Lecture

Operating System

6. Mechanism: Limited Direct Execution

Professor Dr. Michael Mächtel

6. Mechanism: Limited Direct Execution

1. **Goal**
2. **Direct Execution**
3. **Limited Direct Execution**
4. **Context Switch**

6. Mechanism: Limited Direct Execution

1. **Goal**
2. **Direct Execution**
3. **Limited Direct Execution**
4. **Context Switch**

Virtualization of CPU by Time Share

- Goal:
  - Give each process impression it alone is actively using CPU
  - Resources can be shared in **time** and **space**
    - time: time-sharing the CPU
    - space: e.g. memory, disc sharing (later)
  - The OS needs to share the physical CPU by **time sharing**.
    - Issue:
      - *Performance*: How can we implement virtualization without adding excessive overhead to the system?
      - *Control*: How can we run processes efficiently while retaining control over the CPU?

6. Mechanism: Limited Direct Execution

1. Goal
2. **Direct Execution**
3. Limited Direct Execution
4. Context Switch

Direct Execution

- Just run the program directly on the CPU.

OS	Program
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` / `argv` 5. Clear registers 6. Execute call `main()`	
	7. Run `main()` 8. Execute return from `main()`
9. Free memory of process 10. Remove from process list	

Without *limits* on running programs, the OS wouldn't be in control of anything and thus would be "just a library"

Problems of Direct Execution?

1. What if a process wishes to perform some kind of **restricted operation** such as …
   - Issuing an I/O request to a disk
   - Gaining access to more system resources such as CPU or memory
2. How can the OS **regain control** of the CPU so that it can switch between processes?
   - A cooperative Approach: Wait for system calls
   - A Non-Cooperative Approach: The OS takes control

Problem 1: Restricted Operation

- Solution: Using **protected control transfer**
  - **User mode:** Applications do not have full access to hardware resources.
  - **Kernel mode:** The OS has access to the full resources of the machine
- Switch from User mode to Kernel mode: **System Call**
  - Allow the kernel to carefully expose certain key pieces of functionality to user program, such as …
    - Accessing the file system
    - Creating and destroying processes
    - Communicating with other processes
    - Allocating more memory

System Call

- **Trap** instruction
  - Jump into the kernel
  - Raise the privilege level to kernel mode
- **Return-from-trap** instruction
  - Return into the calling user program
  - Reduce the privilege level back to user mode

6. Mechanism: Limited Direct Execution

1. Goal
2. Direct Execution
3. **Limited Direct Execution**
4. Context Switch

Limited Direction Execution Protocol

OS@boot (Kernel Mode)	Hardware	
Initialize trap table		
	remember address of syscall handler	
…	…	…
OS@run (Kernel Mode)	**Hardware**	**Program (User Mode)**
• Create entry for process list • Allocate memory for program • Load program into memory • Setup user stack with `argv` • Fill kernel stack with reg/PC • **return-from -trap**		
	• restore regs from kernel stack • move to user mode • jump to main	
		• Run main() • … • Call **system trap** into OS

—- —- —- *(Cont.)* —- —- —-

Limited Direction Execution Protocol

OS@run (Kernel Mode)	Hardware	Program (User Mode)
	—- —- —- *(Cont.)* —- —- —-	
	• save regs to kernel stack • move to kernel mode • jump to trap handler	
• Handle trap • Do work of syscall • **return-from-trap**		
	• restore regs from kernel stack • move to user mode • jump to PC after trap	
		• … • return from main • trap (via `exit()`)
• Free memory of process • Remove from process list		

Problem 2: Switching Between Processes

- How can the OS **regain control** of the CPU so that it can switch between processes?
  - A cooperative Approach: **Wait for system calls**
  - A Non-Cooperative Approach: **The OS takes control**

Wait for system calls

- Processes **periodically give up the CPU** by making system calls such as `yield`.
  - The OS decides to run some other task.
  - Application also transfer control to the OS when they do something illegal.
    - Divide by zero
    - Try to access memory that it shouldn't be able to access
  - Example:
    - Early versions of the Macintosh OS, The old Xerox Alto system

A process gets stuck in an infinite loop.
→ Reboot the machine

OS Takes Control

- **A timer interrupt**
  - During the boot sequence, the OS start the timer.
  - The timer **raise an interrupt** every so many milliseconds.
  - When the interrupt is raised :
    - The currently running process is halted.
    - Save enough of the state of the program
    - A pre-configured interrupt handler in the OS runs.

A timer interrupt gives OS the ability to run again on a CPU.

Saving and Restoring Context

- **Scheduler** makes a decision:
    - Whether to continue running the **current process**, or switch to a **different one**.
    - If the decision is made to switch, the OS executes **context switch.**

6. Mechanism: Limited Direct Execution

1. Goal
2. Direct Execution
3. Limited Direct Execution
4. **Context Switch**

Context Switch

- A low-level piece of assembly code
  - **Save a few register values** for the current process onto its kernel stack
    - General purpose registers
    - PC
    - kernel stack pointer
- **Restore a few register values** for the *soon-to-be-executing* process from its kernel stack
- **Switch to the kernel stack** for the *soon-to-be-executing* process

Limited Direction Execution Protocol

Timer Interrupt

OS@boot (Kernel Mode)	Hardware	
Initialize trap table		
	• remember address of syscall handler and timer handler	
start interrupt timer		
	• start timer • interrupt CPU in X ms	
…	…	…

Limited Direction Execution Protocol

Timer Interrupt

OS@run (Kernel Mode)	Hardware	Program (User Mode)
		Process A ....
	• **timer interrupt** • save regs(A) to k-stack(A) • move to kernel mode • jump to trap handler	
• Handle the trap • Call switch() routine • save regs(A) to proc-struct(A) • restore regs(B) from proc-struct(B) • switch to k-stack(B) • **return-from-trap (into B)**		
	• restore regs(B) from k-stack(B) • move to user mode • jump to B's PC	
		Process B ...

2 Stacks per Process

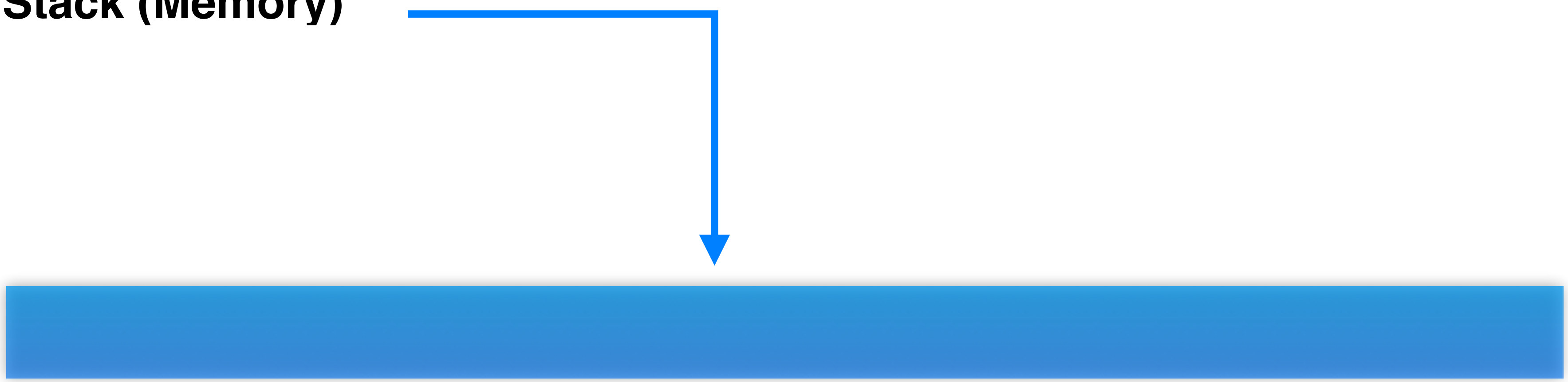
- **Every** process has 2 different stacks
  - **User stack**
    - User code is executed using the normal ‘stack’ called user stack
  - **Kernel stack**
    - If a trap to a systemcall is done, the code of the OS runs.
    - This OS code is executed using the kernel stack

Example Context Switch

- The Algorithmus, that implements the low level work for a context switch is often called the **Dispatcher**.
- We start at: Running in user mode, SP points to user stack

Representation of Kernel Stack (Memory)

SP -> UserStack



Example Context Switch

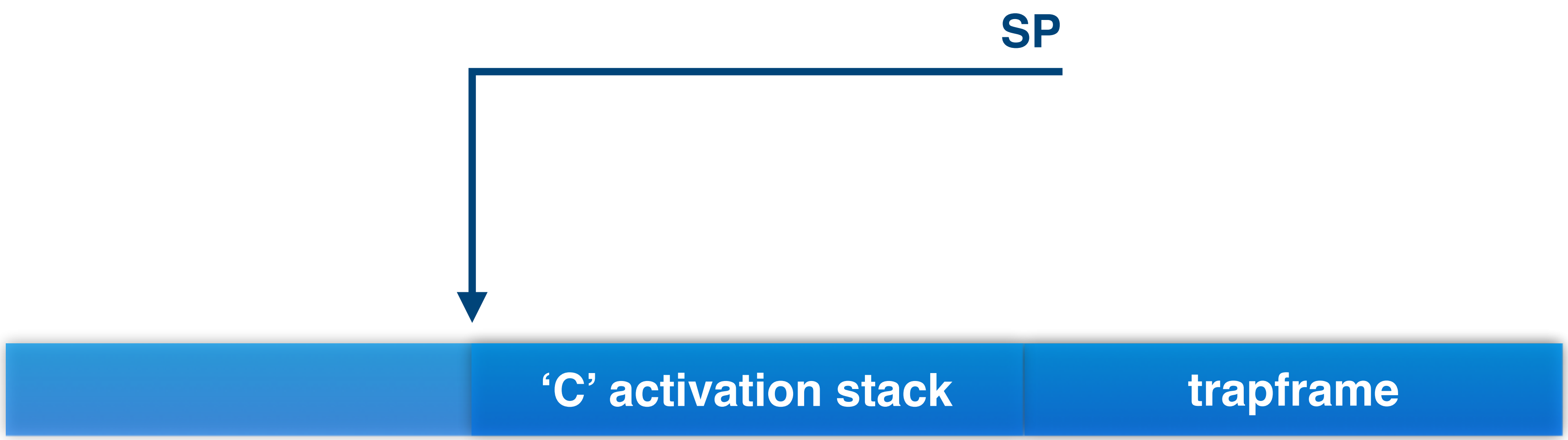
- Take an exception, syscall, or interrupt, and we switch to the kernel stack

Example Context Switch

- We push a trapframe on the stack
  - Also called exception frame, user-level context….
  - Includes the user-level PC and SP

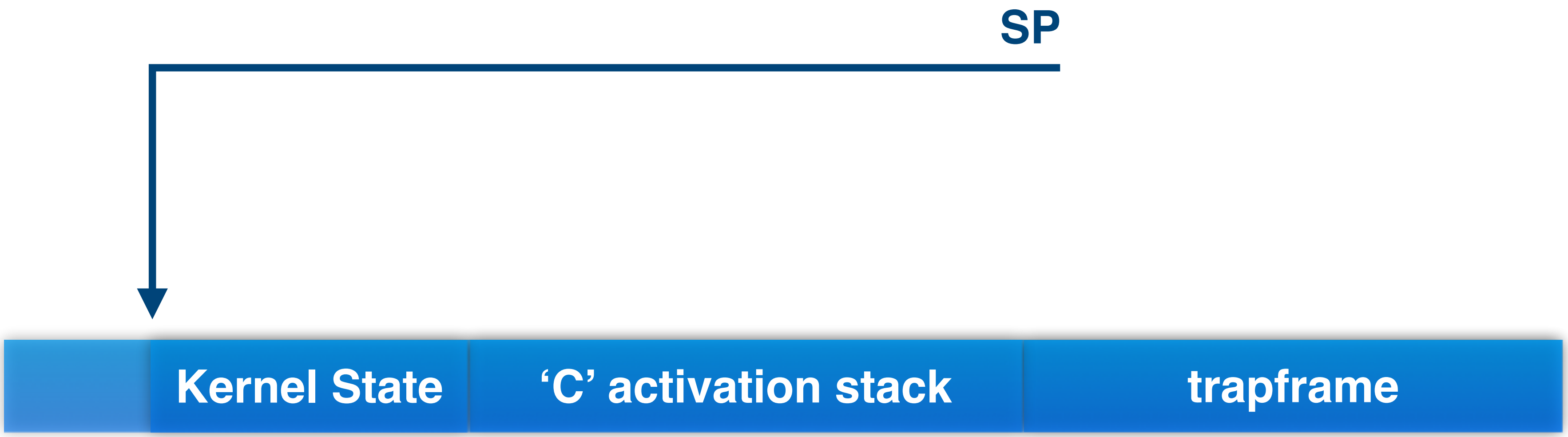
Example Context Switch

- Call ‘C’ code to process syscall, exception, or interrupt
- Results in a ‘C’ activation stack building up, since code of the OS is executed



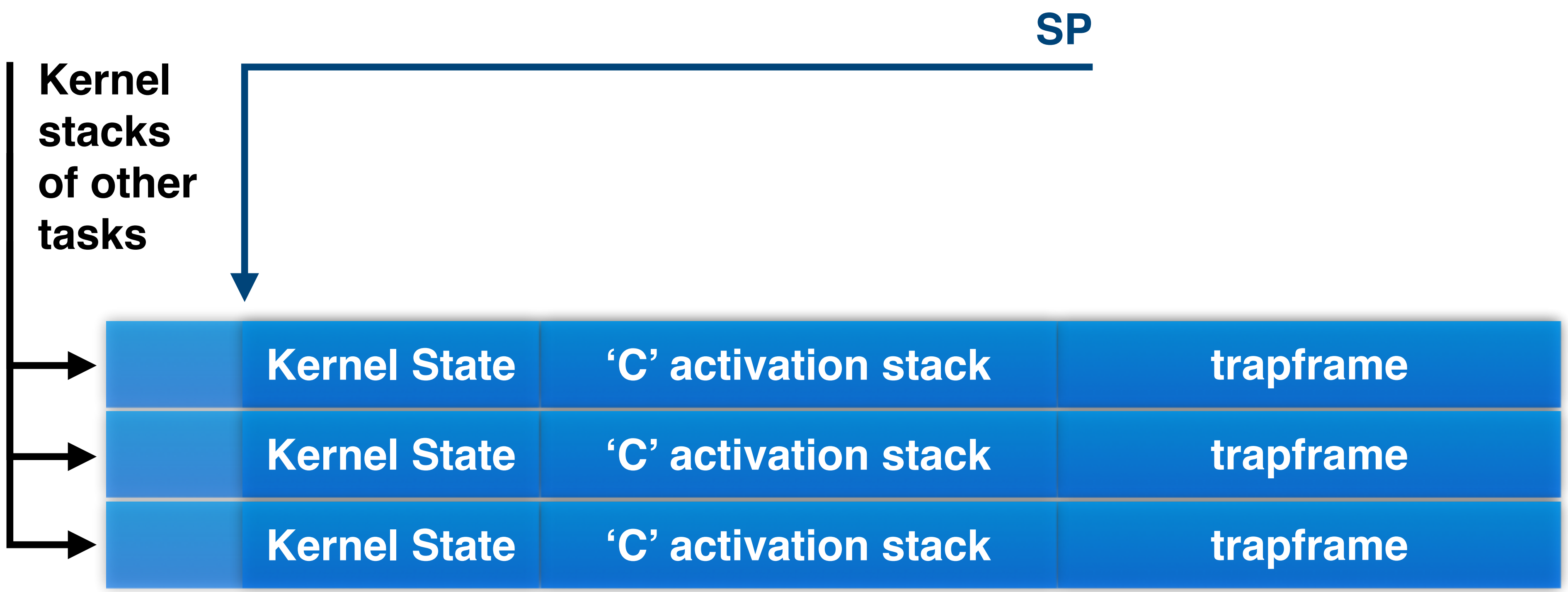
Example Context Switch

- Not the kernel decides to perform a context switch
- It chooses a target task (e.g. process)
- It pushes remaining kernel context onto the stack



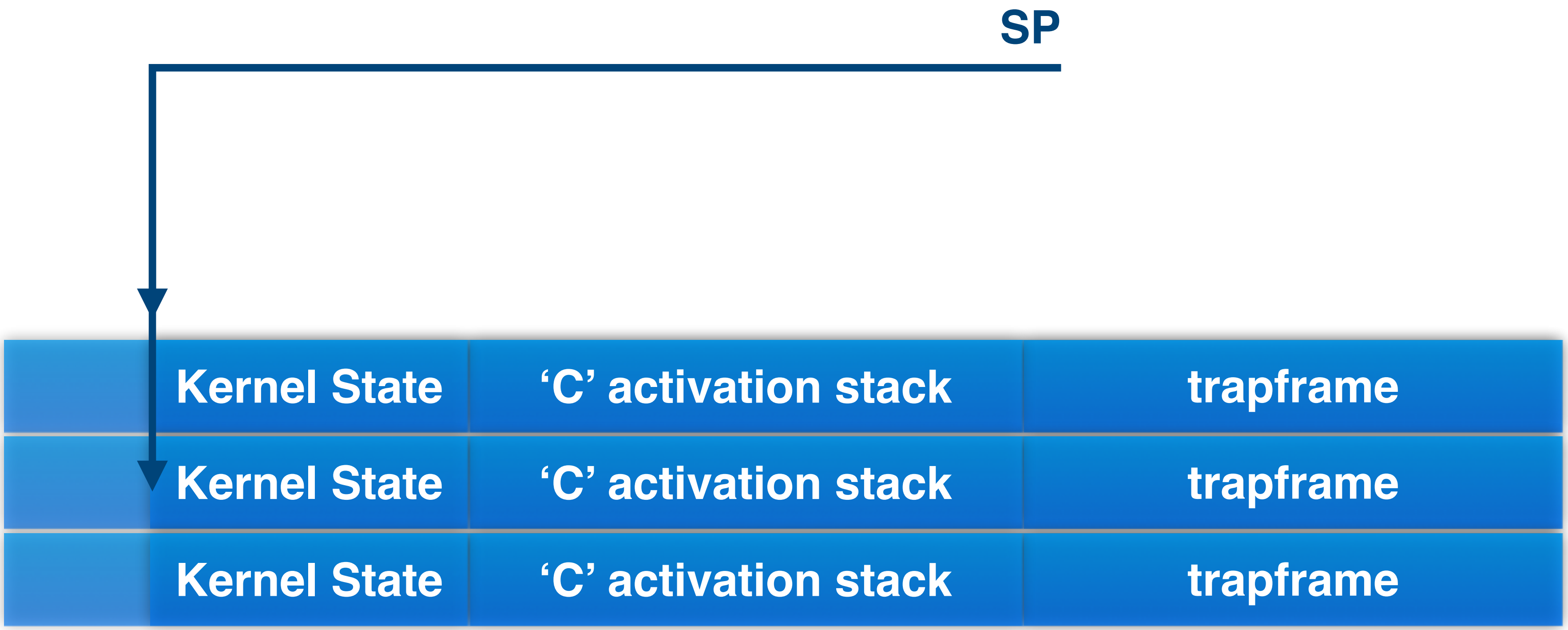
Example Context Switch

- Any other existing task must
  - be in kernel mode (on a uni processor) and
  - have a similar stack layout to the stack we are currently using



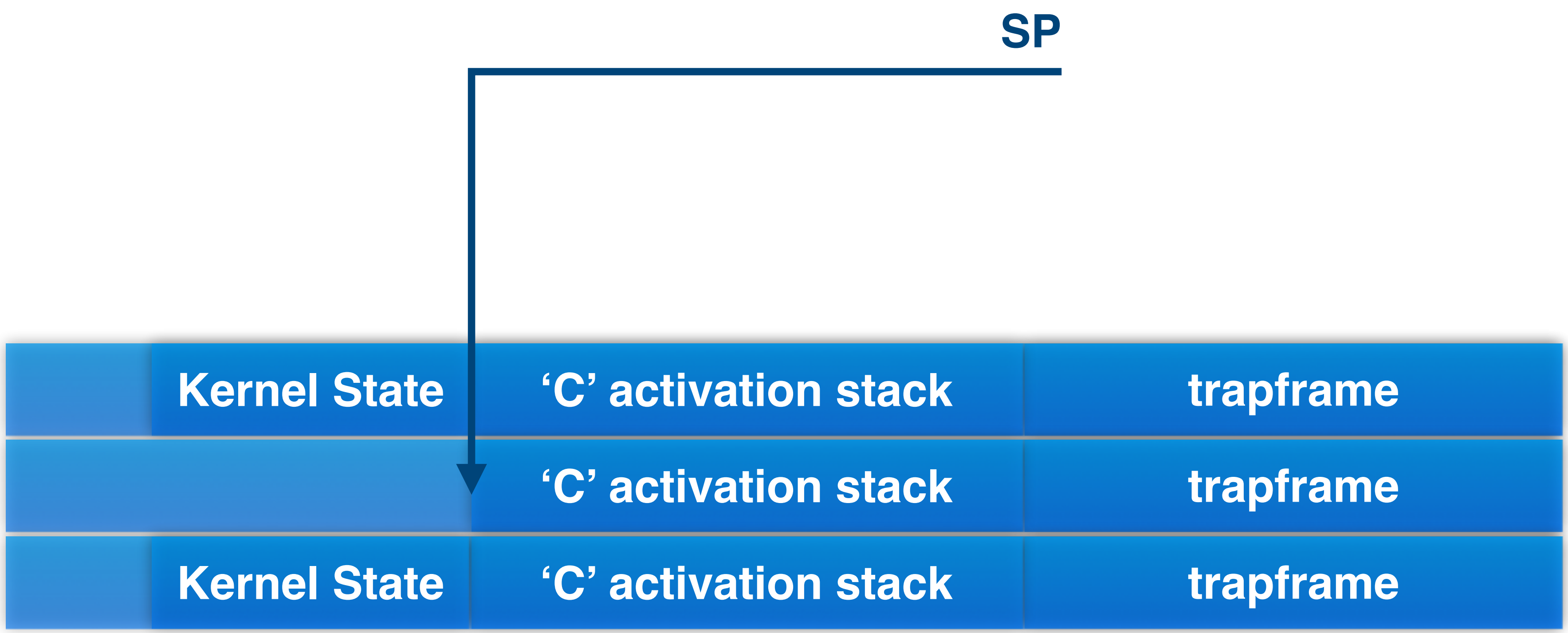
Example Context Switch

- We save the current SP in the PCB (or TCB) and
- load the SP of the target thread.
- Thus we have switched contexts



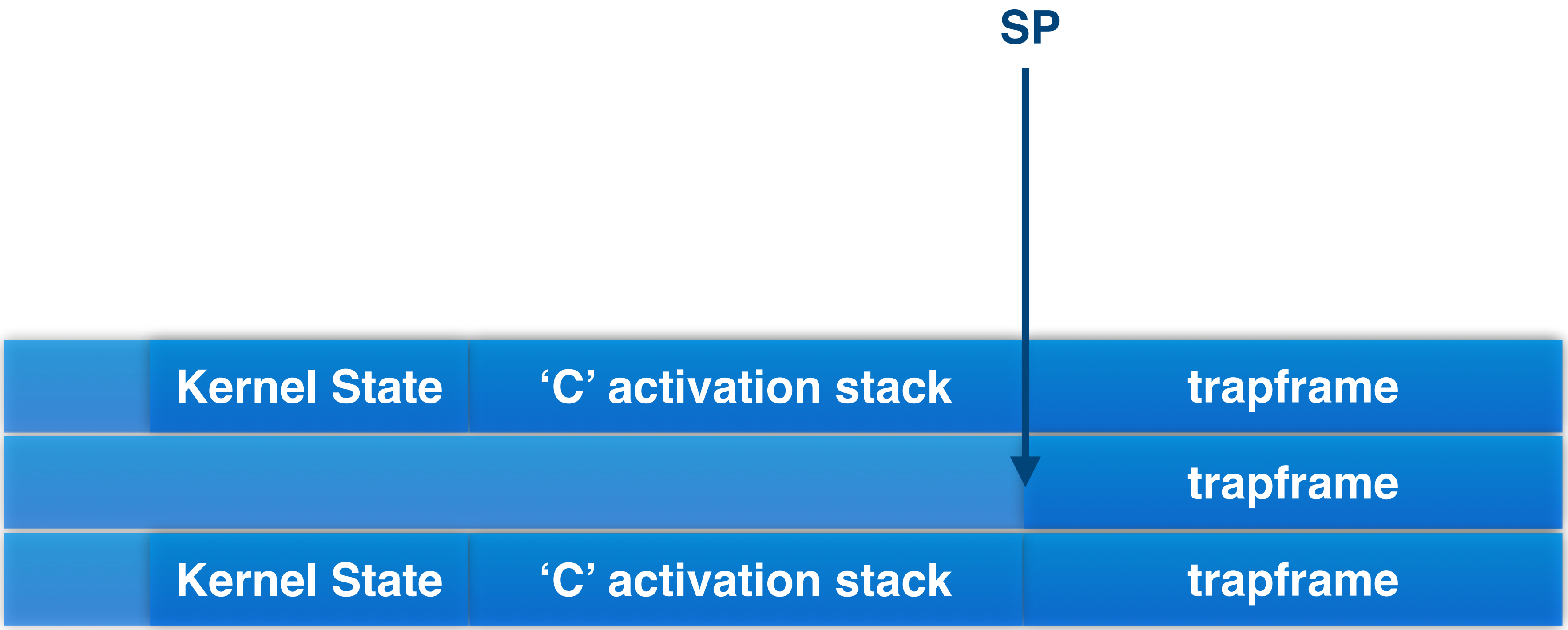
Example Context Switch

- Load the target thread's previous context, and return to C



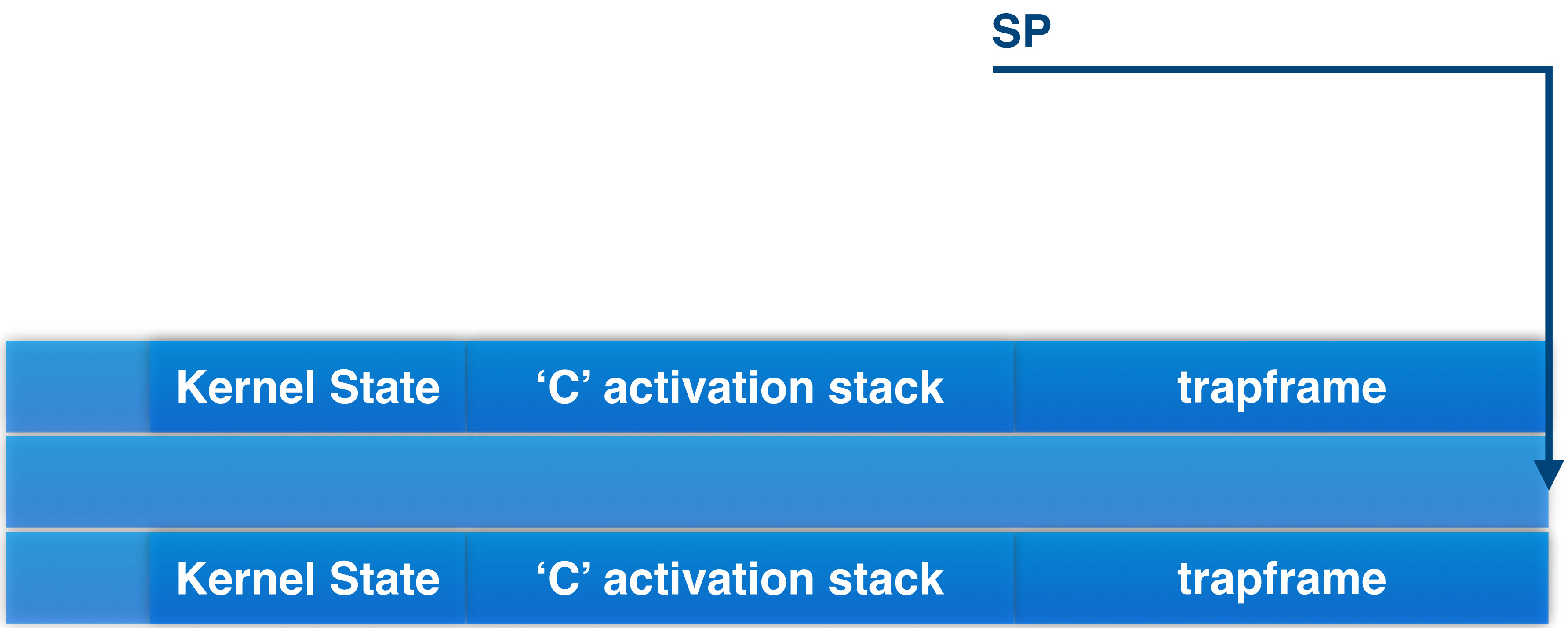
Example Context Switch

- Load the target thread's previous context, and return to C
  - C-Code of the operating system will be finished, where we left it because of an different context switch (long time before …)



Example Context Switch

- The user-level context is restored
  - Trapframe is saved back, preparing to run the user task



Example Context Switch

- SP points to user stack
  - The process continues as if nothing had happend

SP -> UserStack

Worried About Concurrency?

- What happens if, during interrupt or trap handling, another interrupt occurs?
- OS handles these situations:
  - **Disable interrupts** during interrupt processing
  - Use a number of sophisticate **locking** schemes to protect concurrent access to internal data structures.

Thanks

Questions