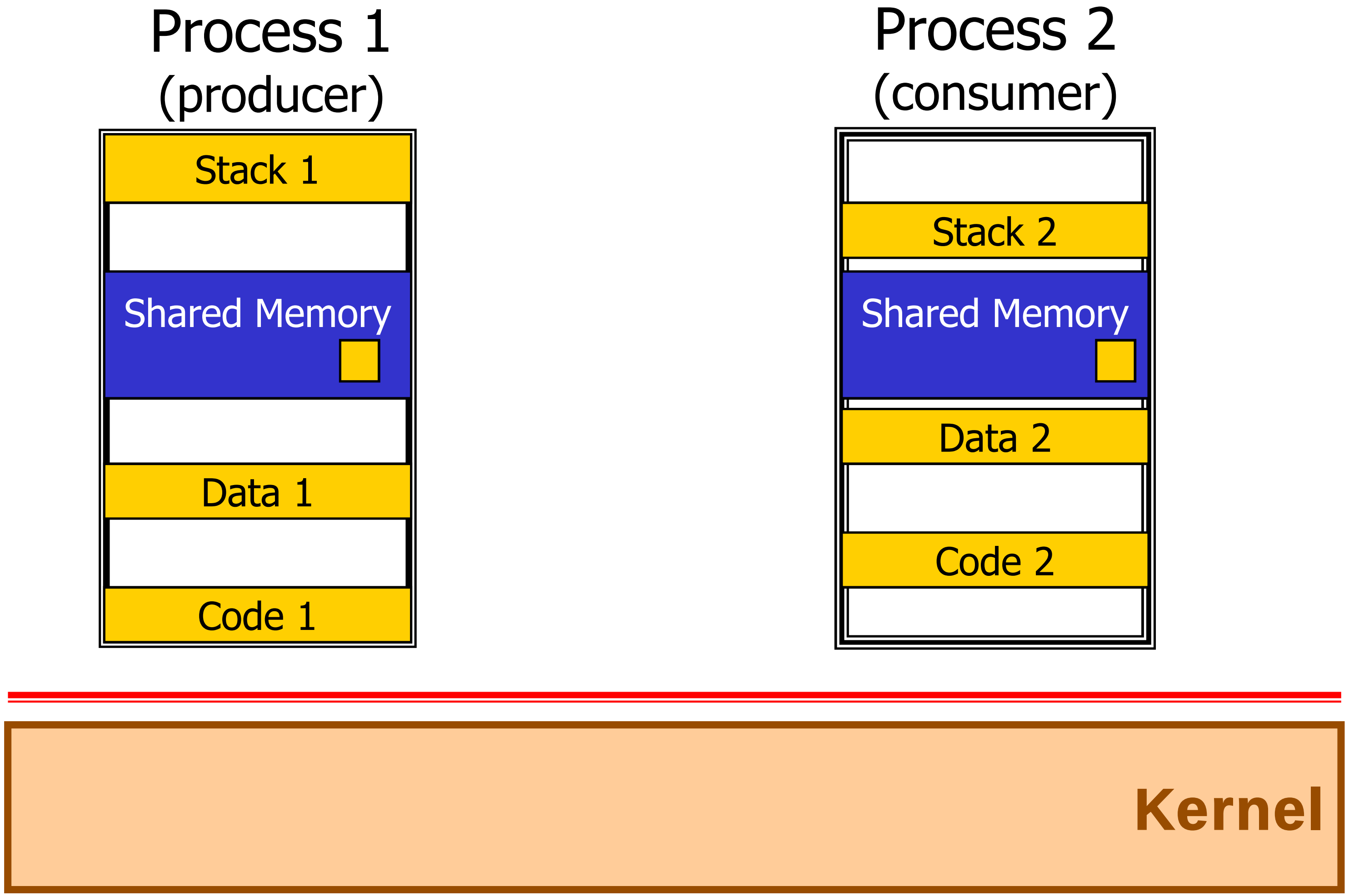
Shared Memory (0)



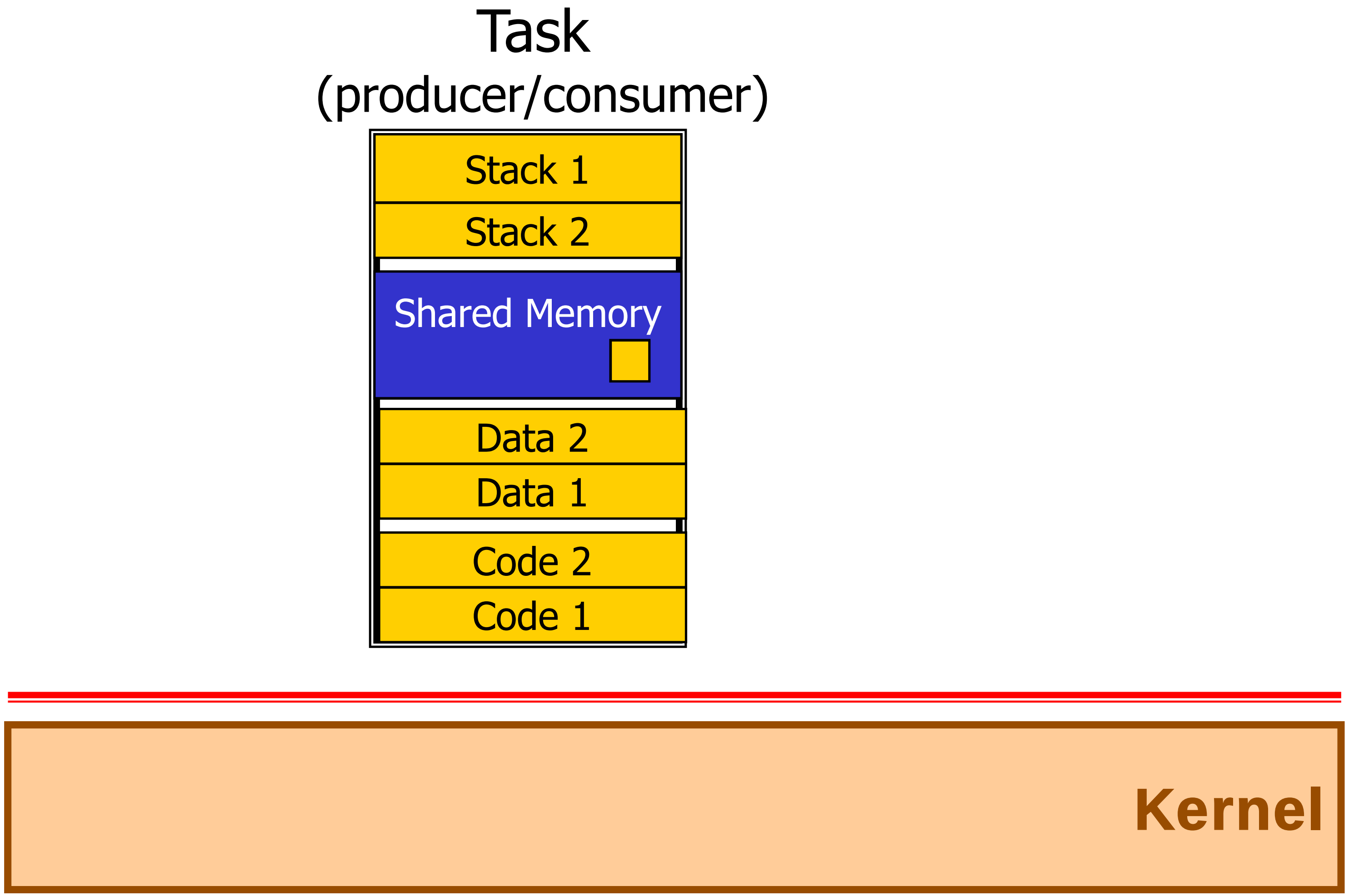
Shared Memory (1)



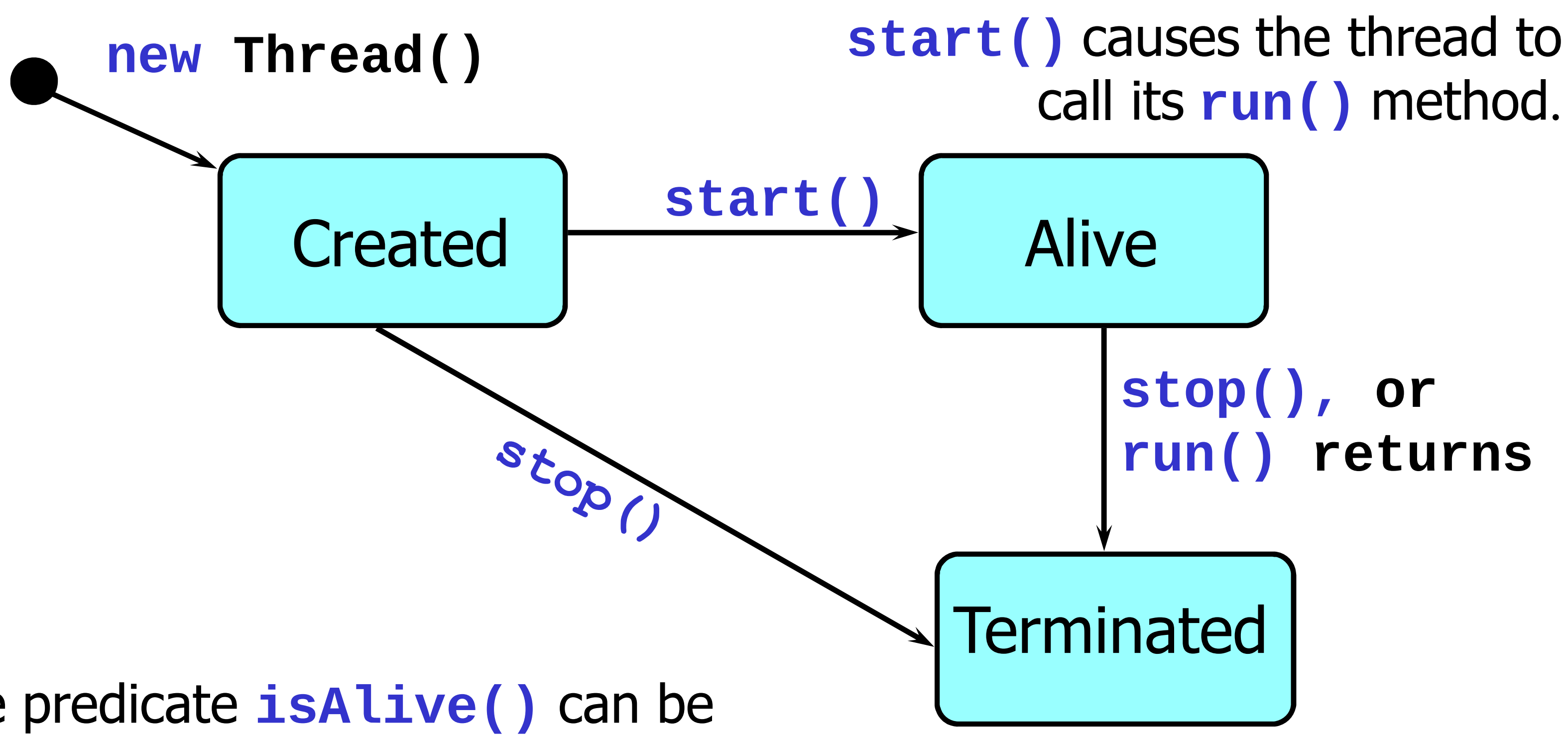
Shared Memory (2)



Shared Memory (3)



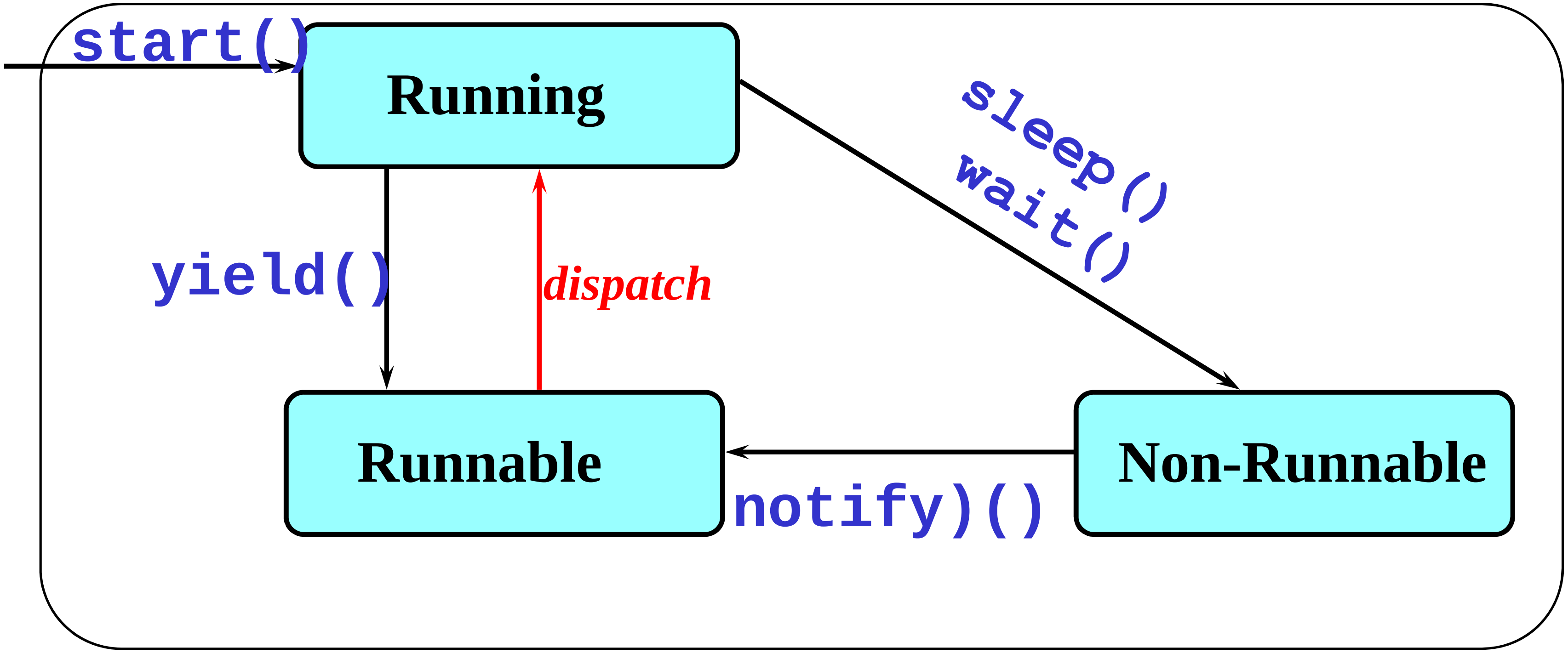
Thread Life-Cycle in Java



The predicate `isAlive()` can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted.

Thread Alive States in Java

Once started, an **alive** thread has a number of substates:



Thread Models

Pure User Level

Kernel Level

Hybrid

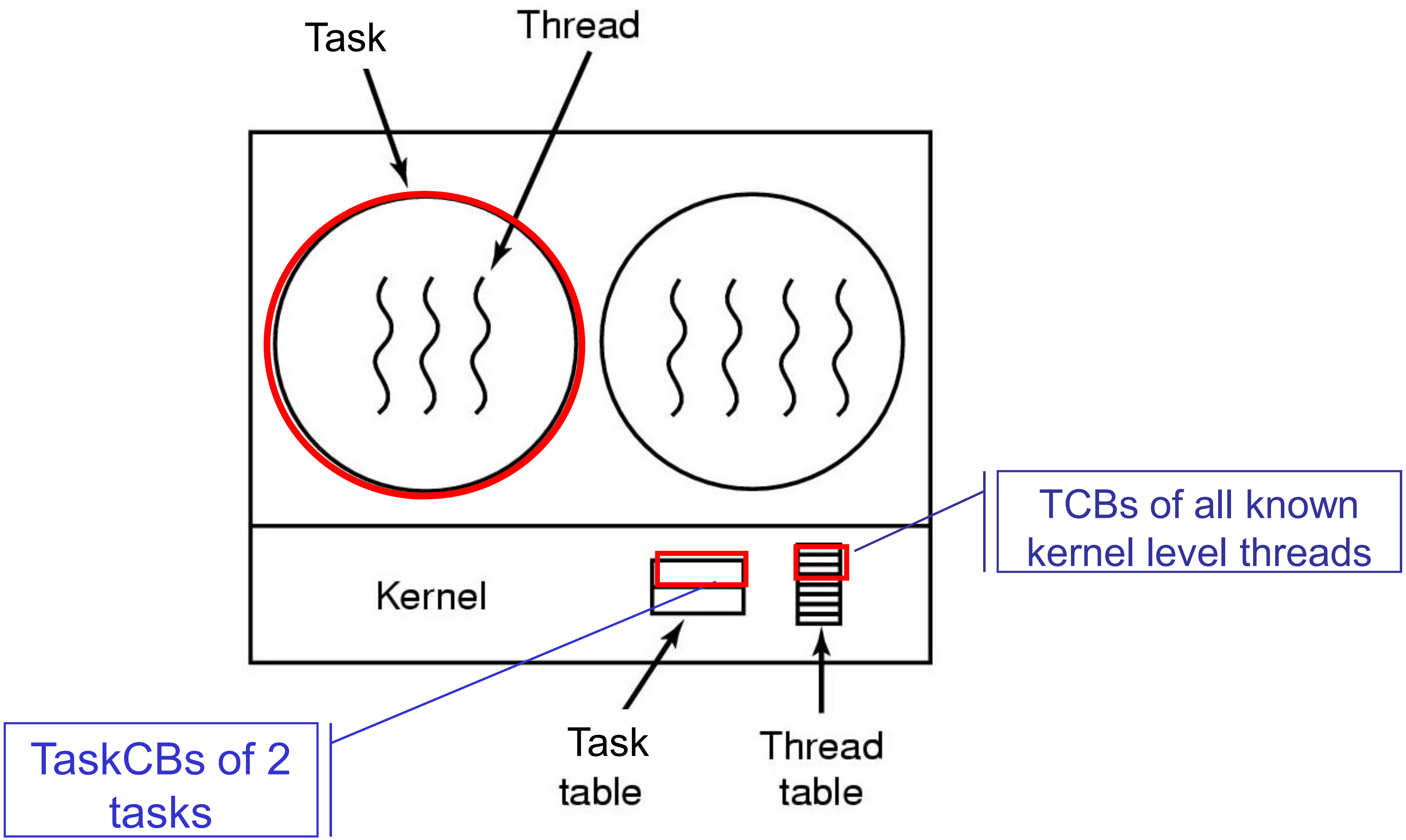
Types of Threads

- Kernel Level* Threads (KLT)
  - Known to the system wide thread management *implemented* inside the kernel, i.e. the corresponding TCBs are located inside the kernel

- User Level* Threads (PULT)
  - Known only within one task or one sub system, often implemented by a thread library, i.e. the corresponding TCBs are located inside an instance of the thread library, i.e in user-land

*This notion is KA-specific

Kernel Level Threads

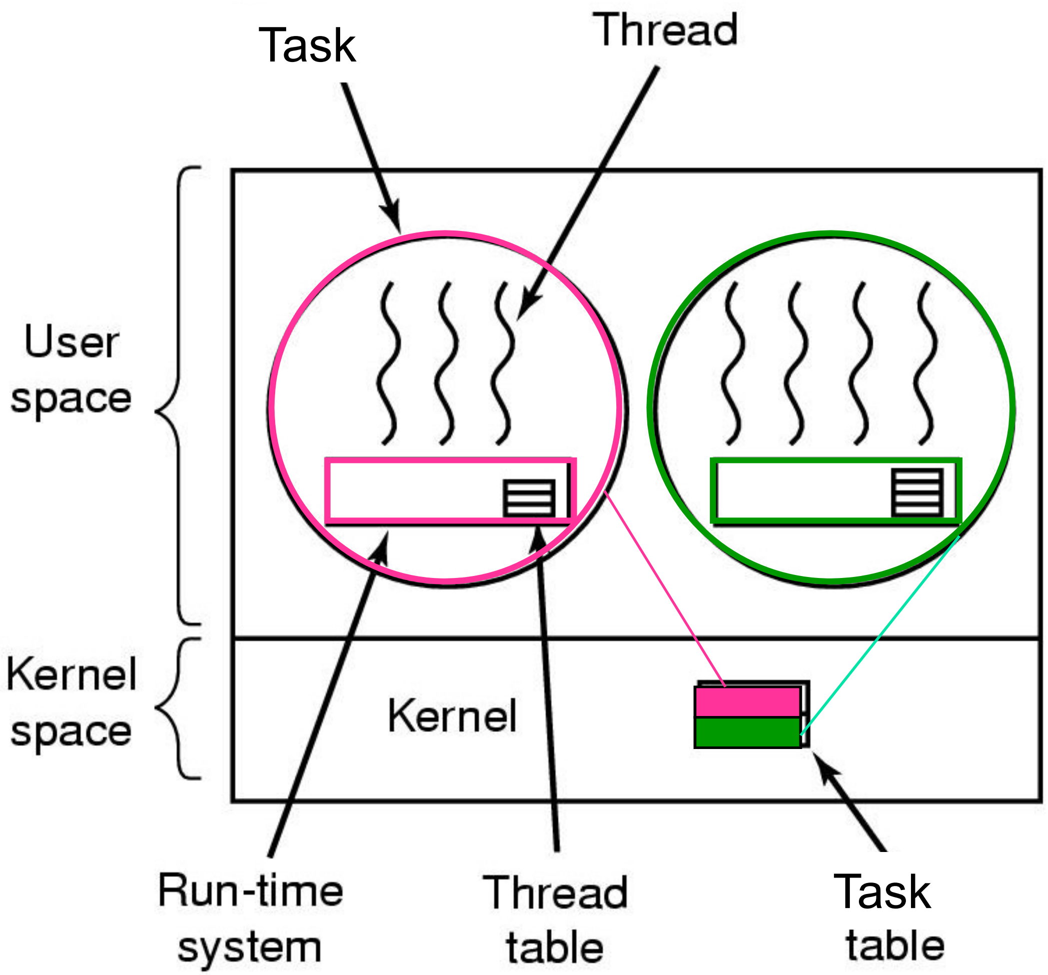


Kernel Level Threads

- Supported by the Kernel

- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

User Level Threads



User Level Threads

- Thread management done by user-level thread library

- Examples

  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*

Analysis of Kernel-Level Threads

Advantages:

Kernel can simultaneously schedule threads of same task on different processors

A *blocking* system call only blocks the calling thread, but no other thread from the same application

Inconveniences:

Thread switching within same task involves the kernel. We have 2 mode switches per thread switch!!

Discuss this very carefully

Analysis of User-Level Threads

Advantages

Thread switch does not involve the kernel: ⇒ no mode switching

Scheduling policy can be application specific: ⇒ best fitting policy

PULTs can run on any OS, if there is thread library

Inconveniences

Many system calls are blocking, ⇒ all threads of the task will be blocked

Kernel can only assign tasks to processors ⇒

2 pure user level threads of the same task can never run on two processors simultaneously

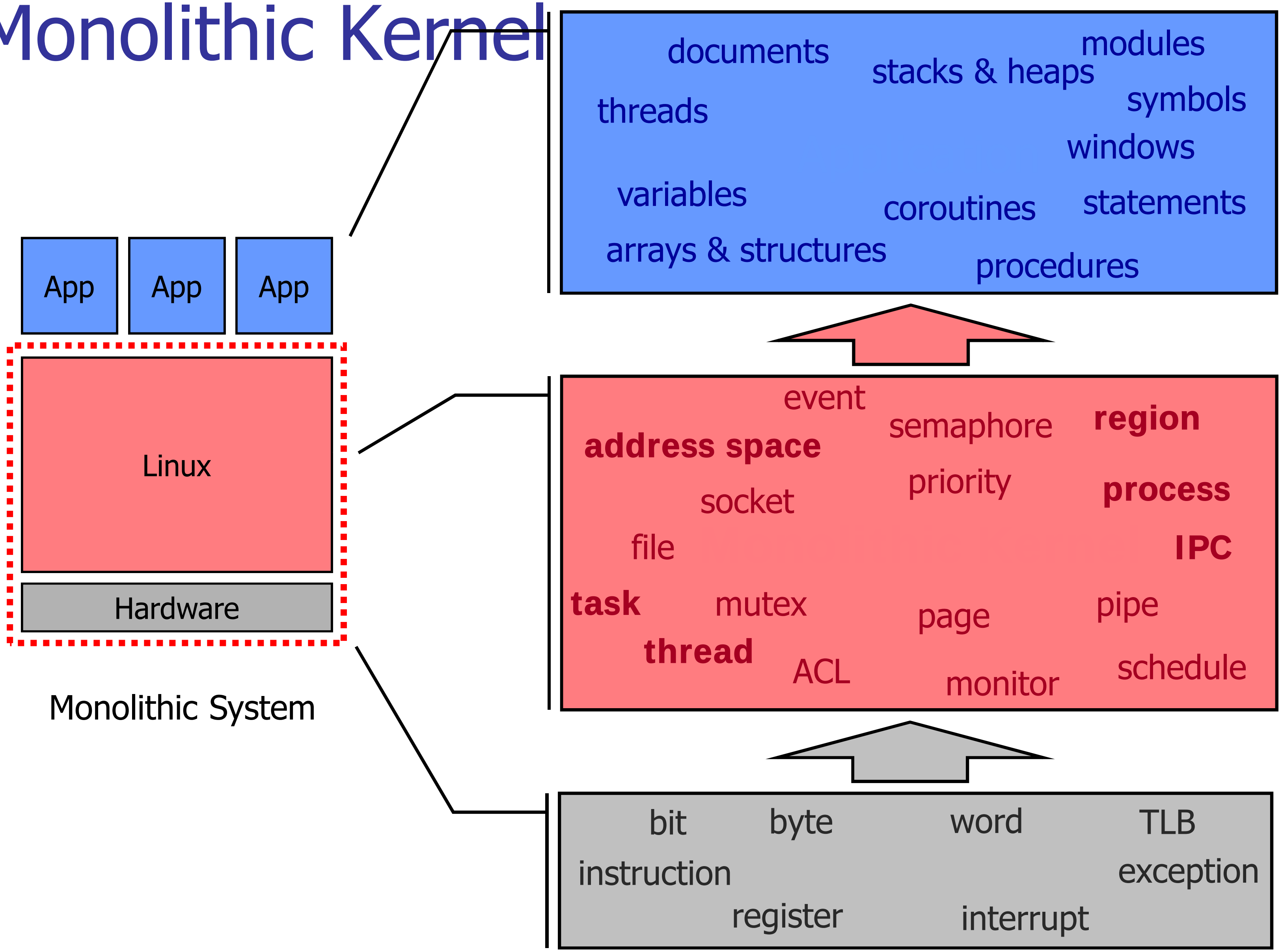
OS Kernels

What's Inside a Kernel?

Depends on the type of kernel

- Monolithic Kernel (traditional approach)
  - Lot of things, e.g.
    - File system
    - Network stack
    - Device Driver
    - Memory management
- Microkernel (our view)
  - Only what's needed
  - 2 major system abstraction + IPC mechanism

Monolithic Kernel



Monolithic System

Microkernel