Karlsruhe Institute of Technology

Microkernel Construction
I.8 – Interrupts, Exceptions, CPU virtualization

Lecture Summer Term 2017
Wednesday 15:45-17:15 R 131, 50.34 (INFO)

Jens Kehne | Marius Hillenbrand
System Architecture Group, Department of Computer Science

L4 Kernel Paradigm

Everything the kernel needs to handle in a secure manner will either become invisible or be hidden behind an abstraction.

- Events that are not handled by the kernel itself will be posted to user land
  - Page faults
  - Hardware interrupts
  - Exceptions

Event Sources

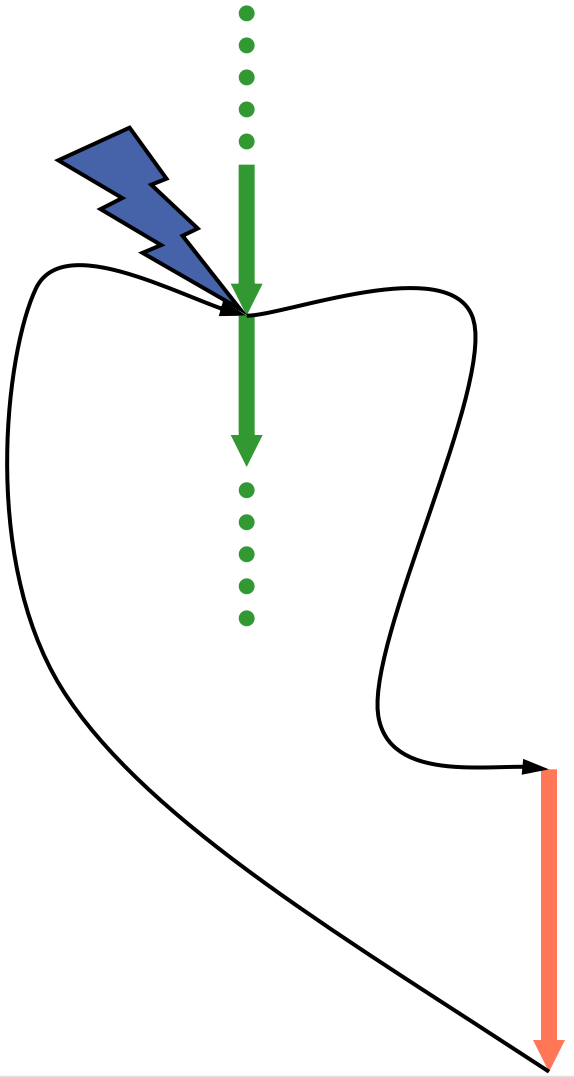
- From current instruction stream (“exceptions”)
  - Page fault
  - Numeric
  - Unaligned data access
  - Debug
  - Speculation
- External (“interrupts”)
  - Device interrupts
  - Timer interrupt
  - Inter-processor interrupt

Event Classes

- Traps / interrupts
  - Sensed after an instruction
  - Deal with the cause, then continue
- Faults
  - Signaled during the execution of the current instruction
  - Fix the problem, then retry (or skip)

Event Handling

1. Program executes happily
2. Event occurs
3. Activate event handler
   - Save current state
   - Switch to privileged mode
   - Execute event handler
4. Fix the problem / handle event
5. End of event handling
   - Restore state
   - Switch to previous mode
   - Continue interrupted program
6. Program executes happily again

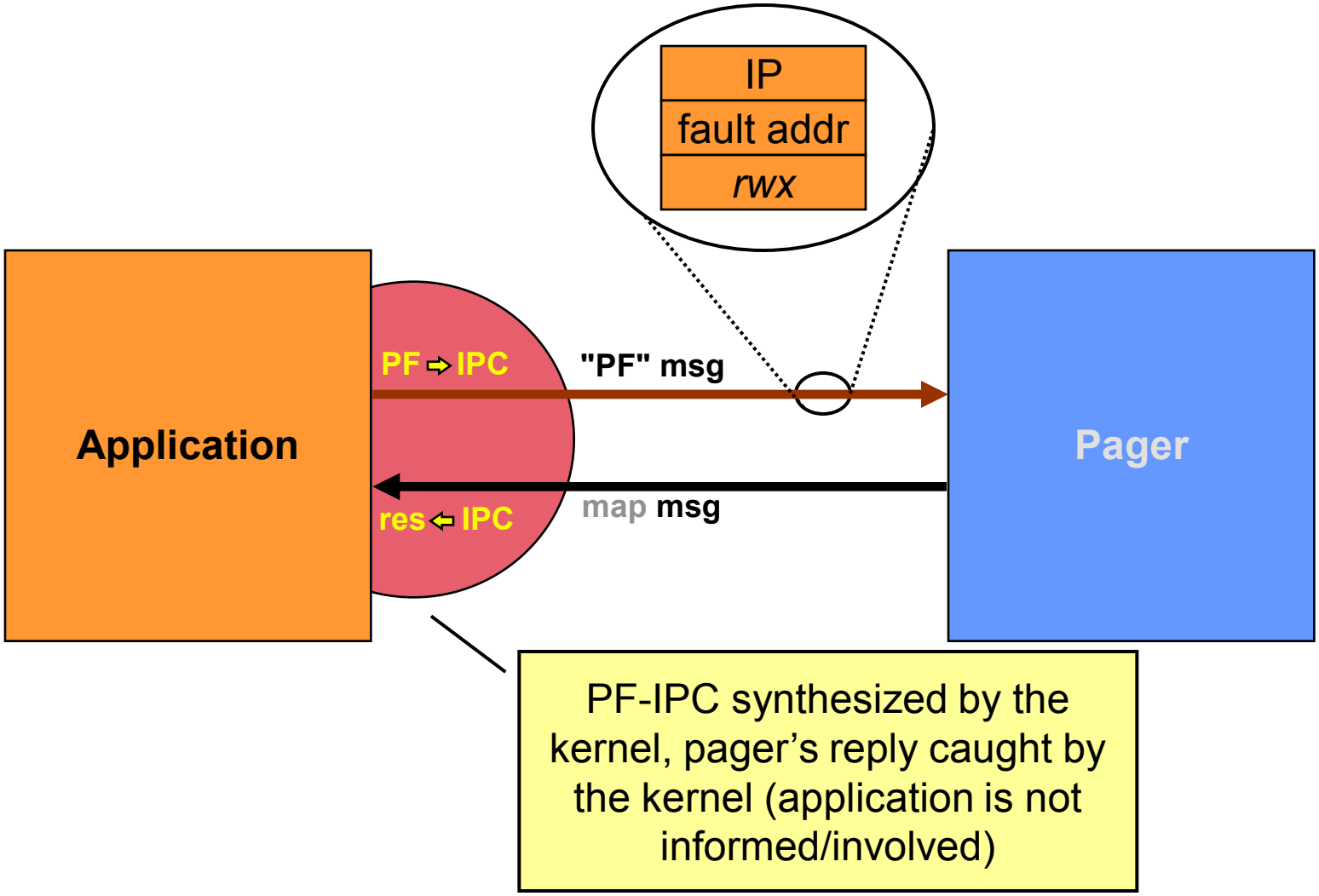


L4 Kernel Paradigm

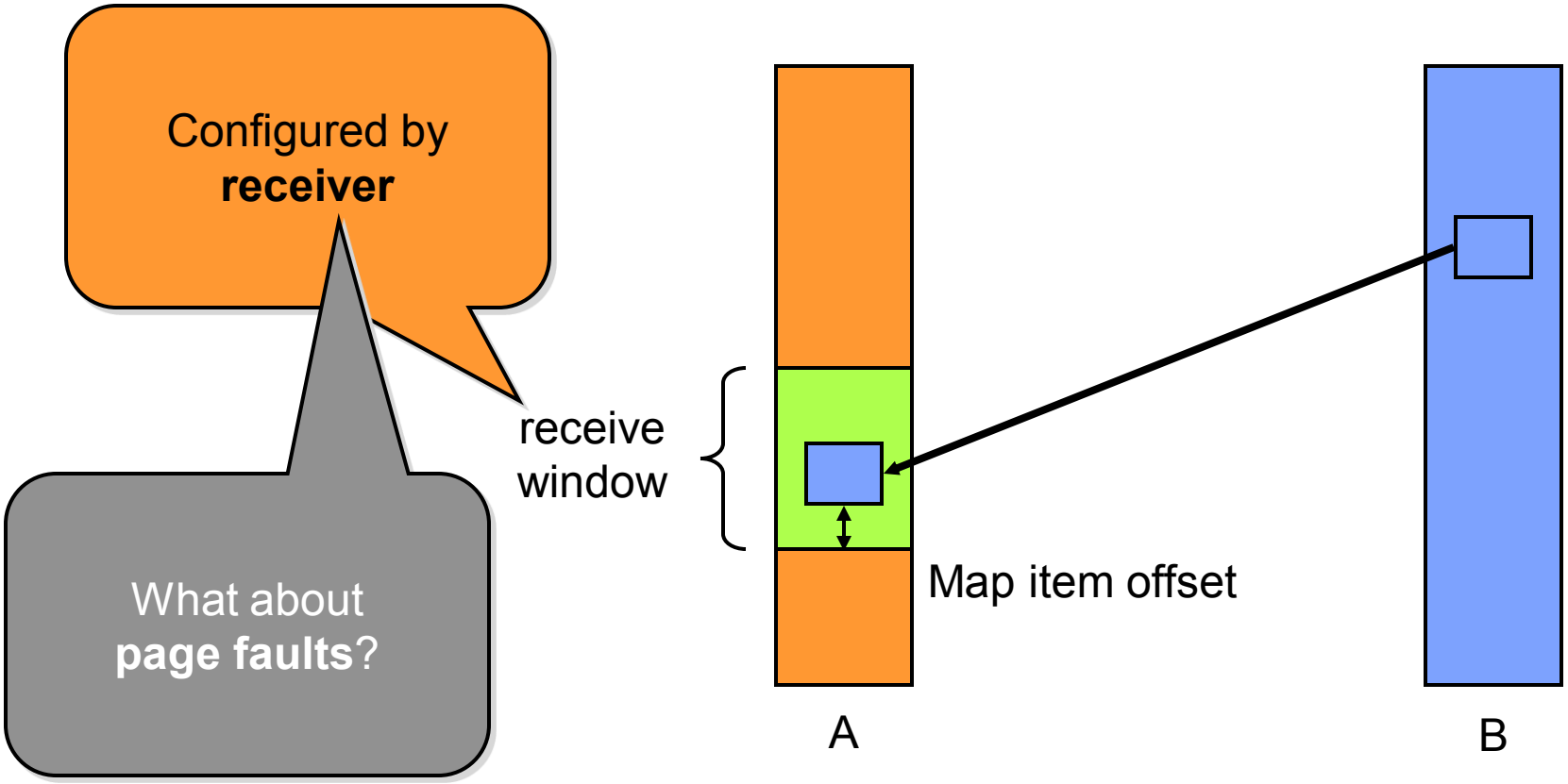
Everything the kernel needs to handle in a secure manner will either become invisible or be hidden behind an abstraction.

- Events that are not handled by the kernel itself will be posted to user land
  - Page faults
  - Hardware interrupts
  - Exceptions

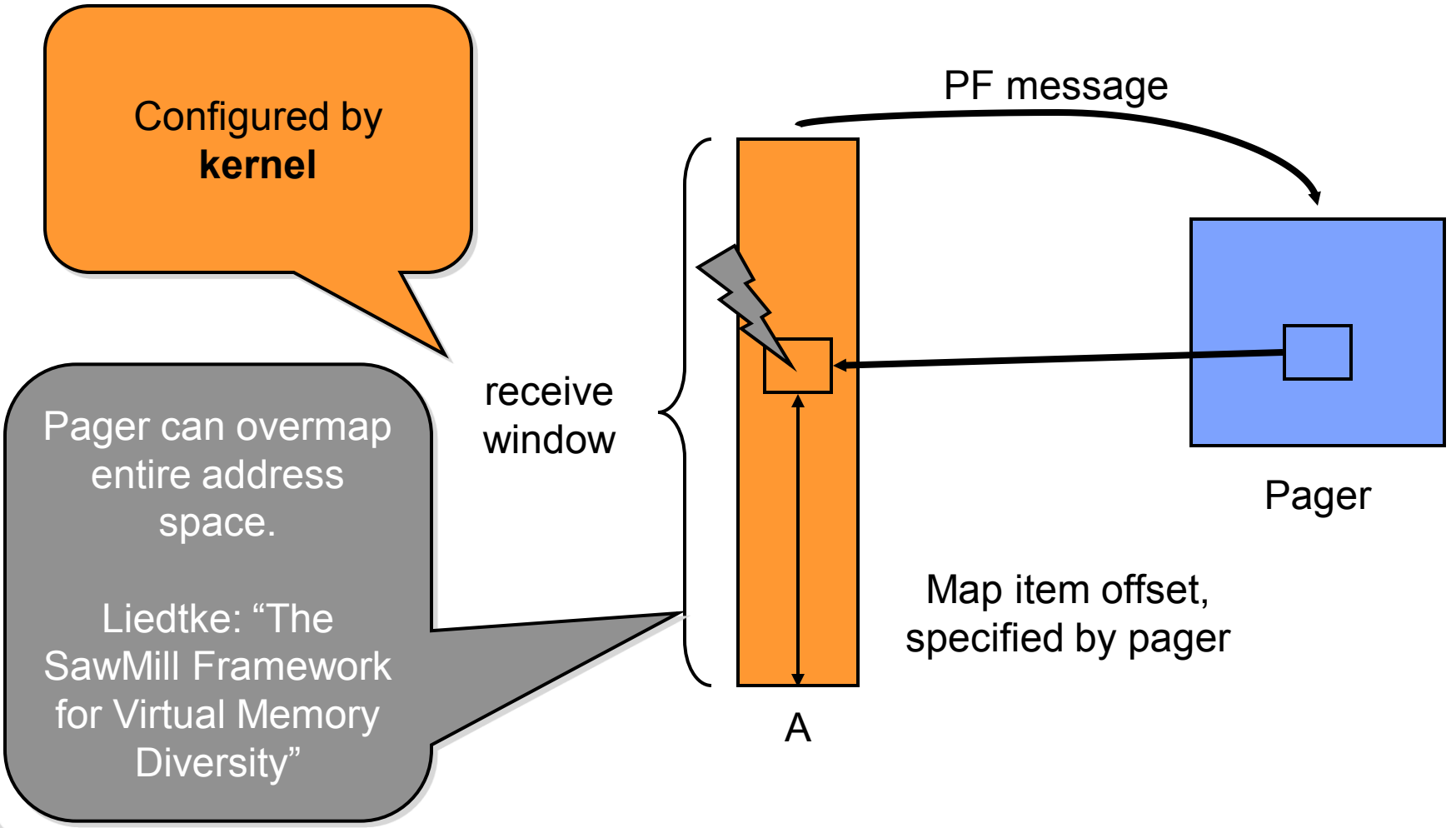
Page Fault IPC



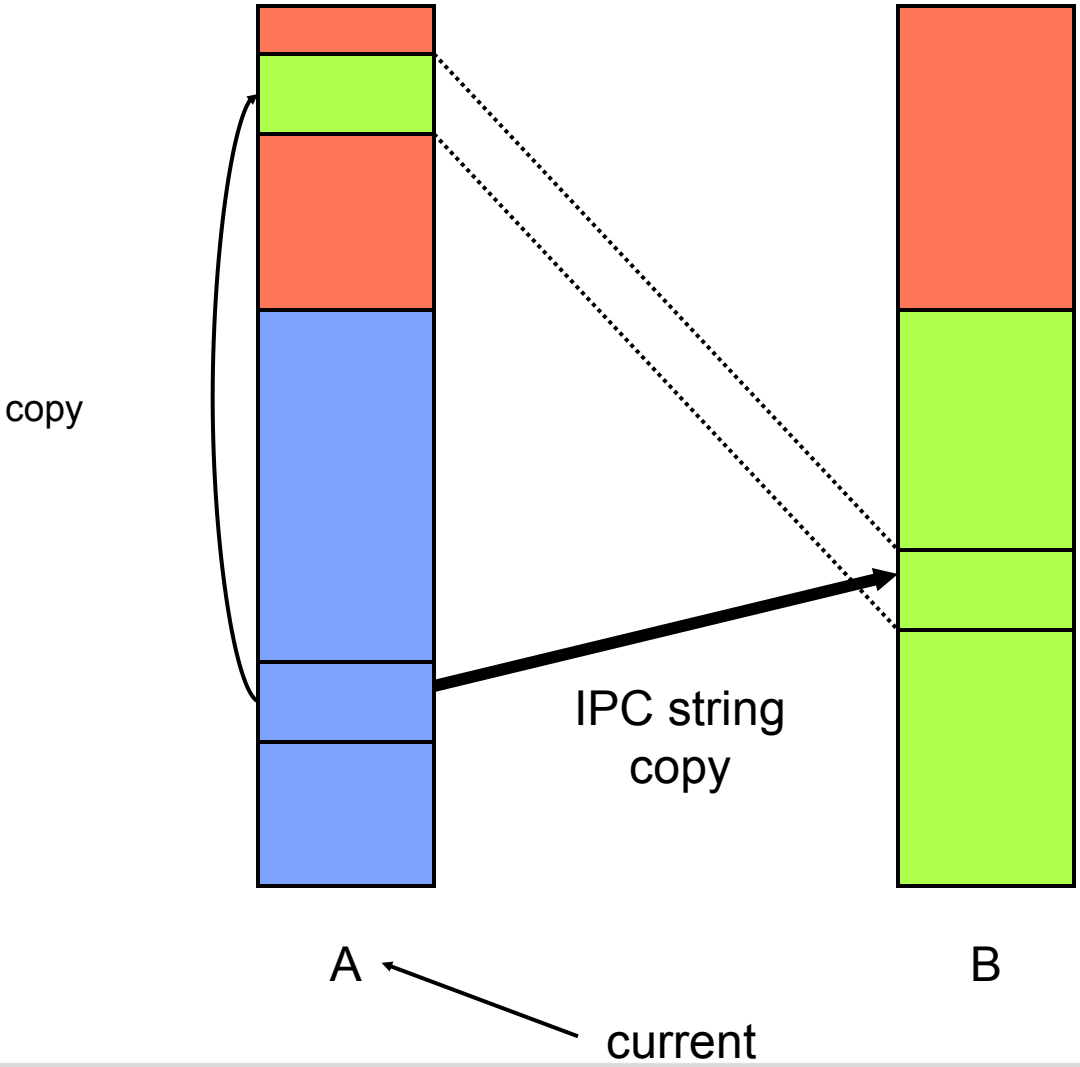
IPC Map


Configured by
receiver
What about
page faults?
receive
window
Map item offset
A
B

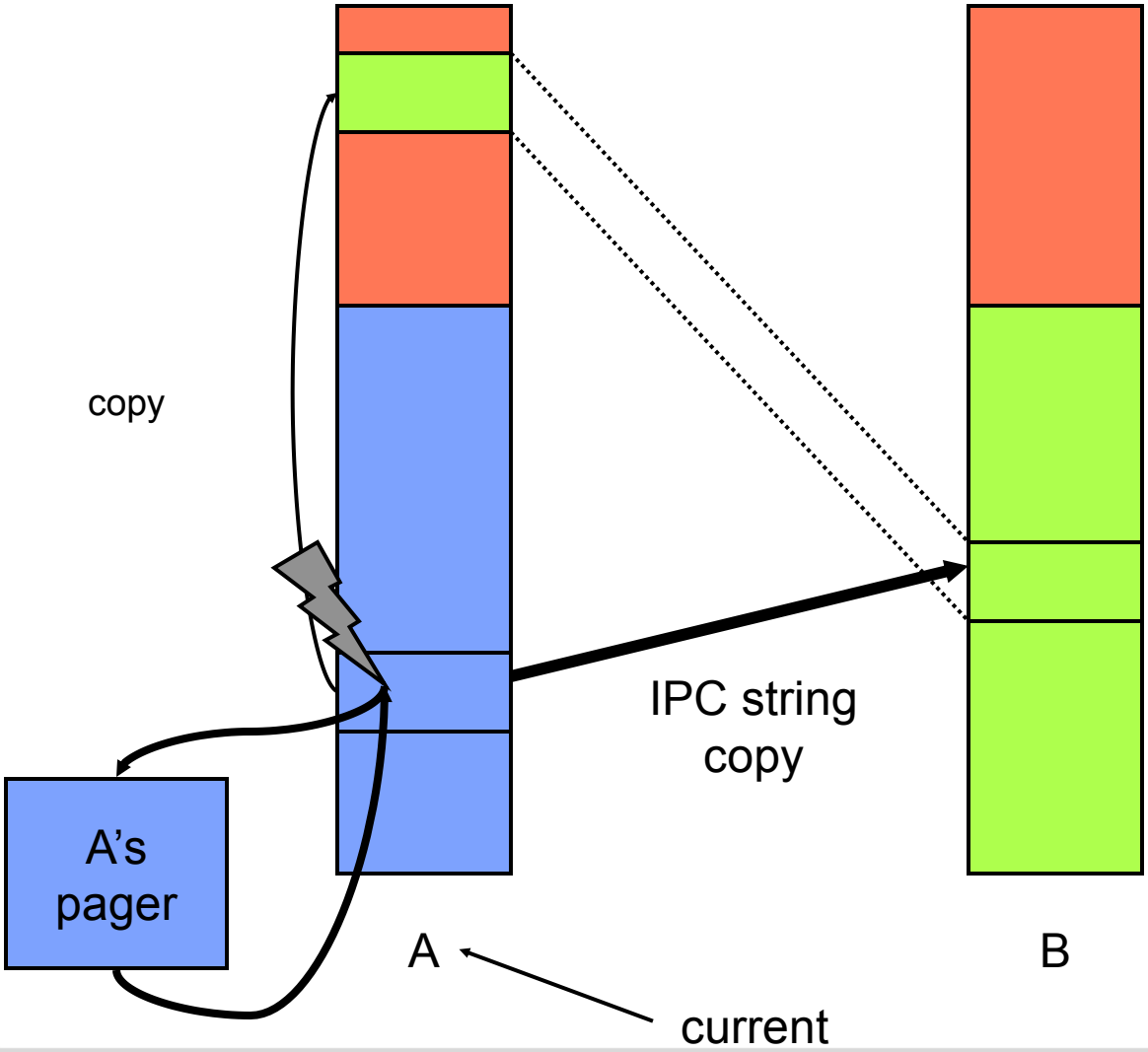
Page Fault Receive Window



String Copy

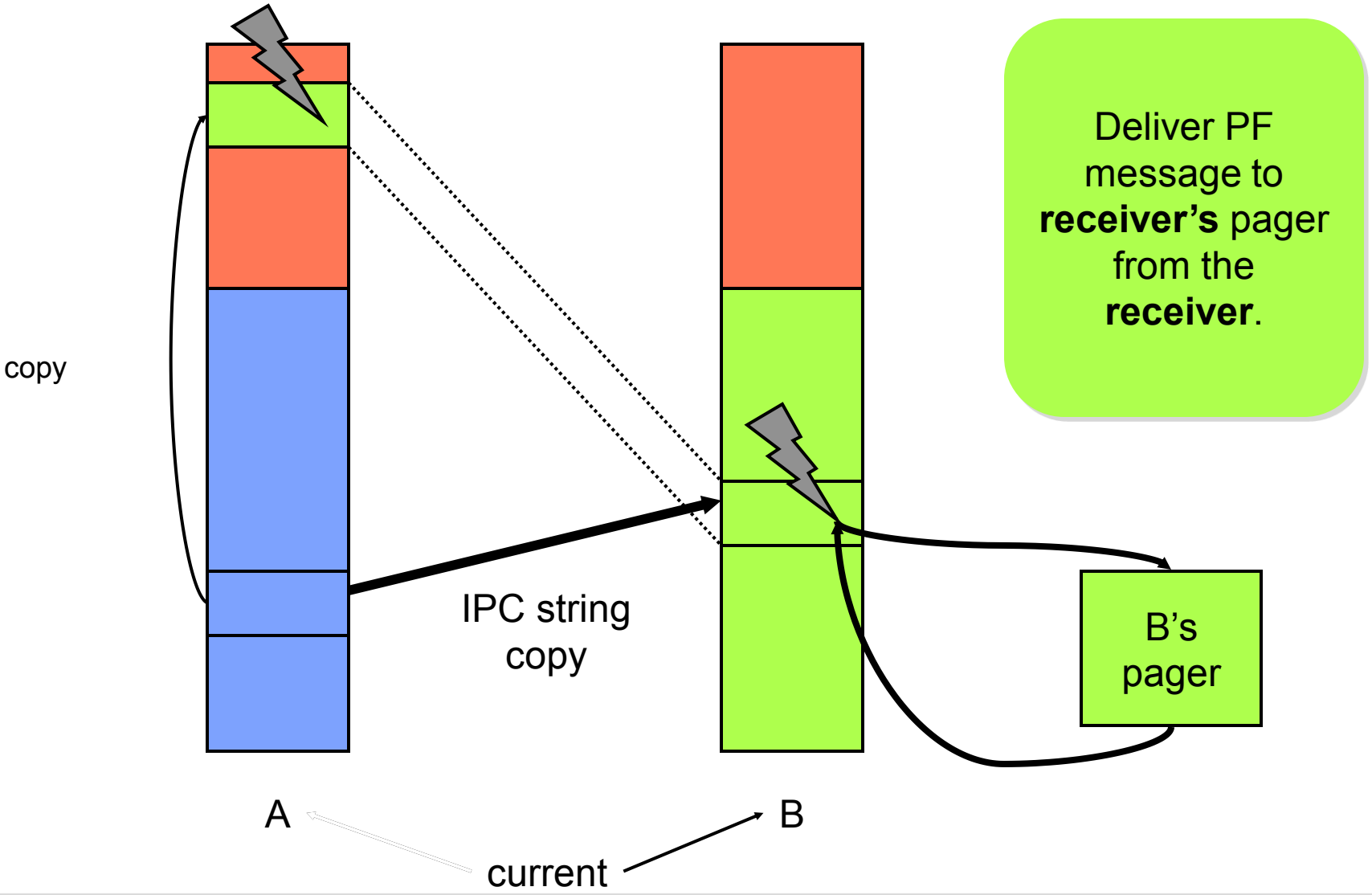


String Copy: Sender Page Fault



Deliver PF message to **sender's** pager from the **sender**.

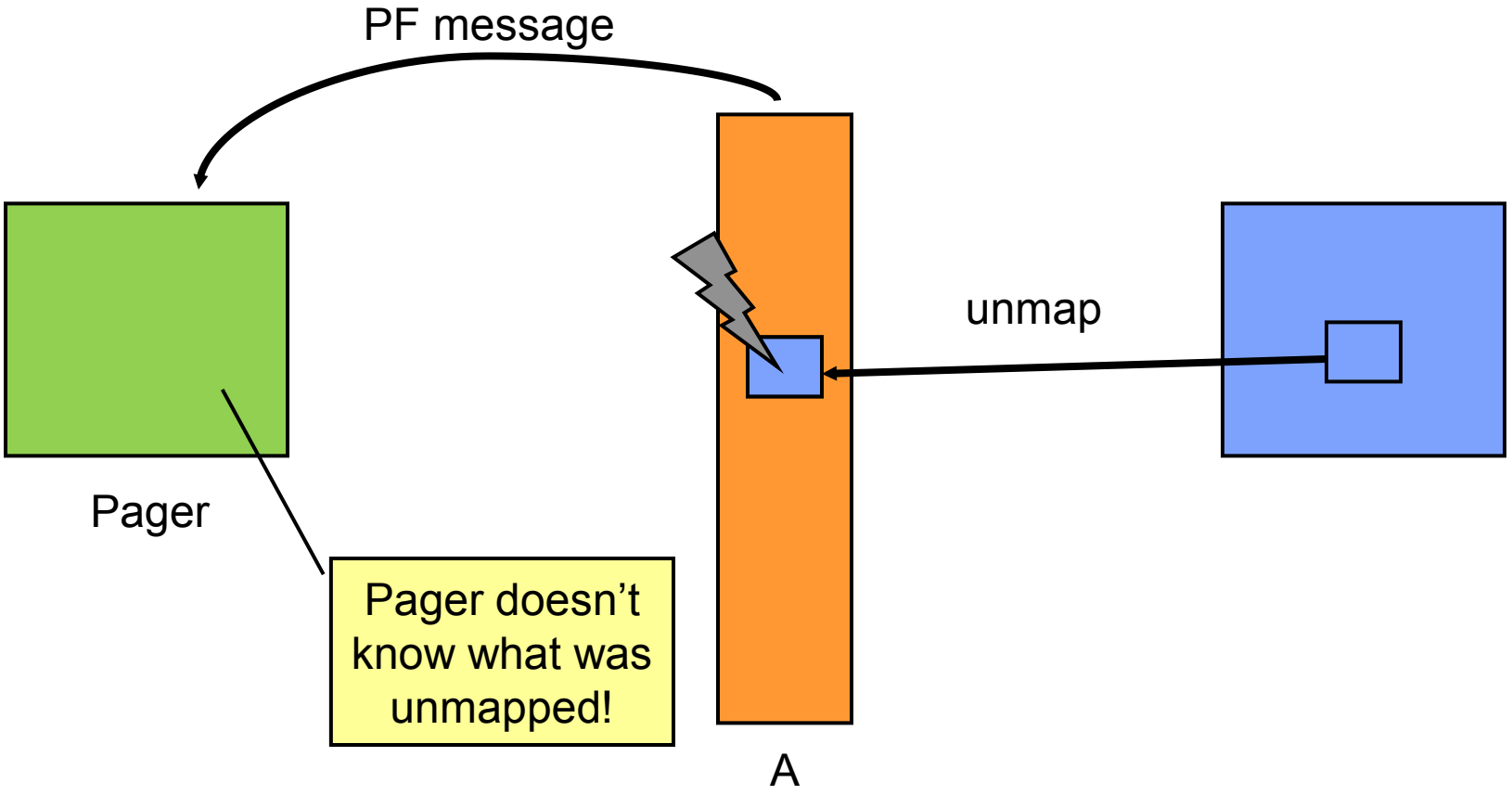
String Copy: Receiver Page Fault



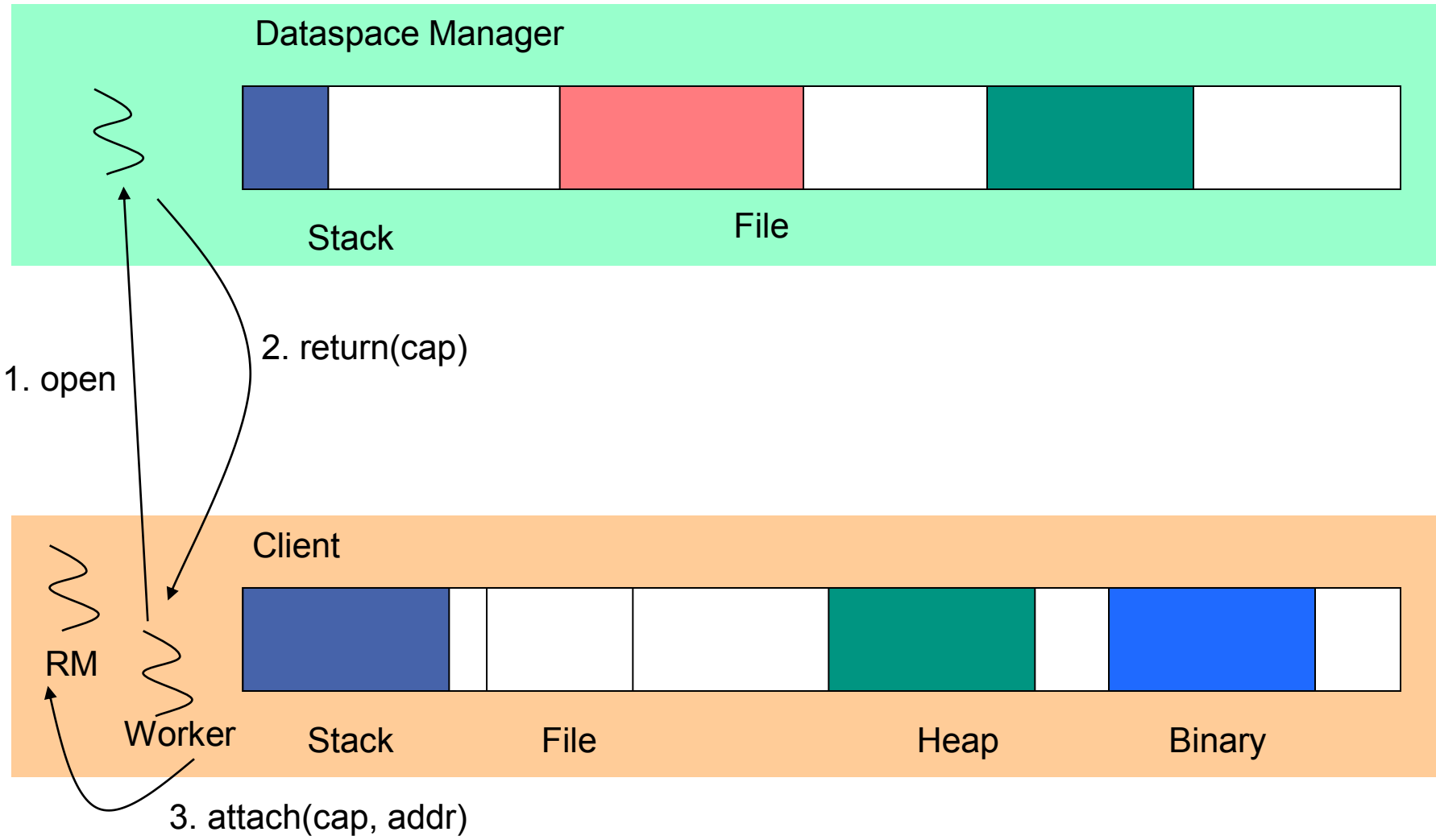
Deliver PF message to **receiver's** pager from the **receiver**.

Dataspaces

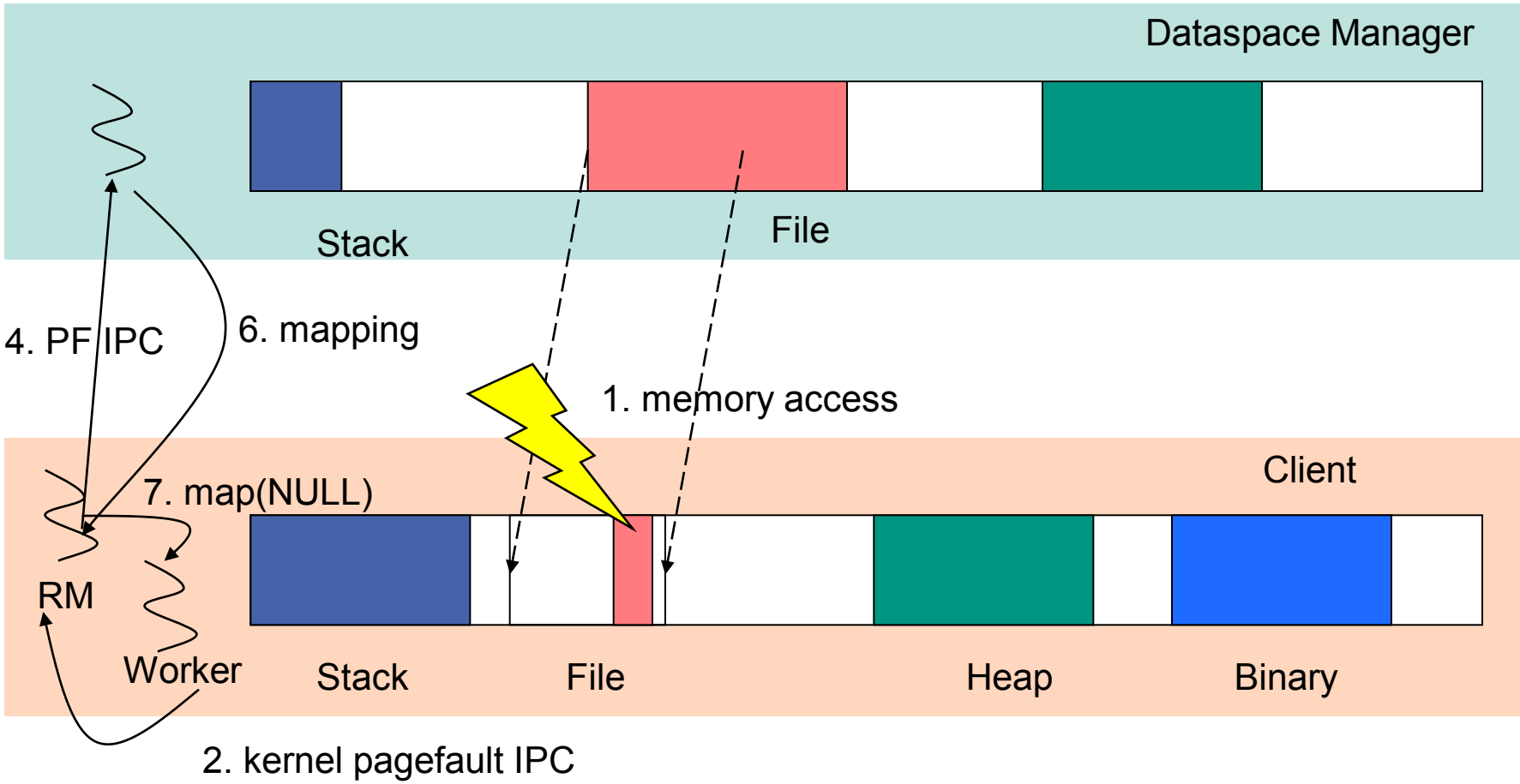
- What if we want to receive mappings from more than one task?



Dataspaces

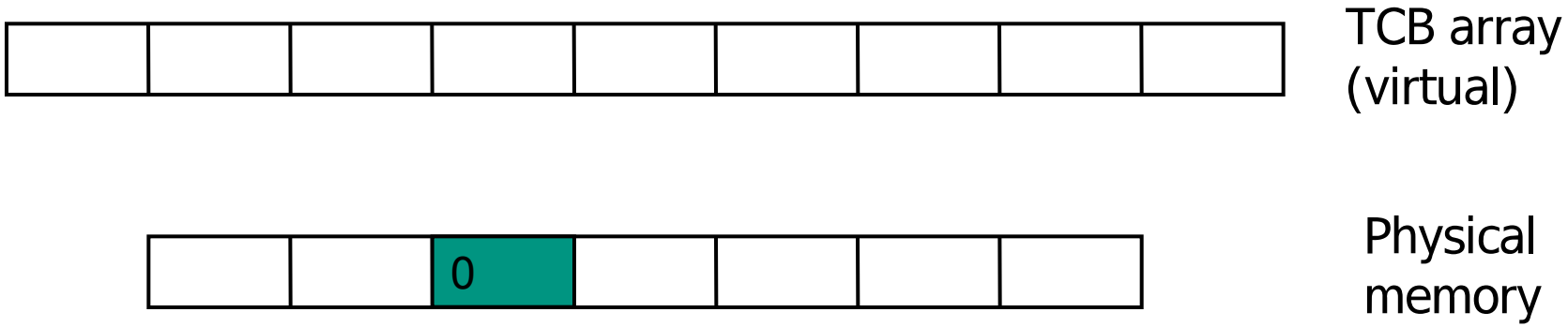

Dataspace Manager
Stack
File
1. open
2. return(cap)
Client
RM
Worker
Stack
File
Heap
Binary
3. attach(cap, addr)

Dataspaces and Pagefaults

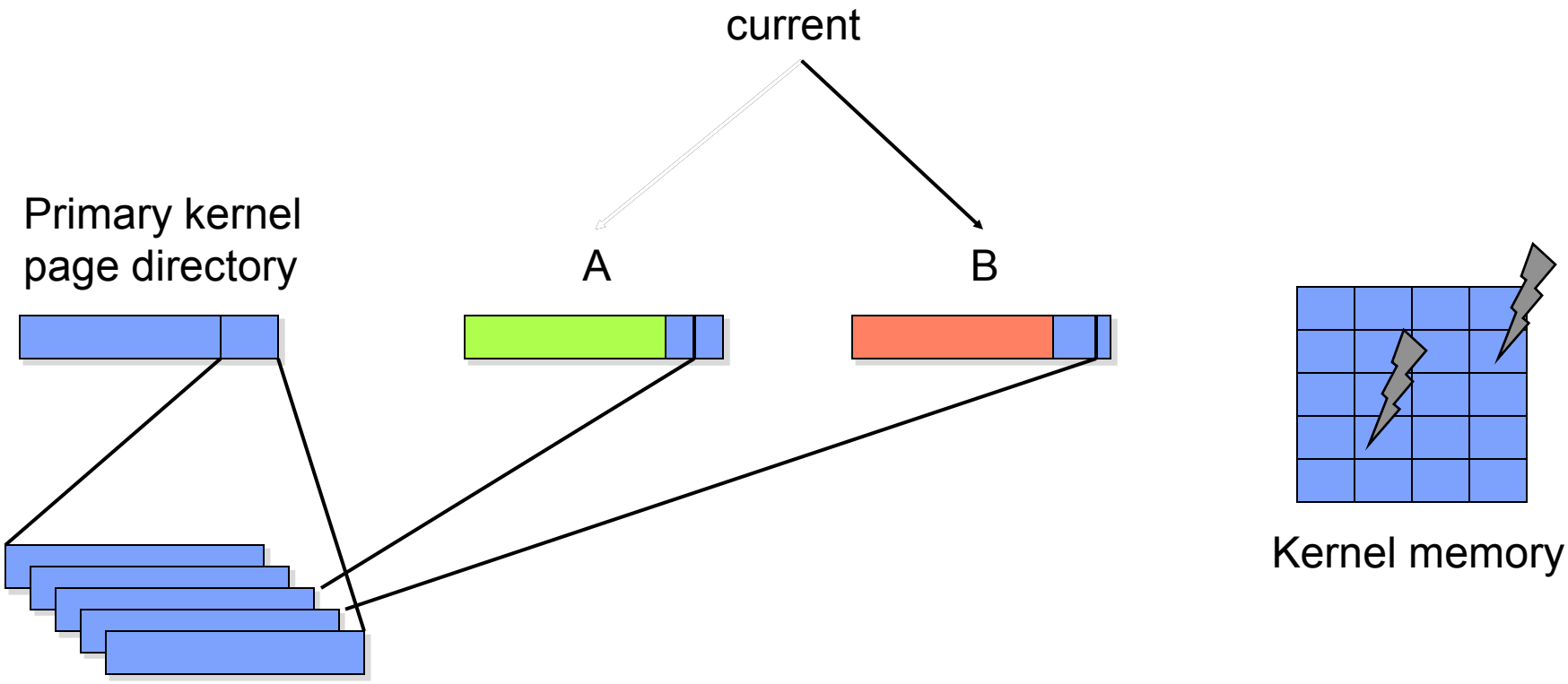


Invisible Page Faults

- Kernel handles some page faults internally
  - Virtual TCB array – map on demand
    - Exclusive 0-filled page on write access
    - Shared 0-filled read-only page on read access
      - Avoid DoS attack on memory used for TCBs
      - Map exclusive 0-filled page on later write



Lazy Kernel Space Building



Note: The kernel shares page **tables**, not page **directories**;
implemented by copying page directory entries.

L4 Kernel Paradigm

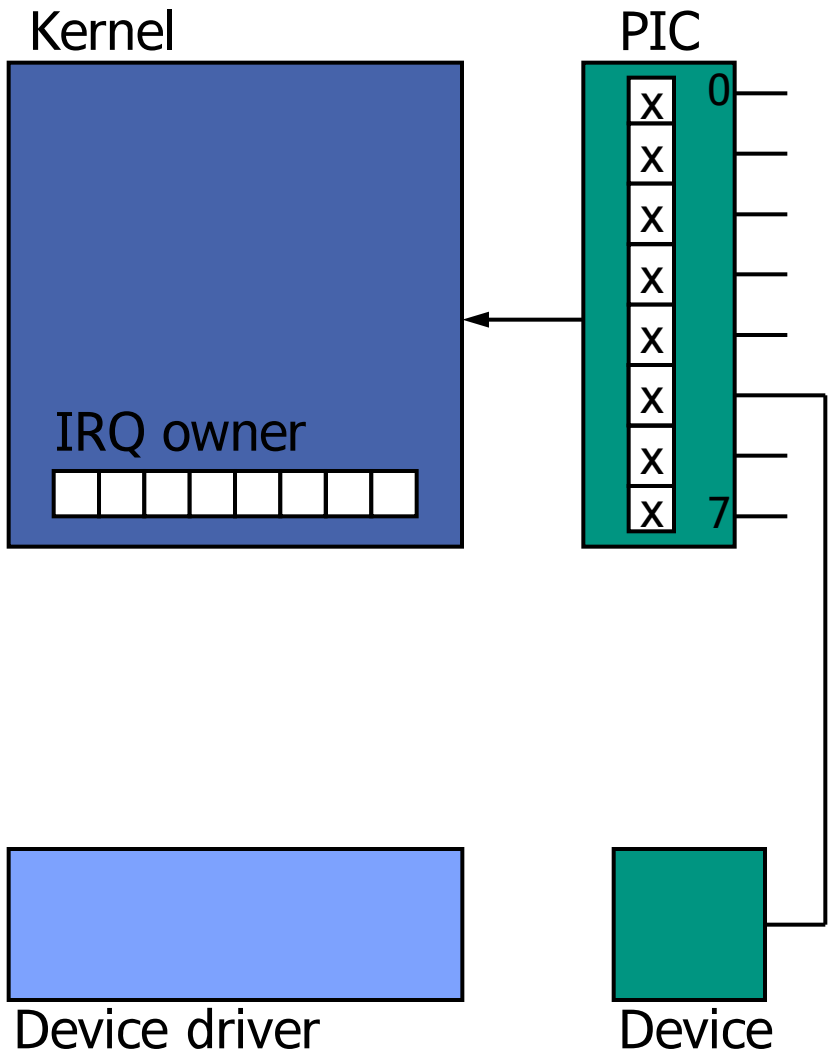
Everything the kernel needs to handle in a secure manner will either become invisible or be hidden behind an abstraction.

- Events that are not handled by the kernel itself will be posted to user land
  - Page faults
  - Hardware interrupts
  - Exceptions

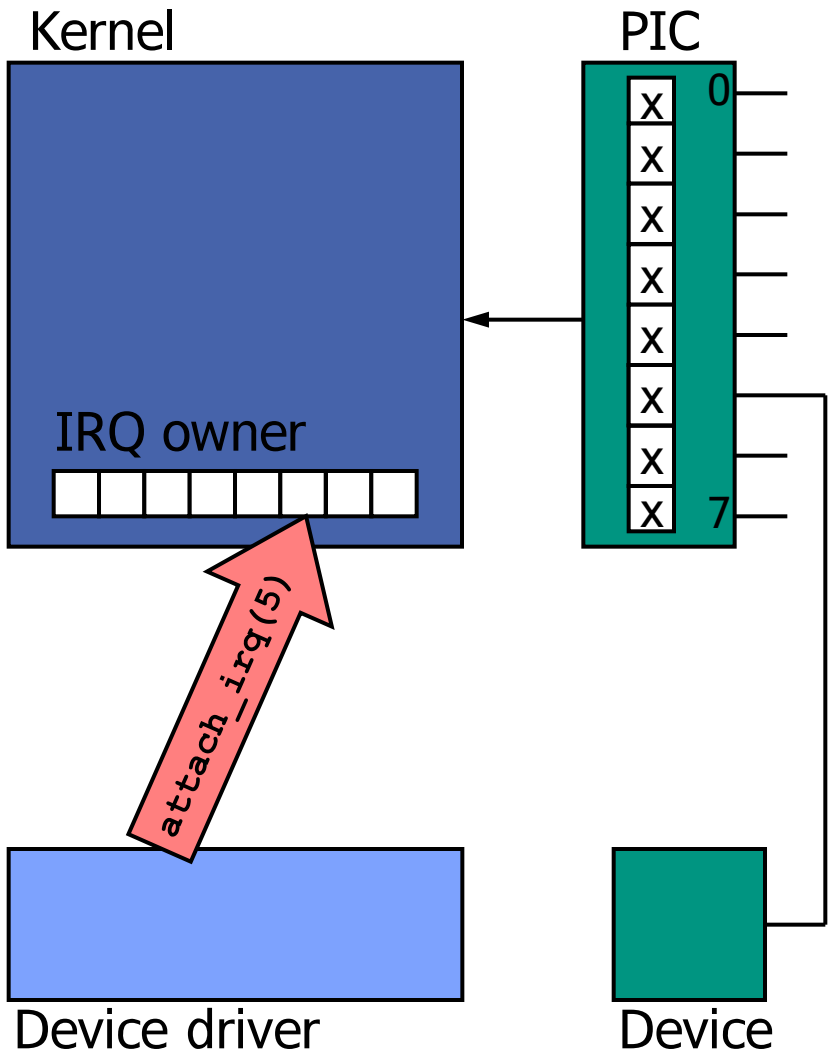
Hardware Interrupts

- Kernel hides first-level interrupt logic
  - No user messing with interrupt hardware
  - Deliver interrupts via IPC
  - More portable software
- Kernel interrupt handler
  - Translates interrupt into IPC
  - Sender: interrupt thread ID
    - Represents interrupt request line
  - Receiver: attached thread (user interrupt handler)
- Message destination
  - A thread needs to “attach to an interrupt”

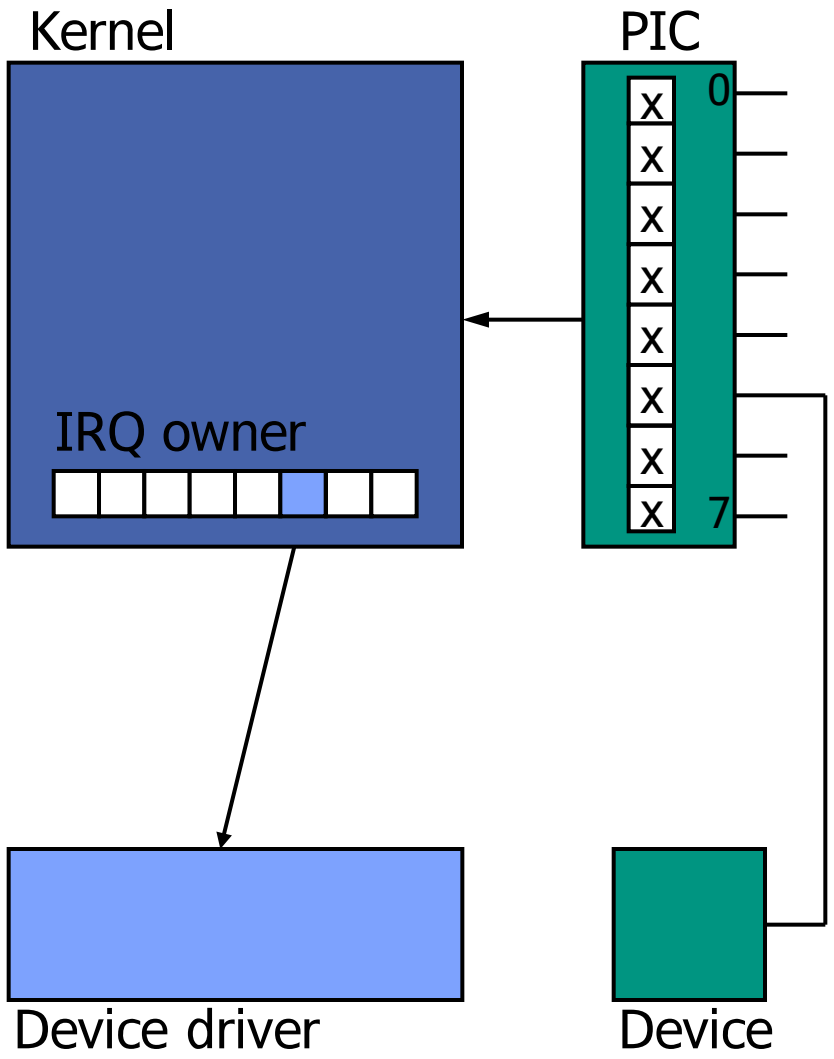
Interrupt Handling



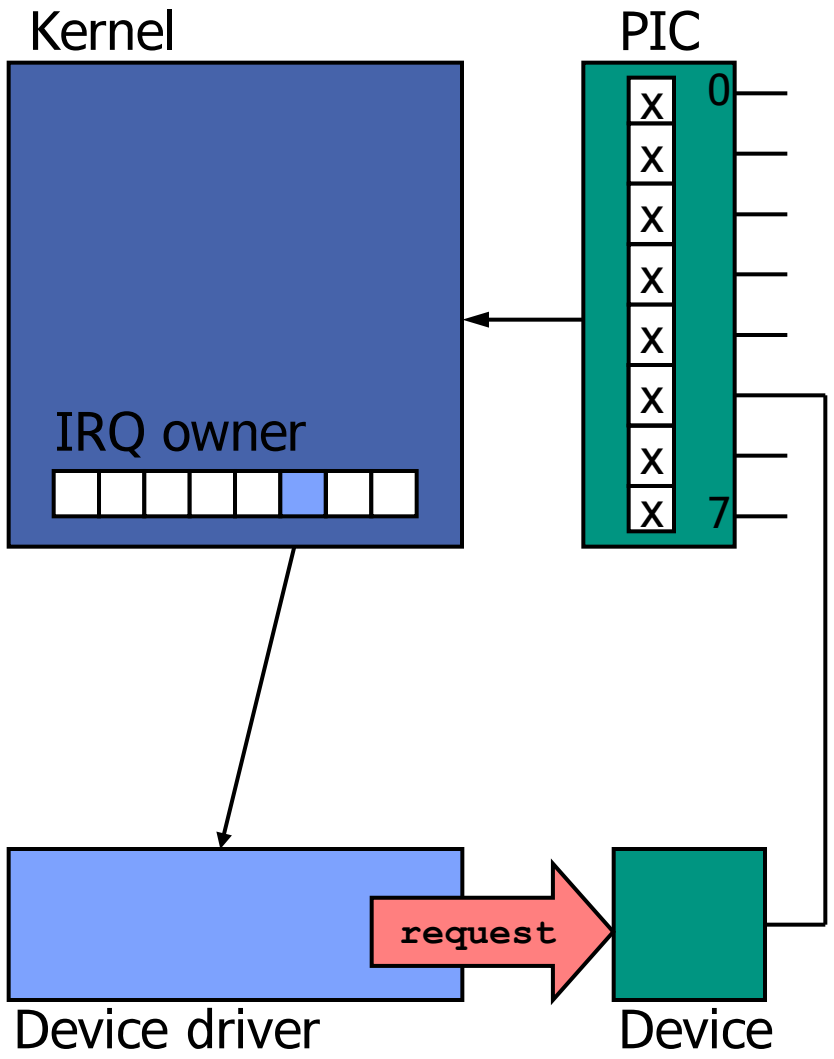
Interrupt Handling



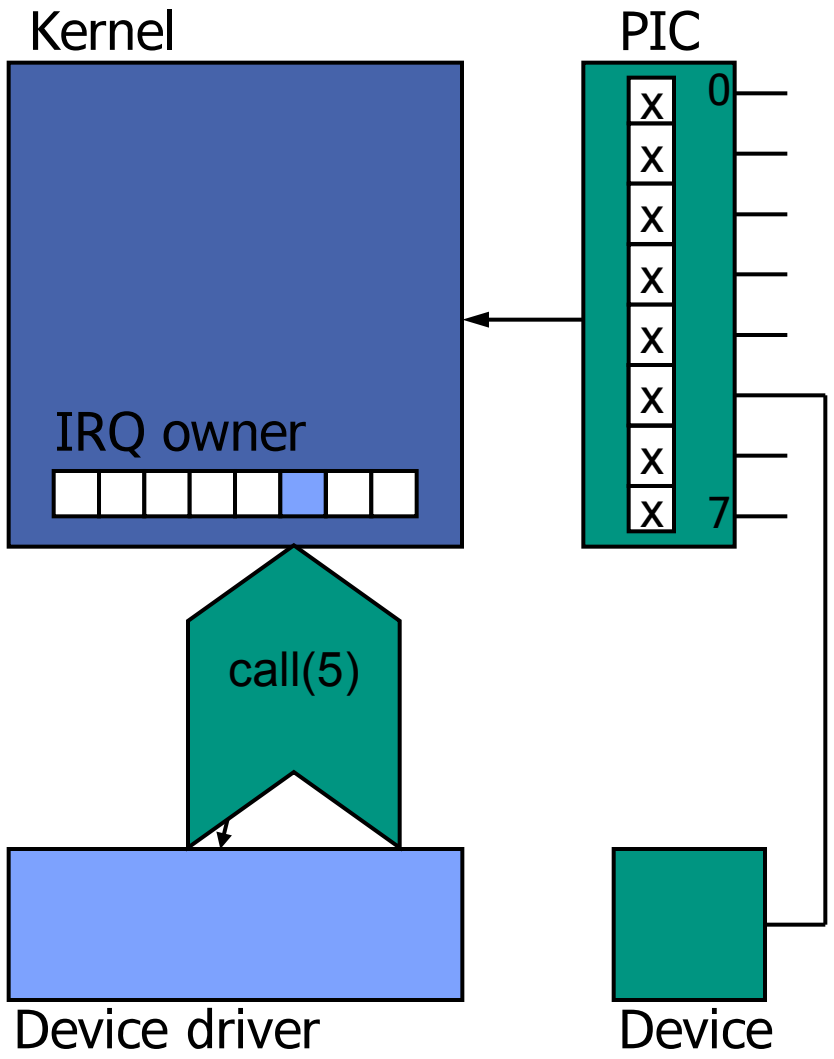
Interrupt Handling



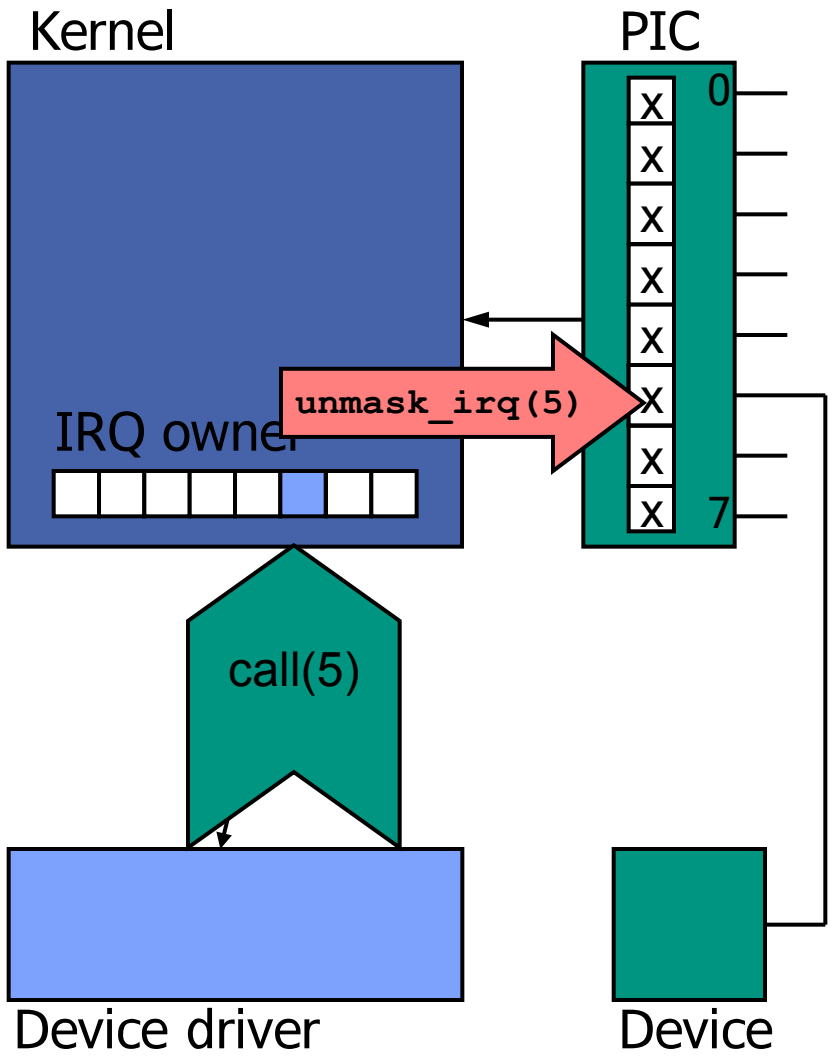
Interrupt Handling



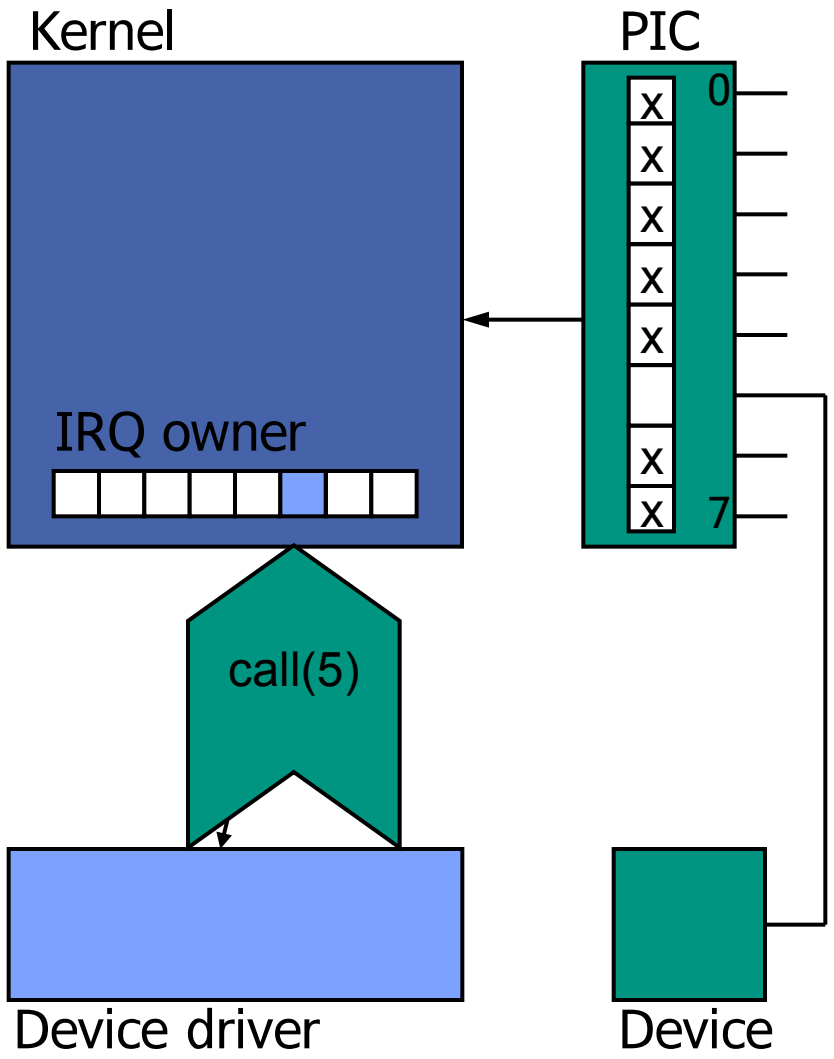
Interrupt Handling



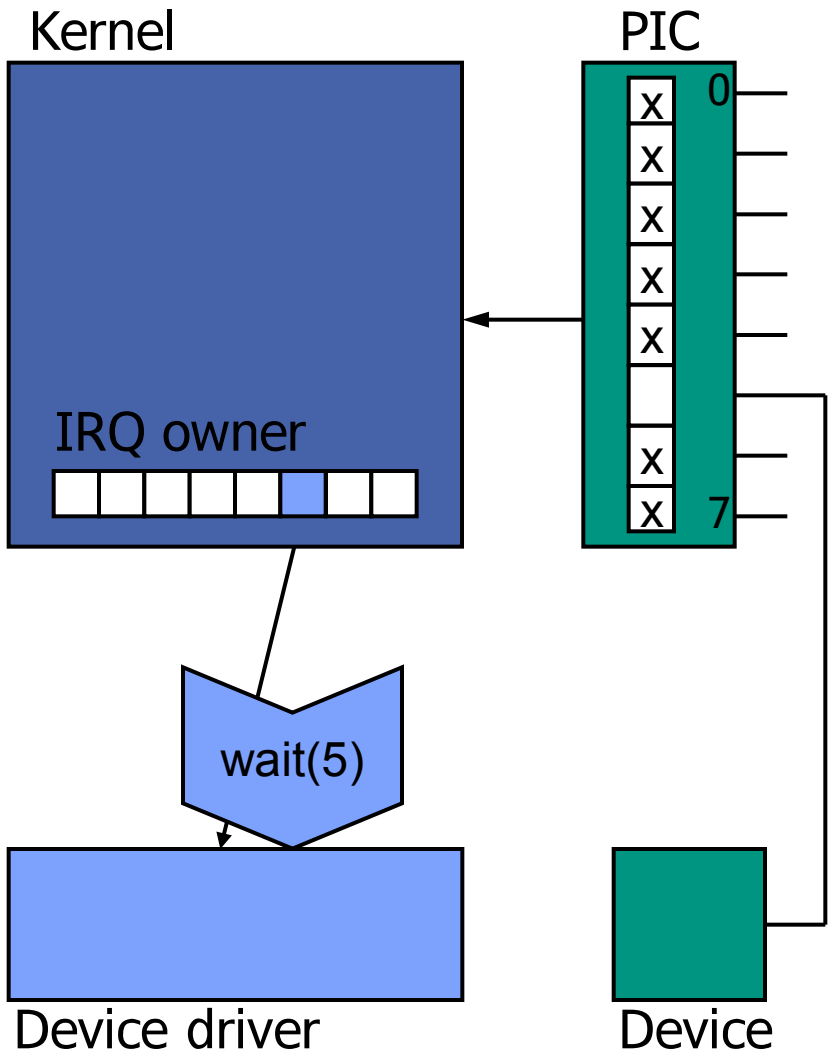
Interrupt Handling



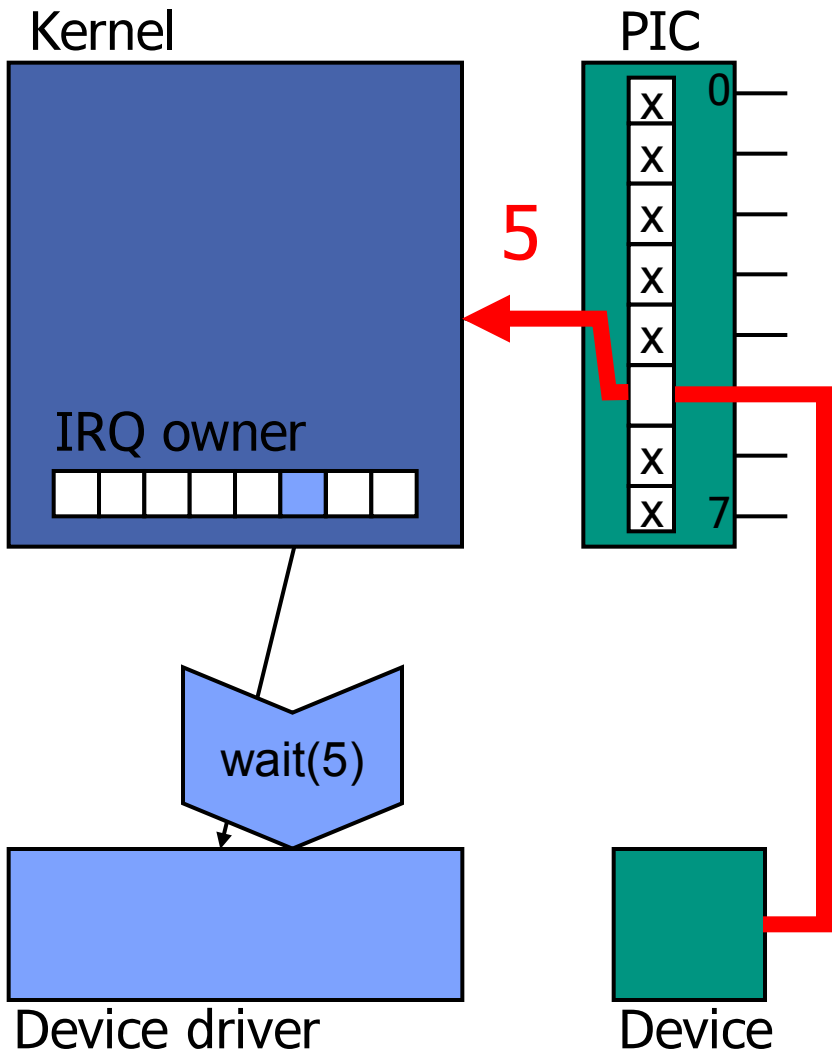
Interrupt Handling



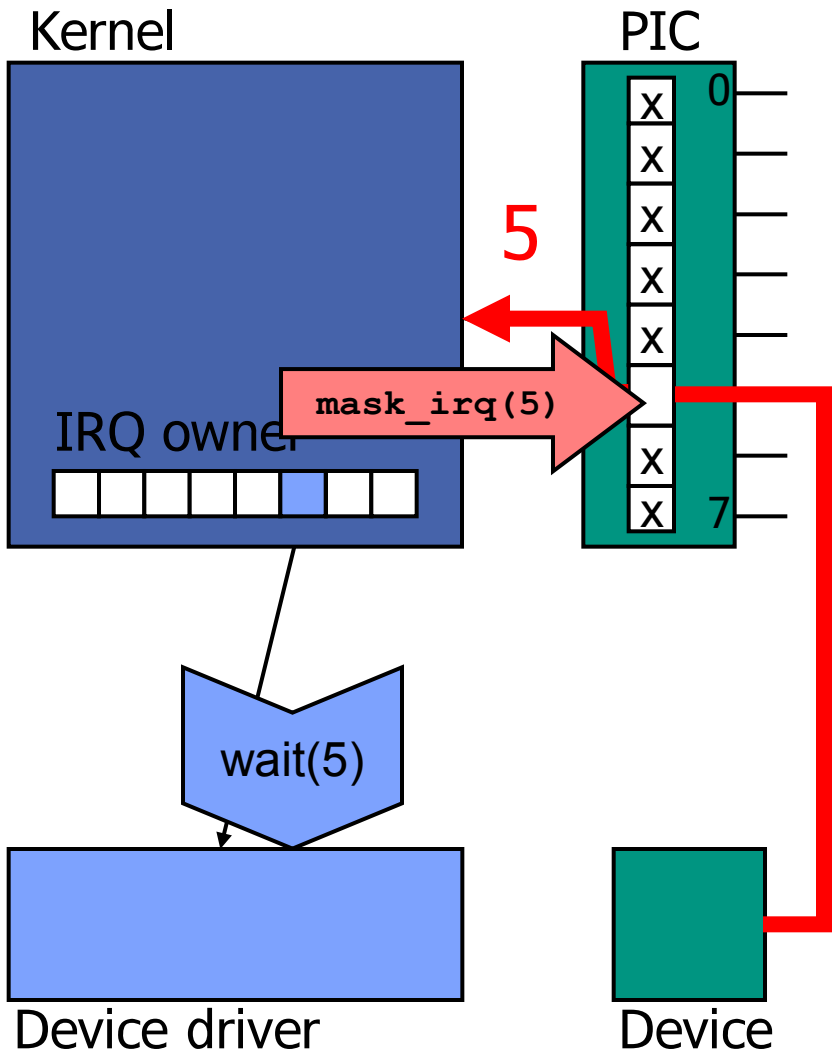
Interrupt Handling



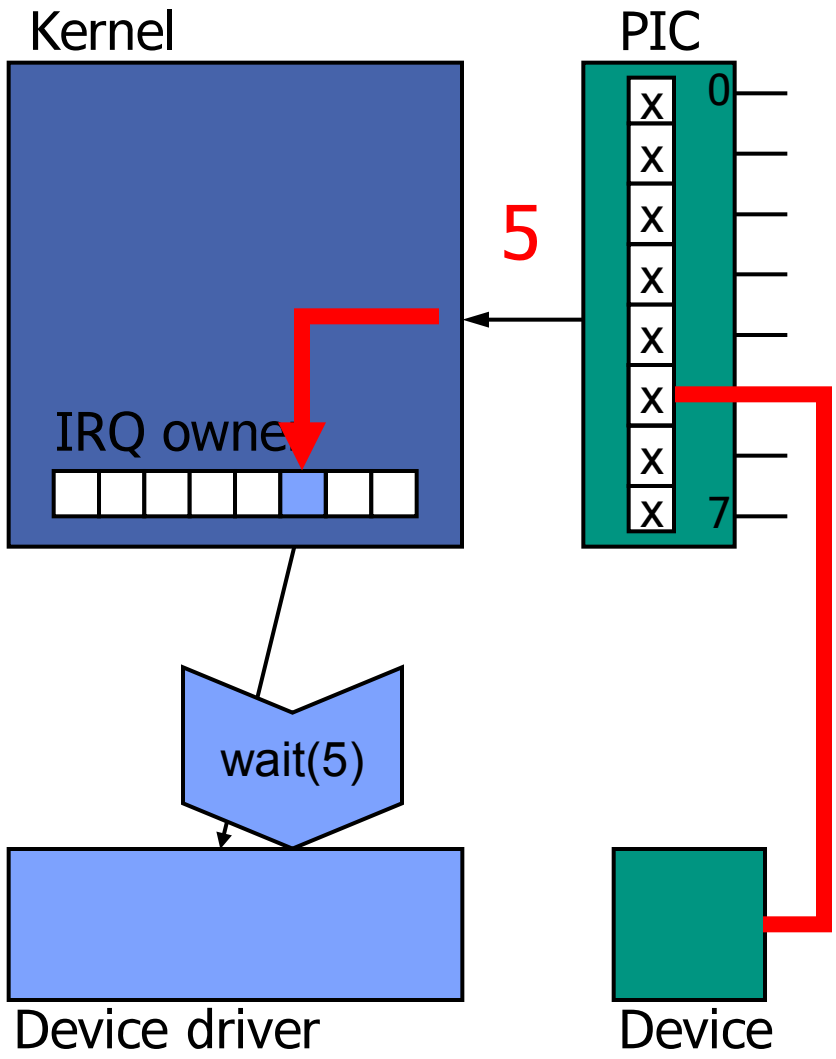
Interrupt Handling



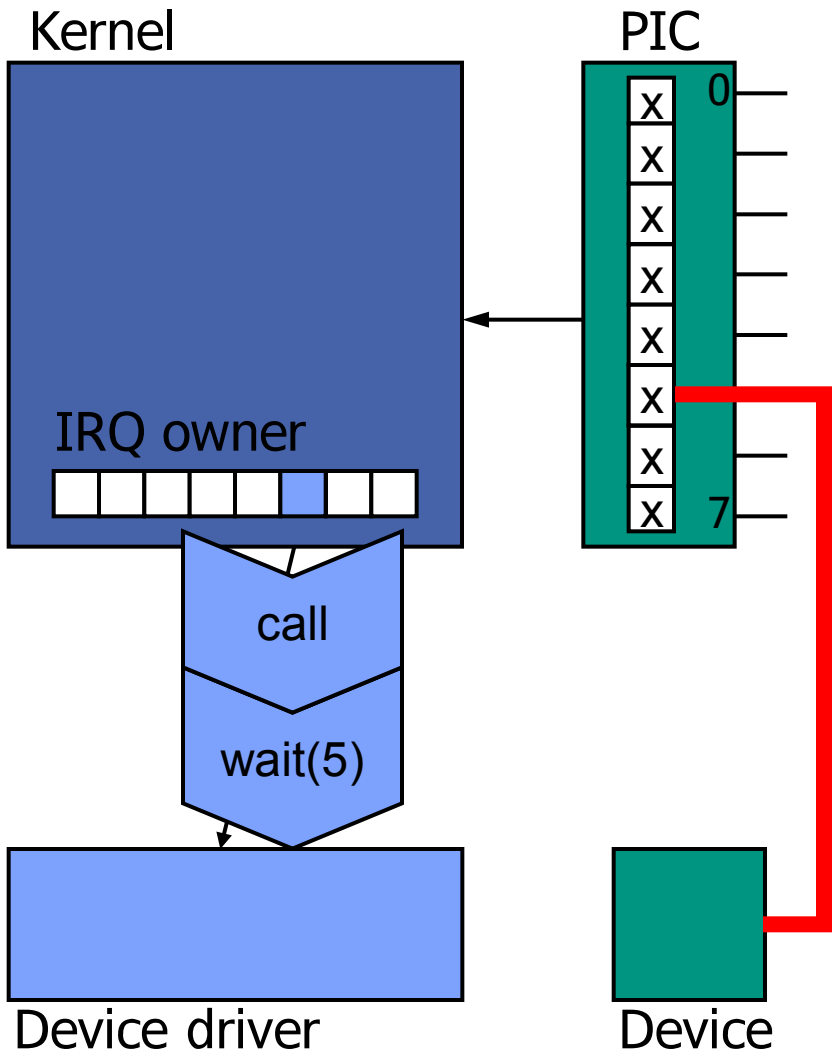
Interrupt Handling



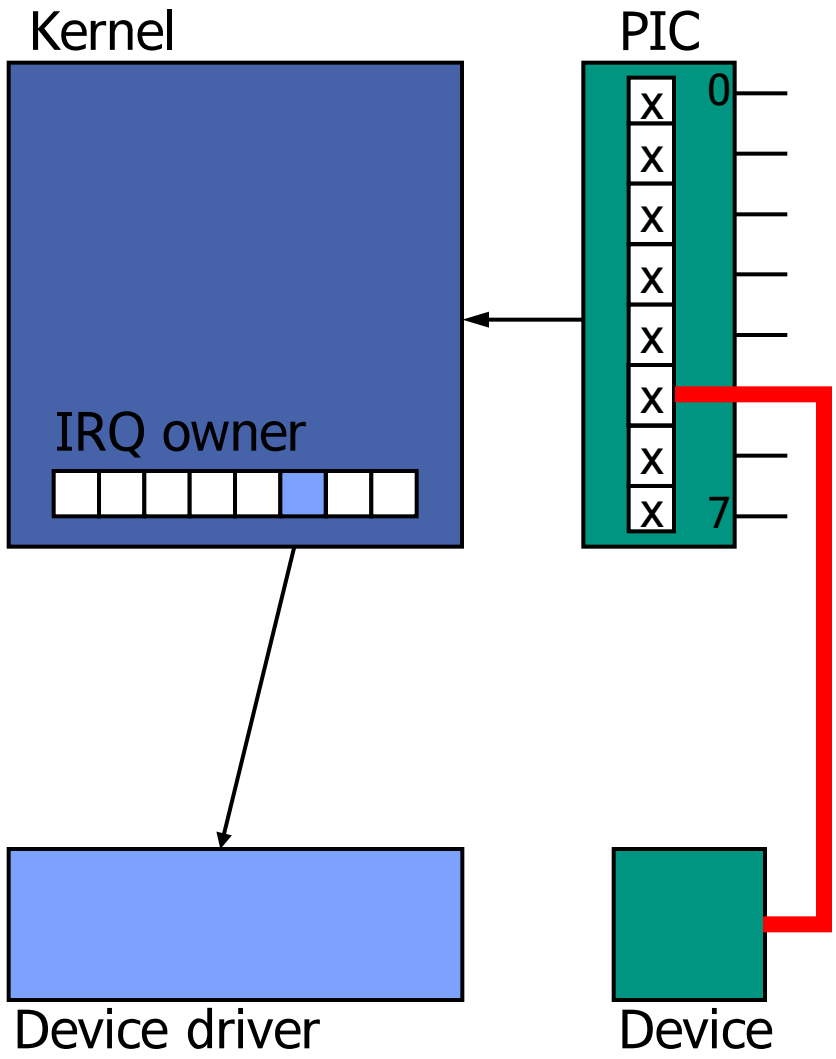
Interrupt Handling



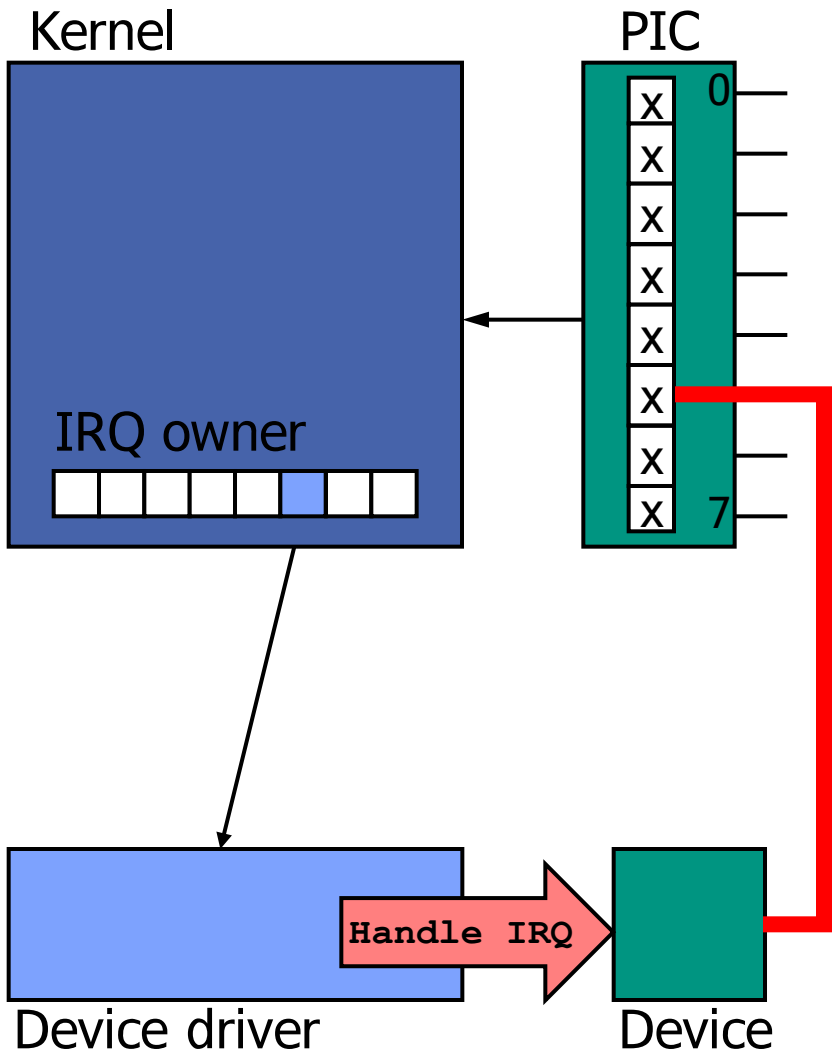
Interrupt Handling



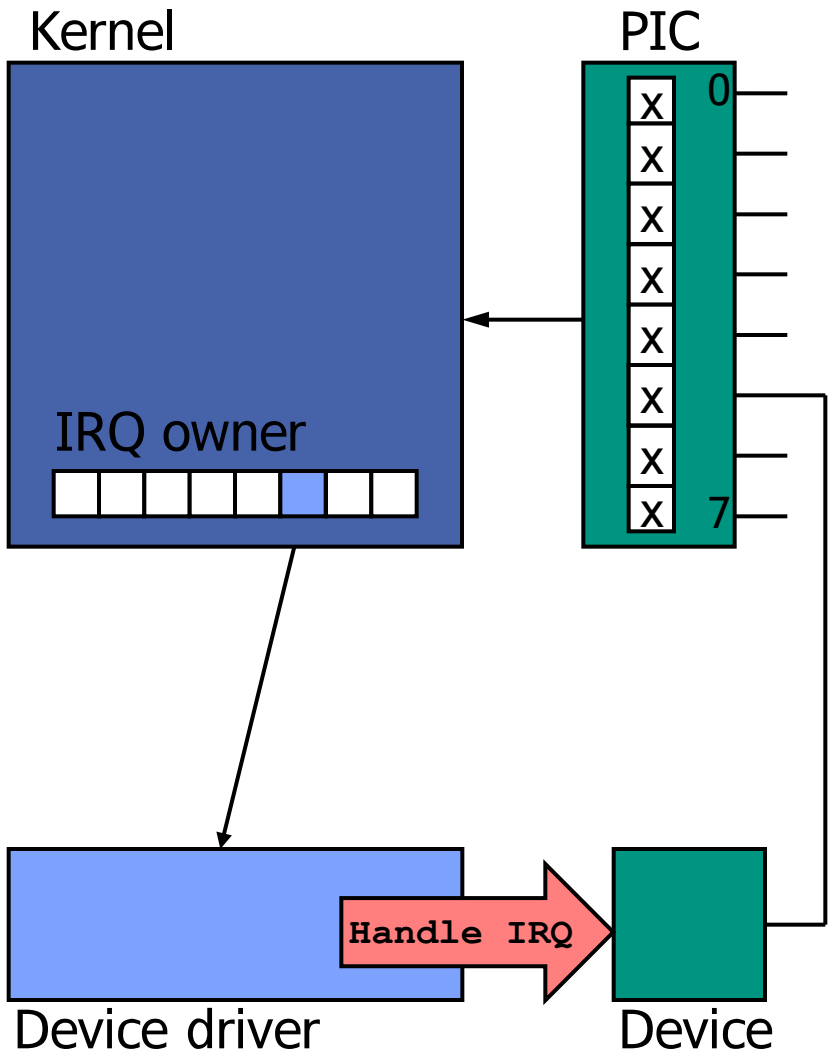
Interrupt Handling



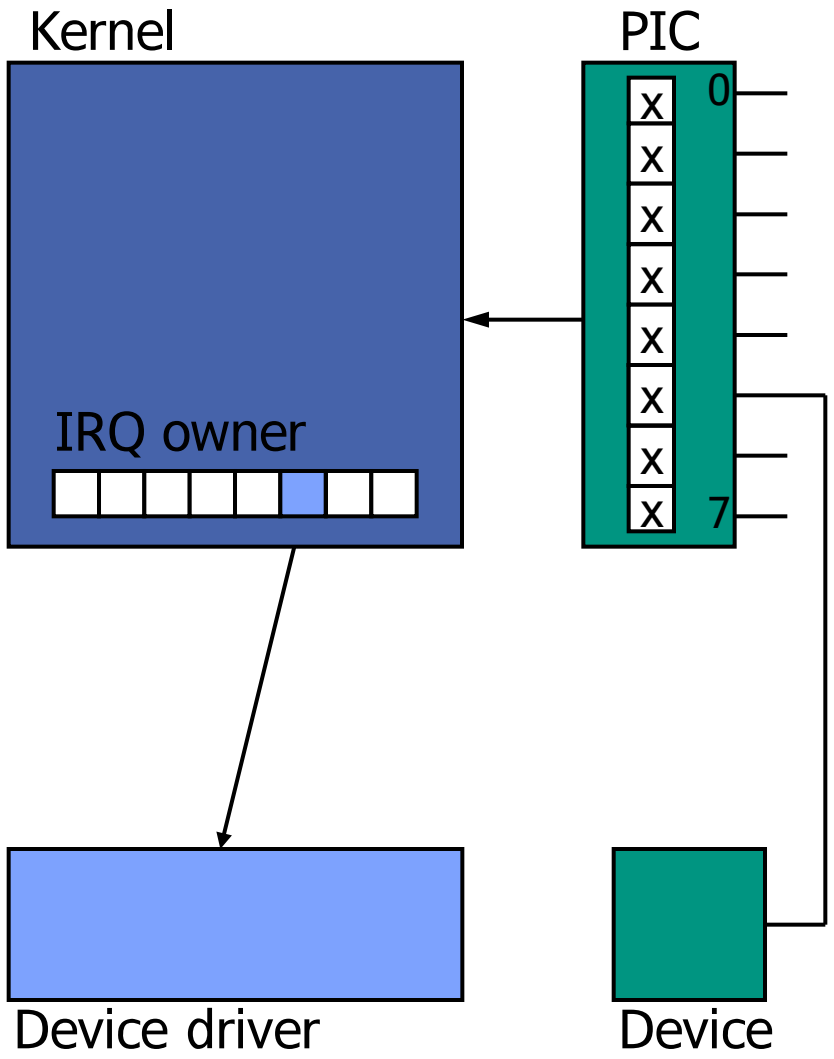
Interrupt Handling



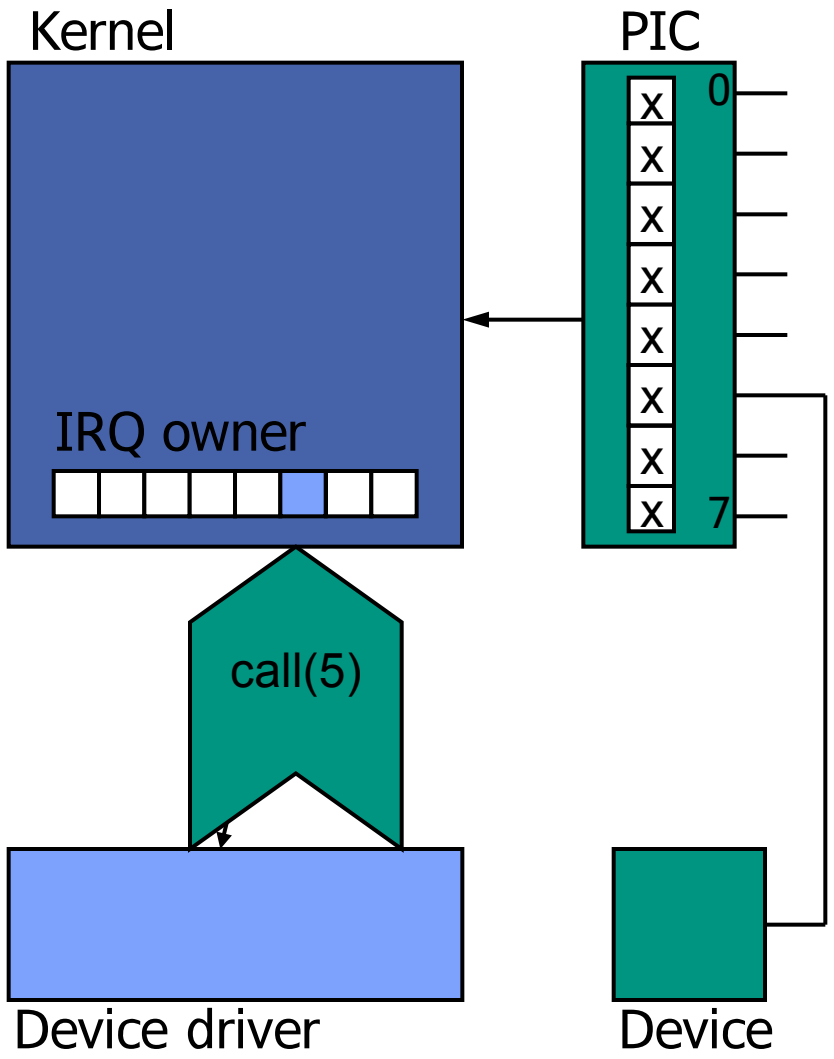
Interrupt Handling



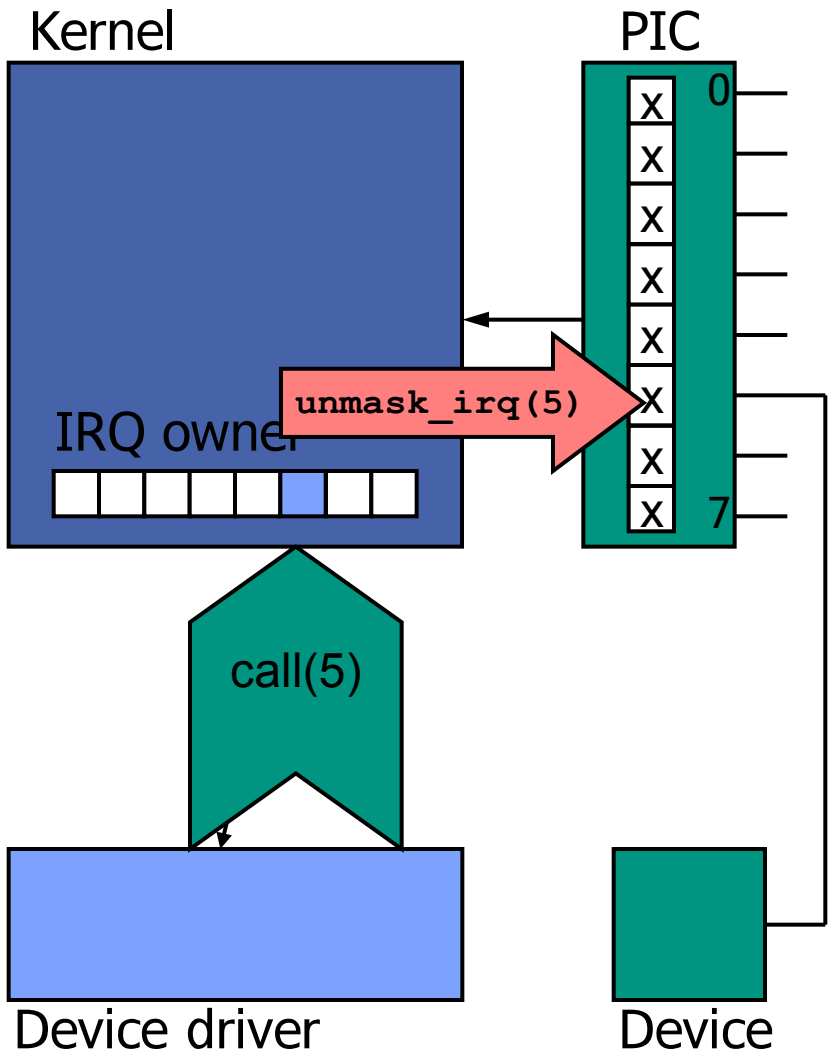
Interrupt Handling



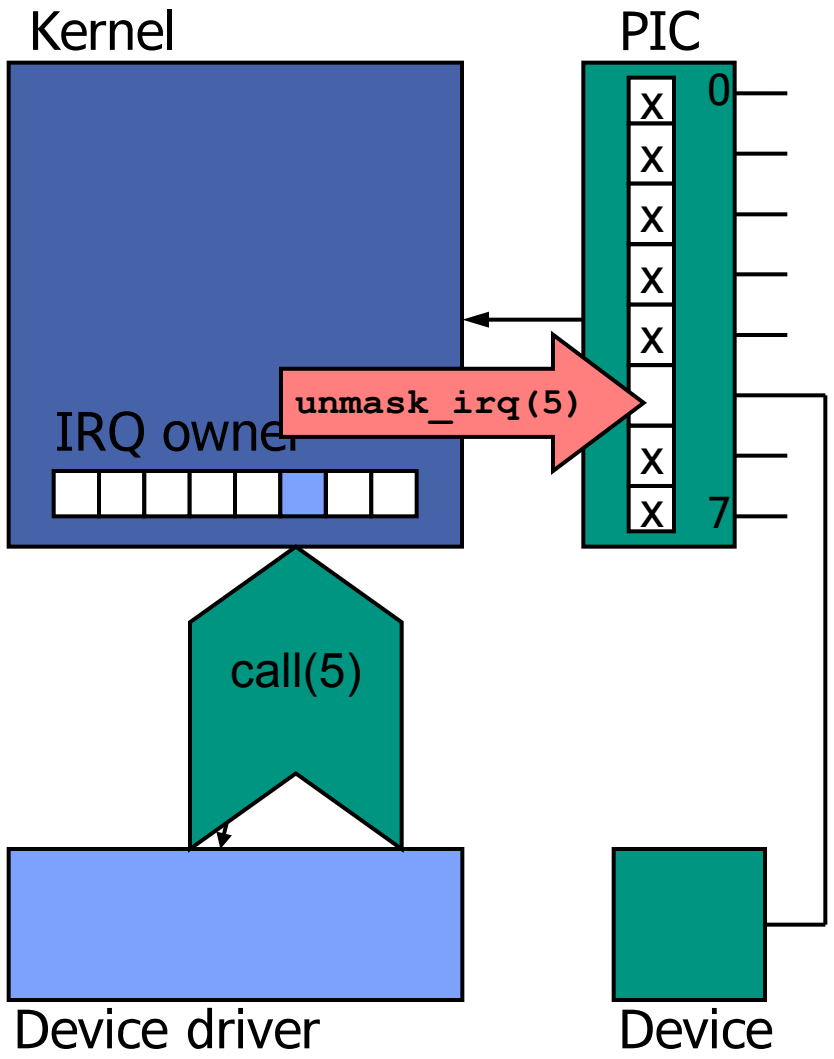
Interrupt Handling



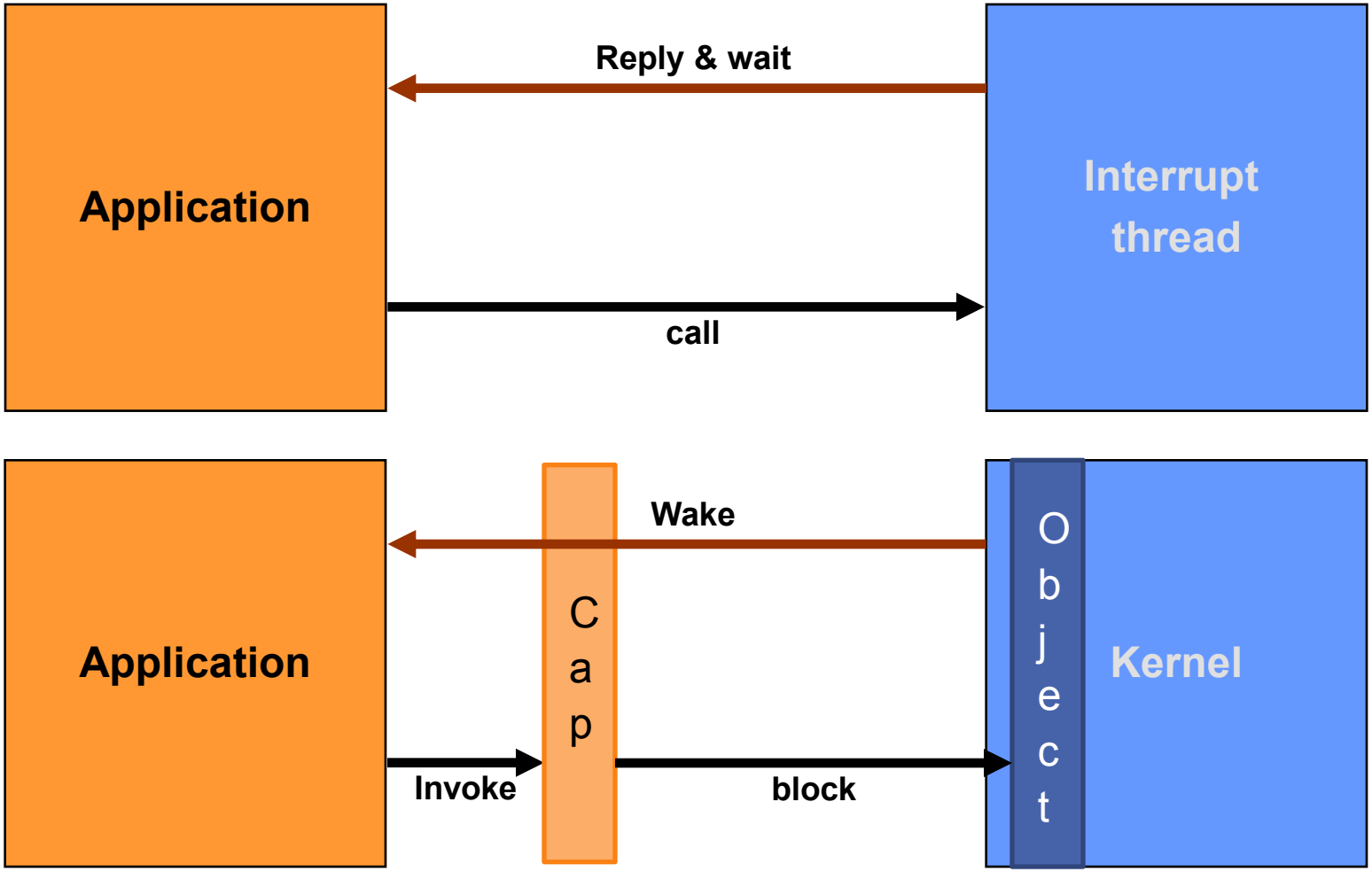
Interrupt Handling



Interrupt Handling

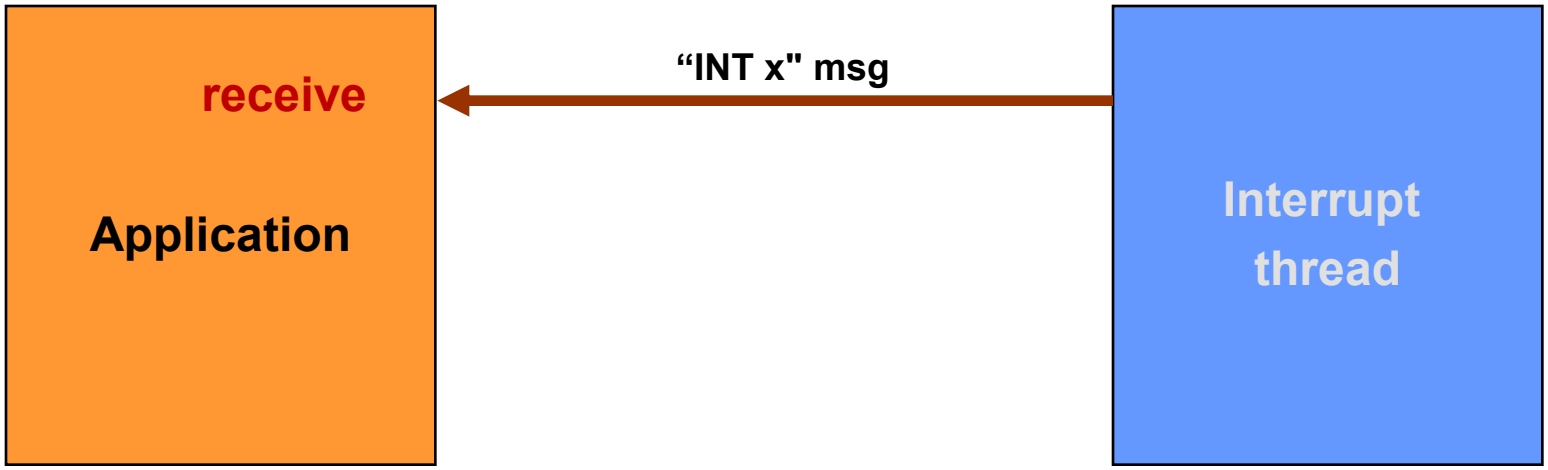


IPC (Pistachio) vs. Objects (Fiasco.OC)

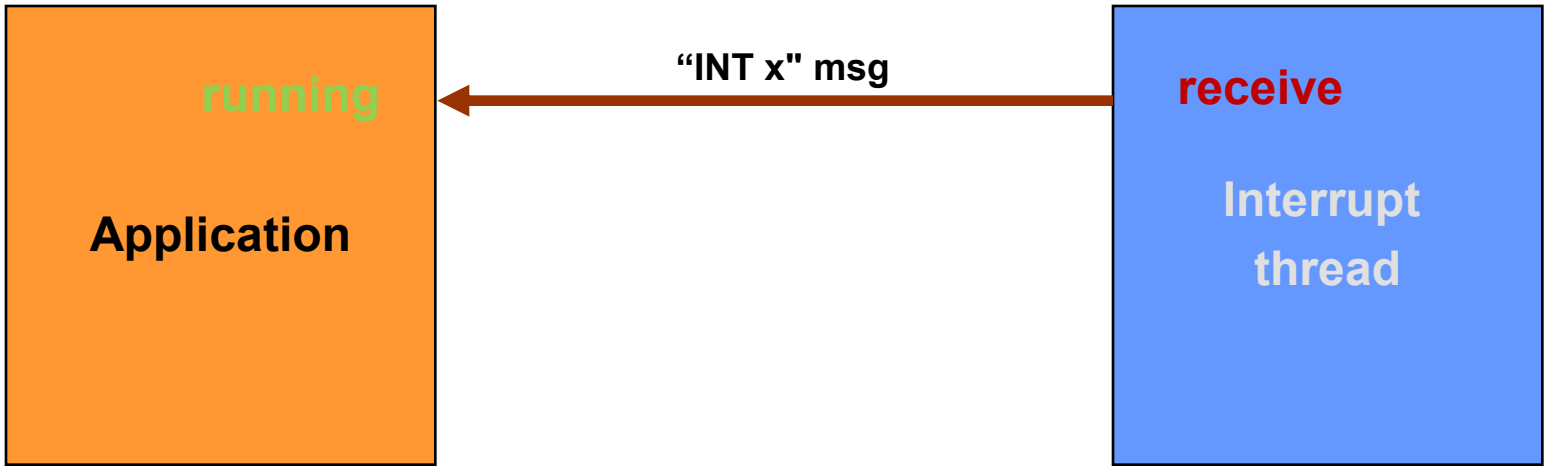


Synchronous vs. asynchronous interrupt IPC

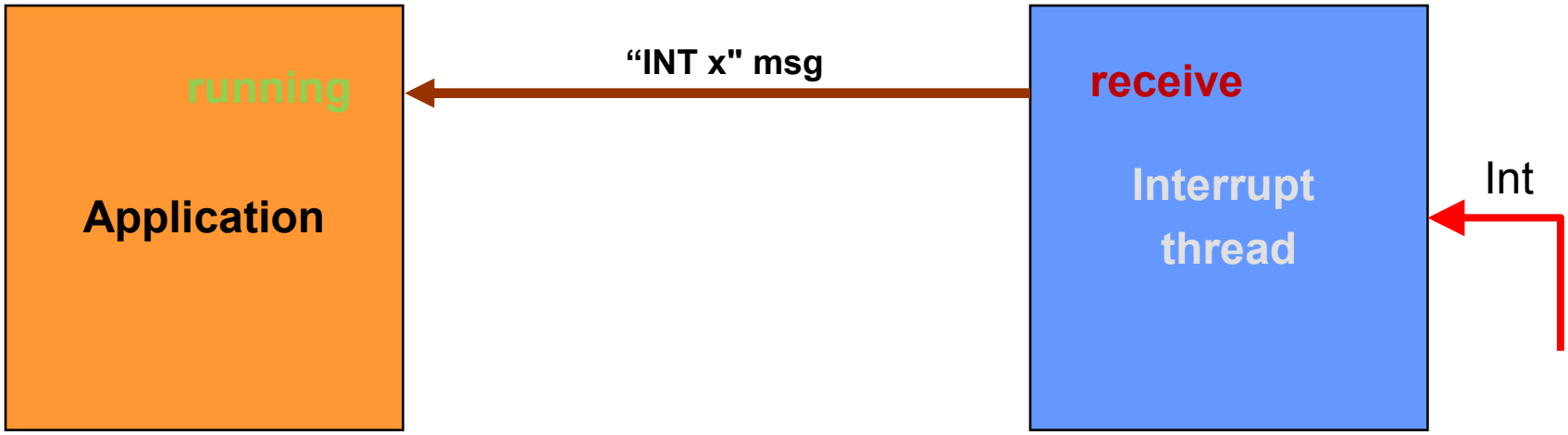
Synchronous vs. asynchronous interrupt IPC



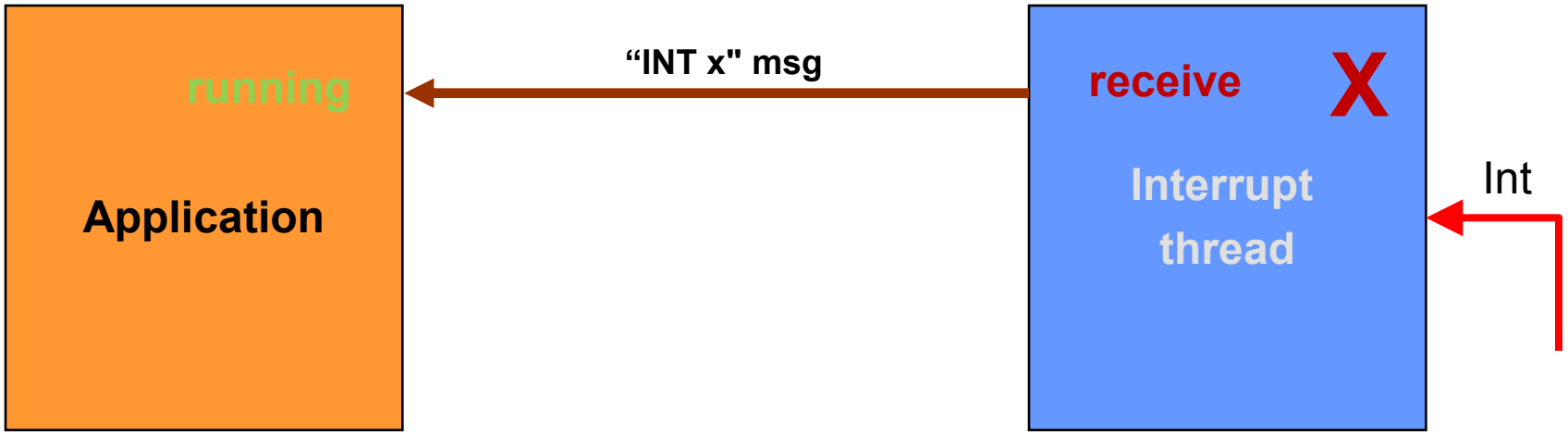
Synchronous vs. asynchronous interrupt IPC



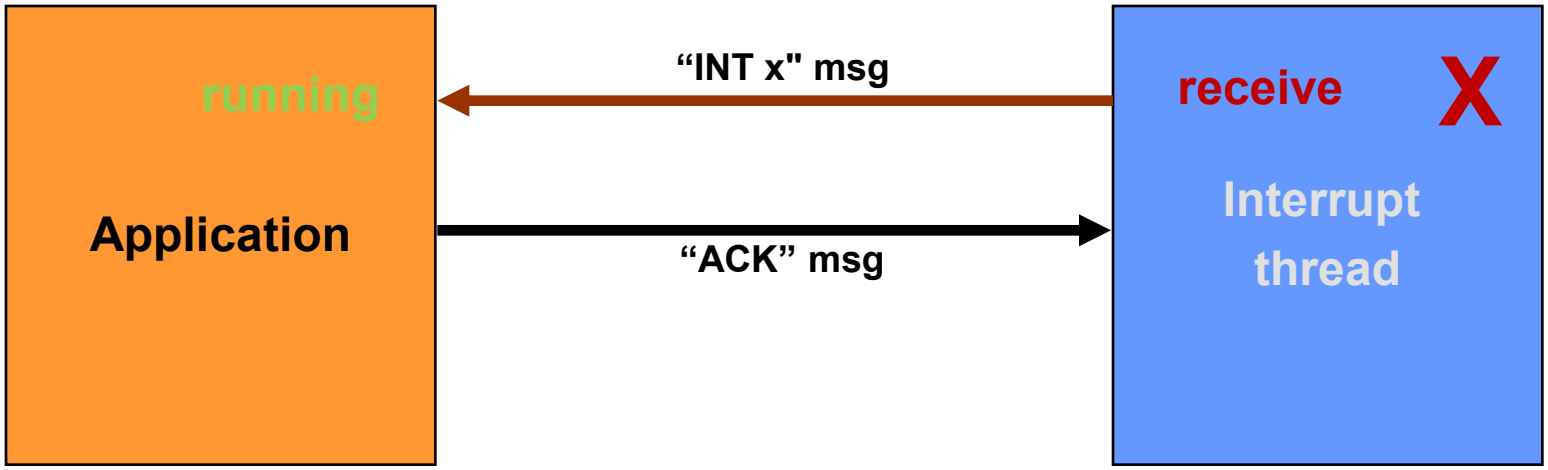
Synchronous vs. asynchronous interrupt IPC



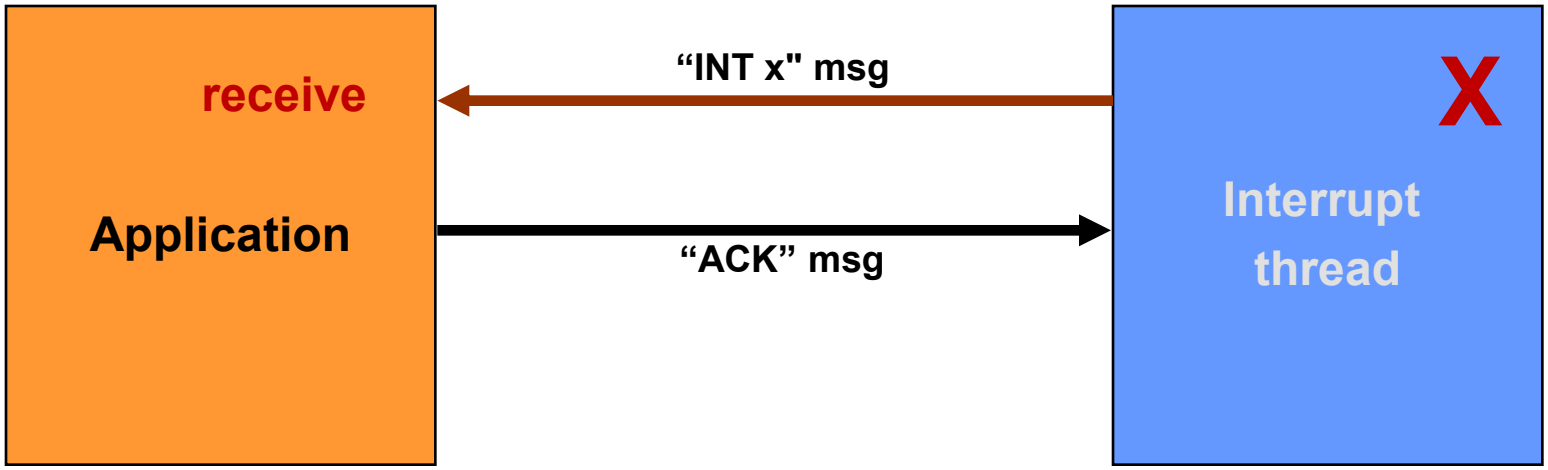
Synchronous vs. asynchronous interrupt IPC



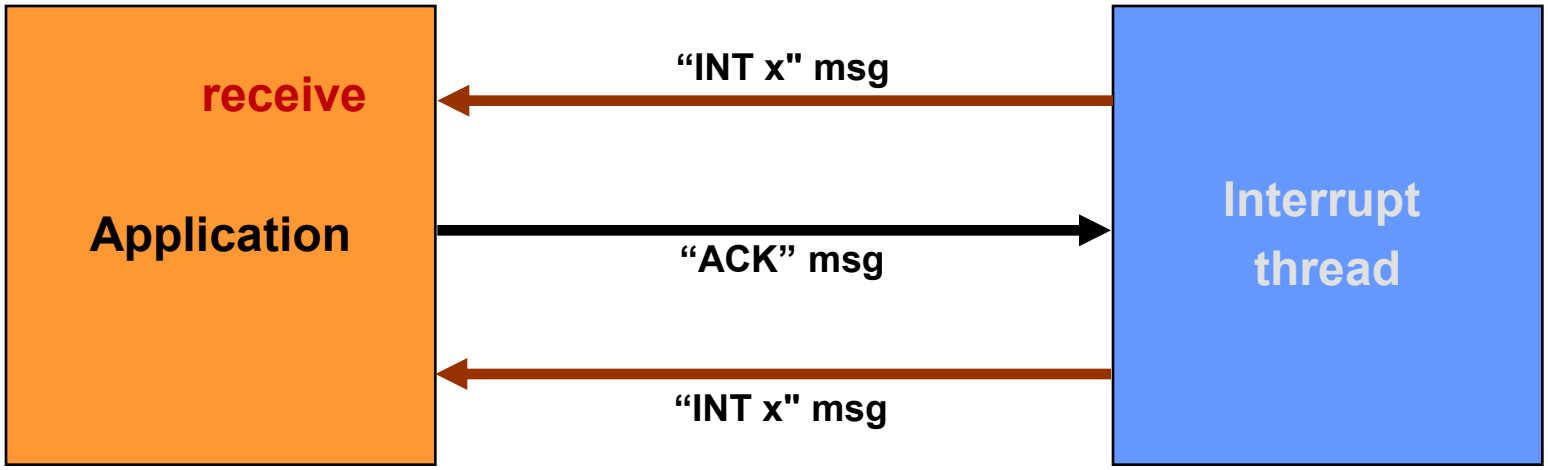
Synchronous vs. asynchronous interrupt IPC



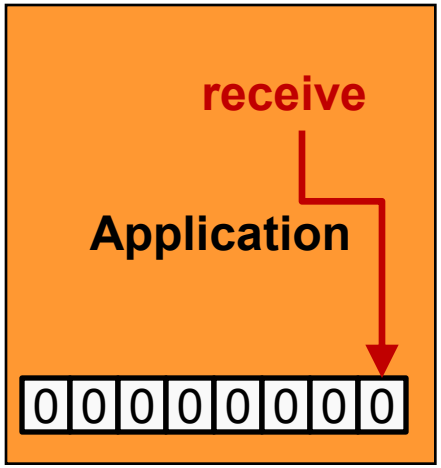
Synchronous vs. asynchronous interrupt IPC

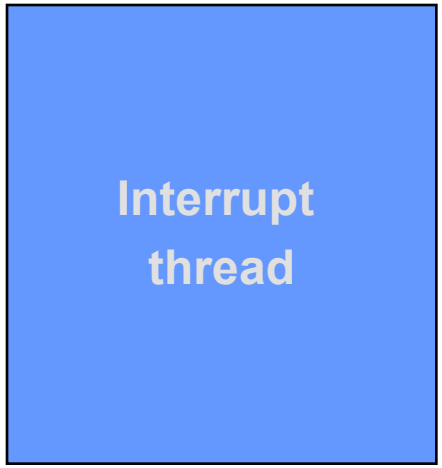


Synchronous vs. asynchronous interrupt IPC

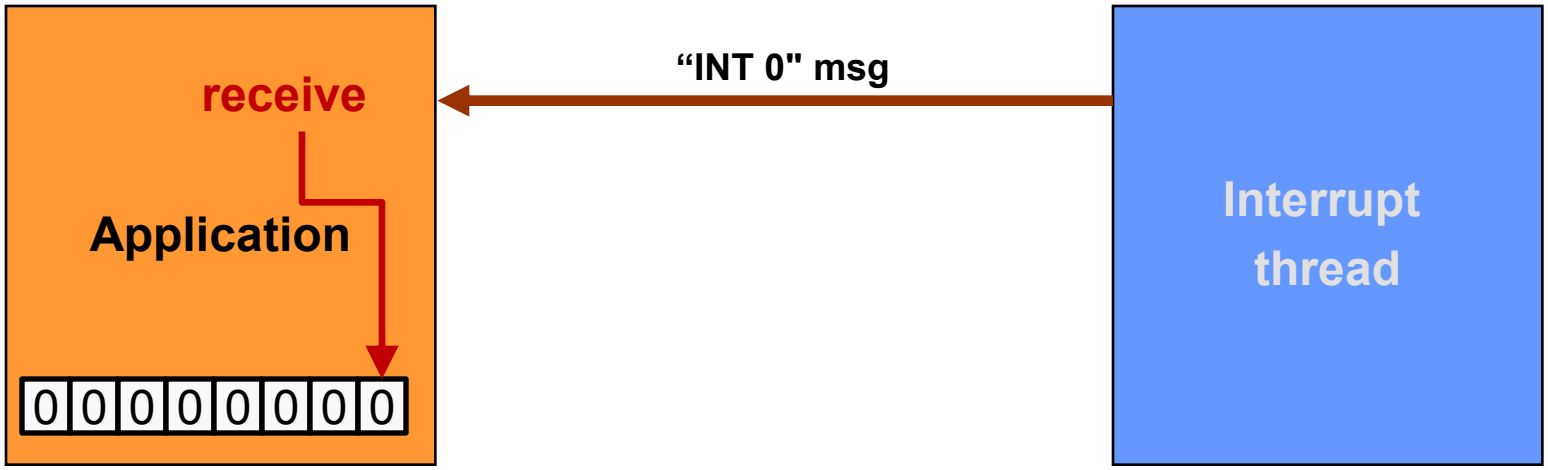


Synchronous vs. asynchronous interrupt IPC

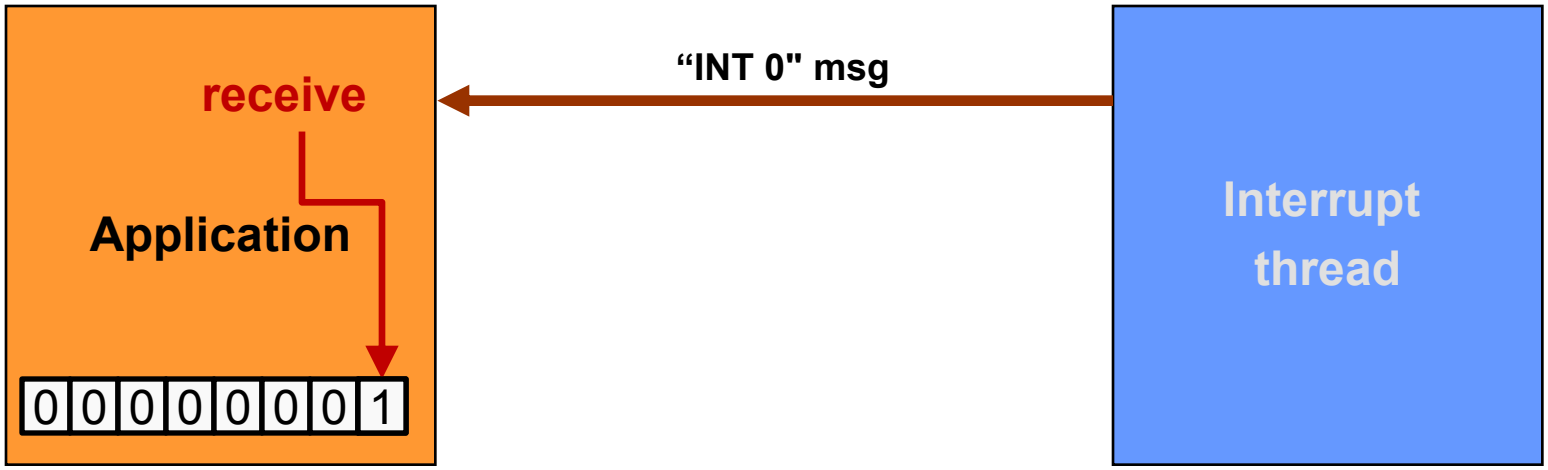




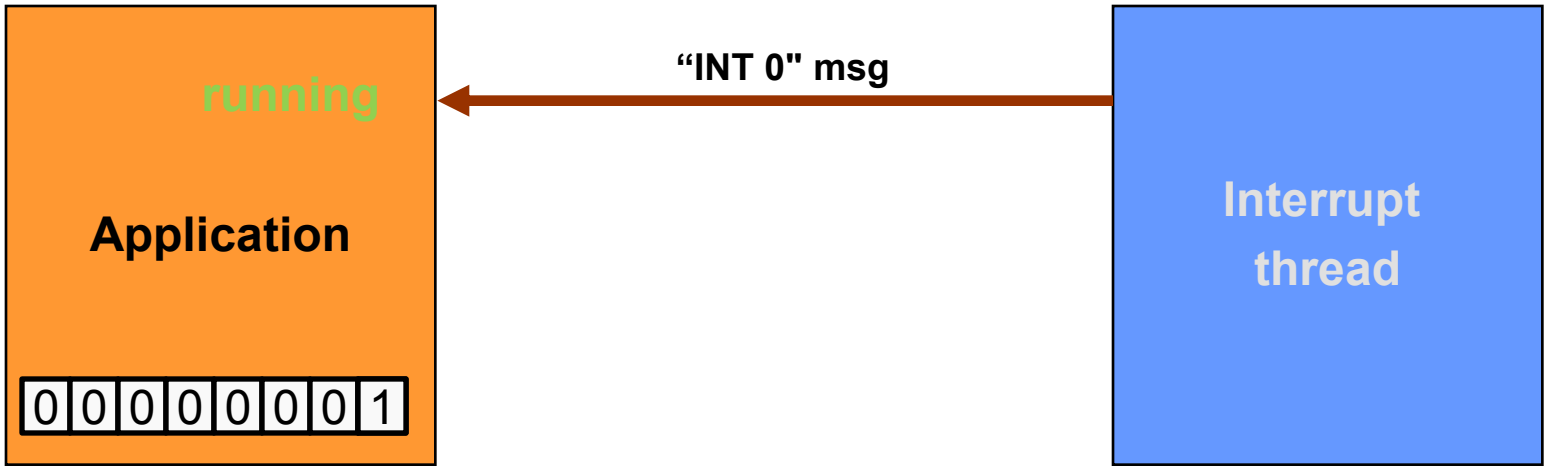
Synchronous vs. asynchronous interrupt IPC



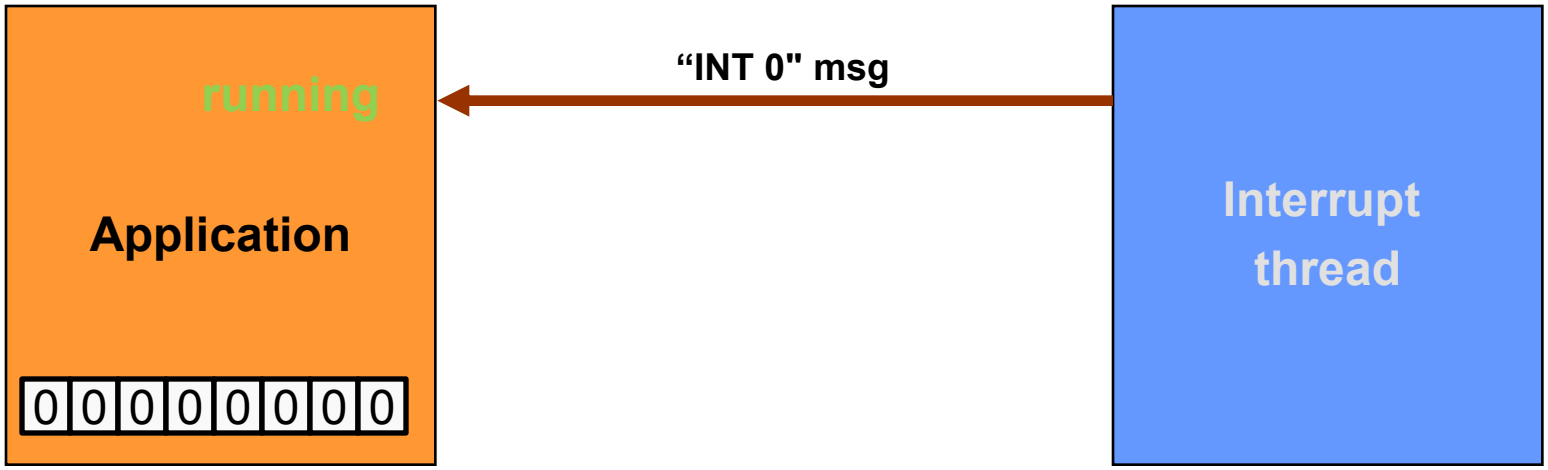
Synchronous vs. asynchronous interrupt IPC



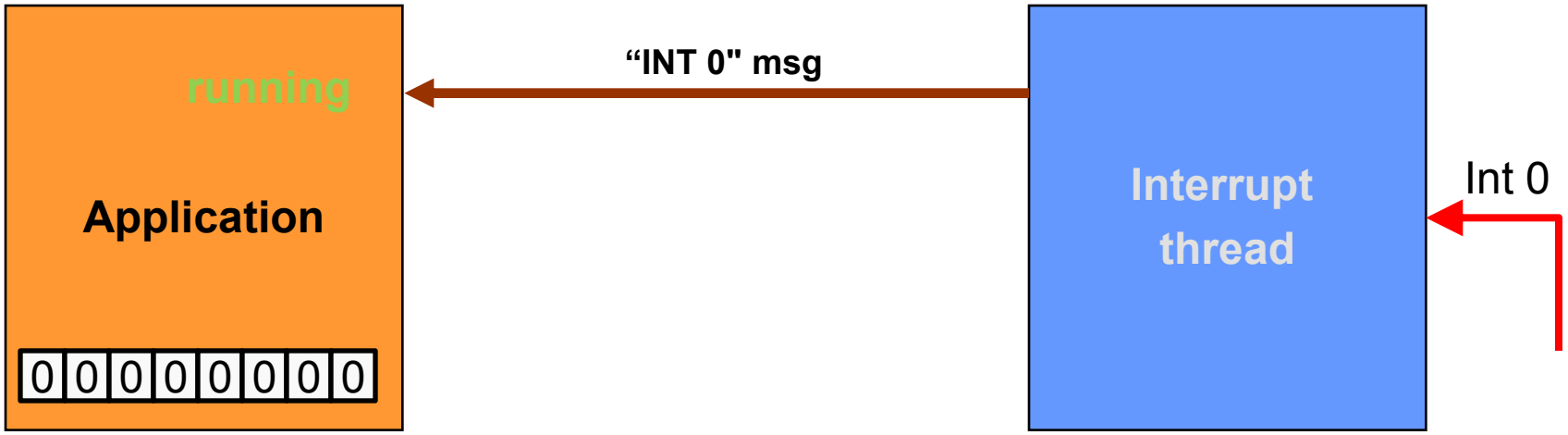
Synchronous vs. asynchronous interrupt IPC



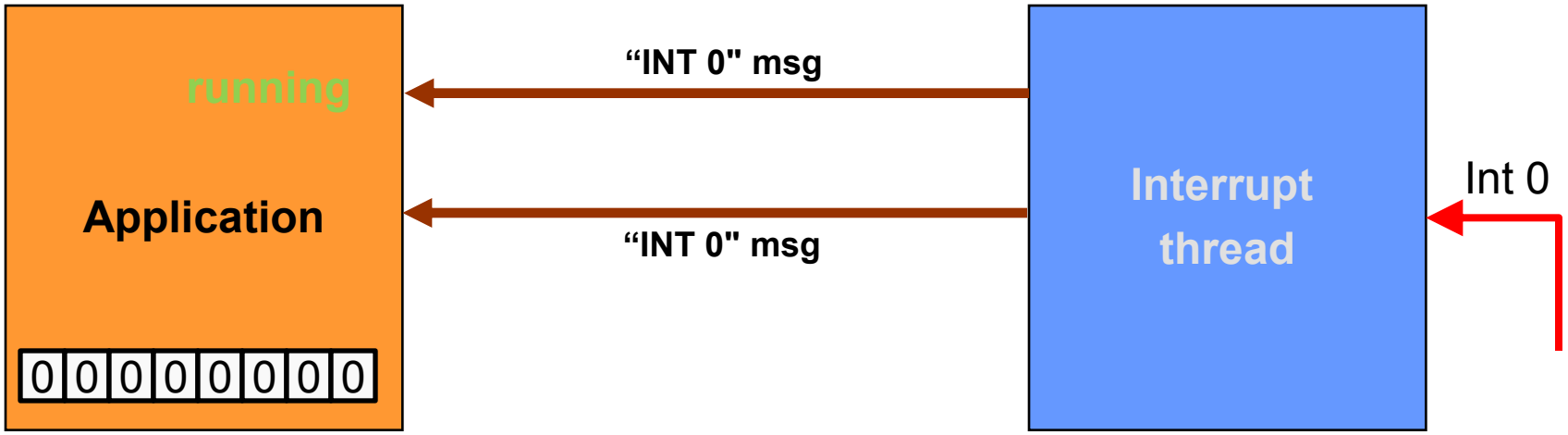
Synchronous vs. asynchronous interrupt IPC



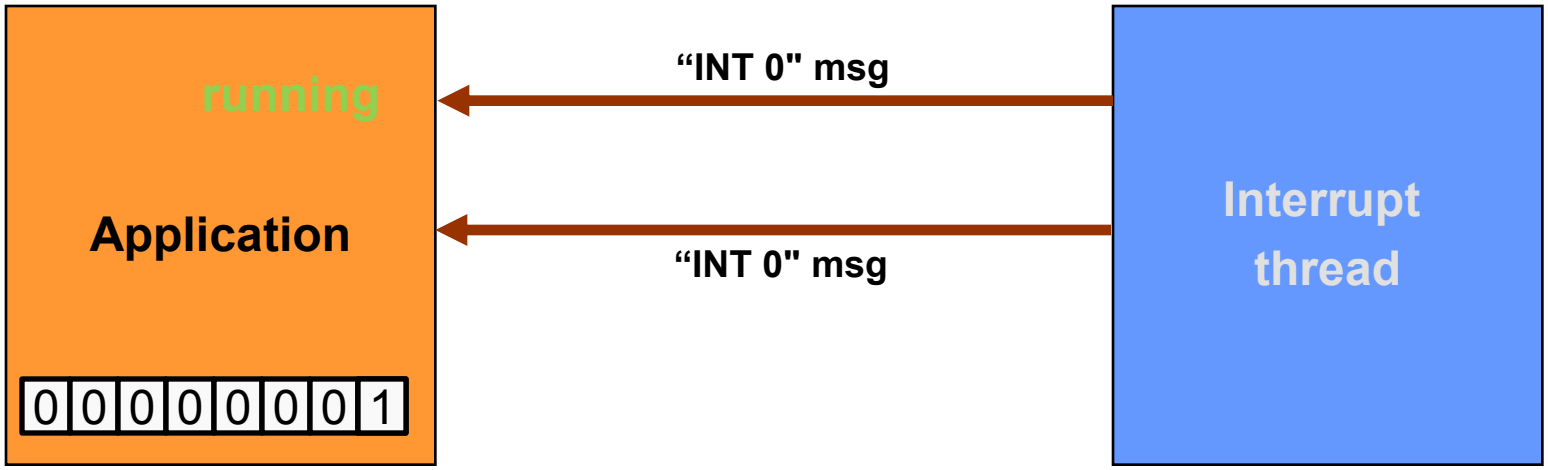
Synchronous vs. asynchronous interrupt IPC



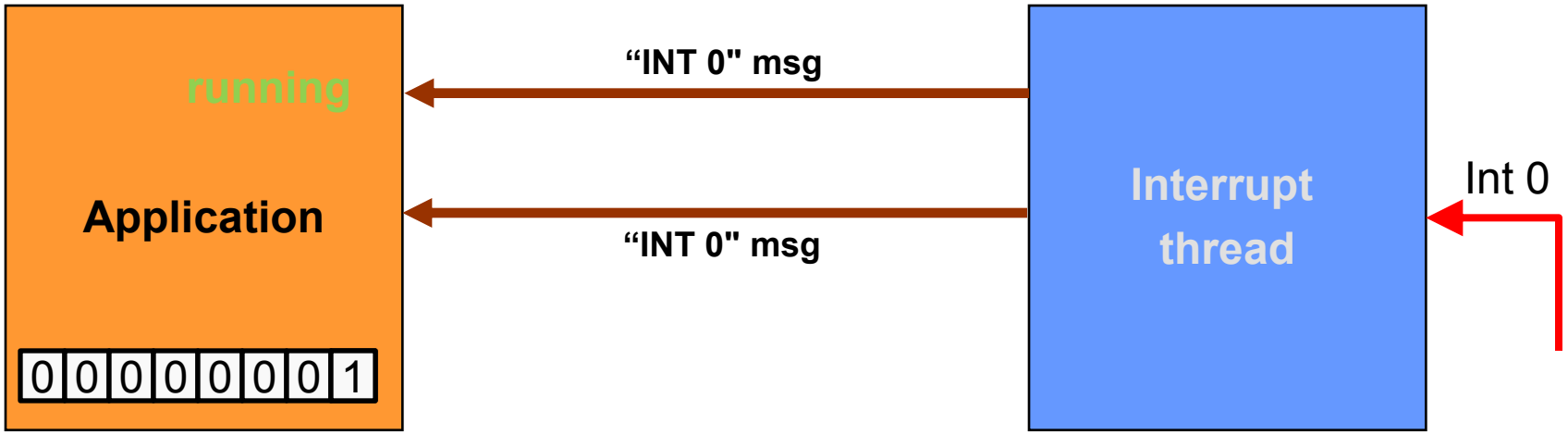
Synchronous vs. asynchronous interrupt IPC



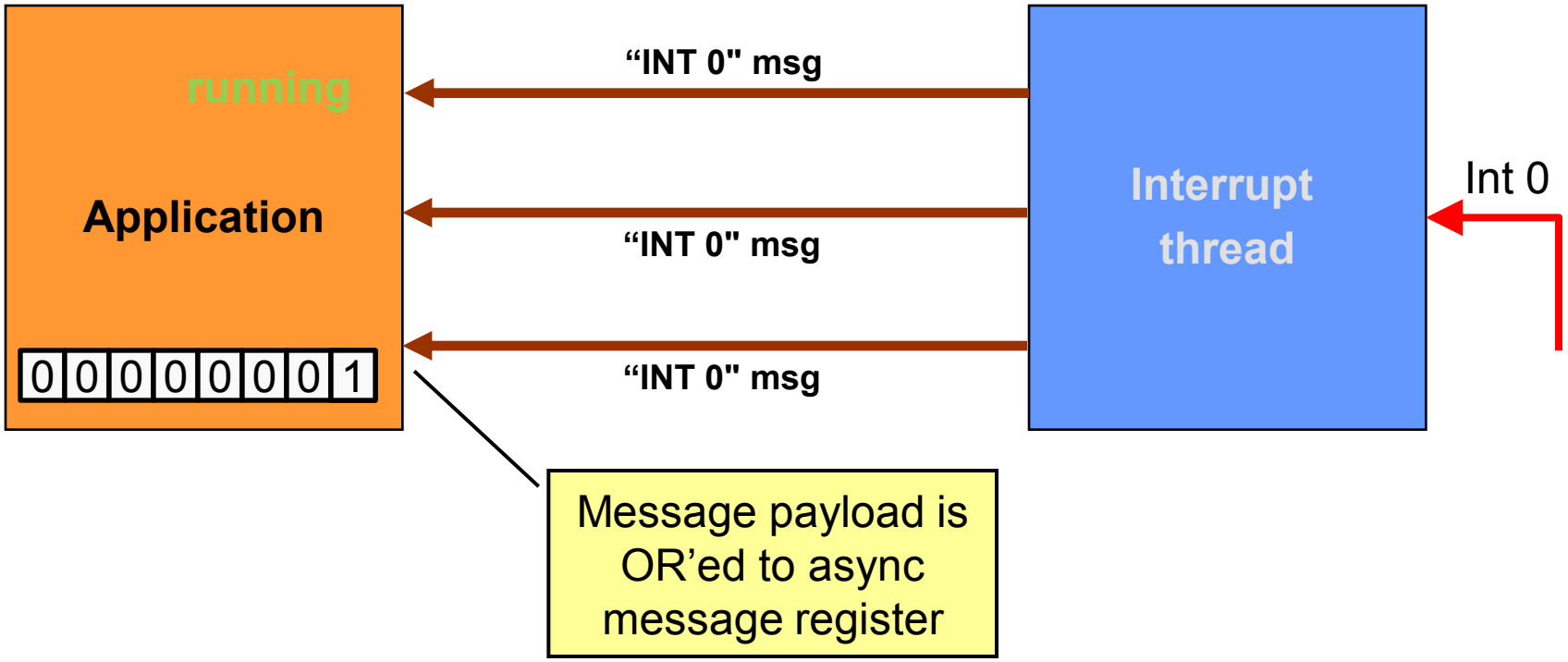
Synchronous vs. asynchronous interrupt IPC



Synchronous vs. asynchronous interrupt IPC



Synchronous vs. asynchronous interrupt IPC



Invisible Interrupts

- Kernel uses some interrupts for itself
  - Timer tick – triggers scheduler
    - Timer device and interrupt line not available to user
    - Remember soon/late/late late lists?
  - Inter-processor interrupts (SMP)
    - Kernel hides IPI hardware
    - Cross-CPU user communication via IPC
- Kernel debugger may use interrupts
  - Performance counters – profiling
  - NMI – last resort debug aid

L4 Kernel Paradigm

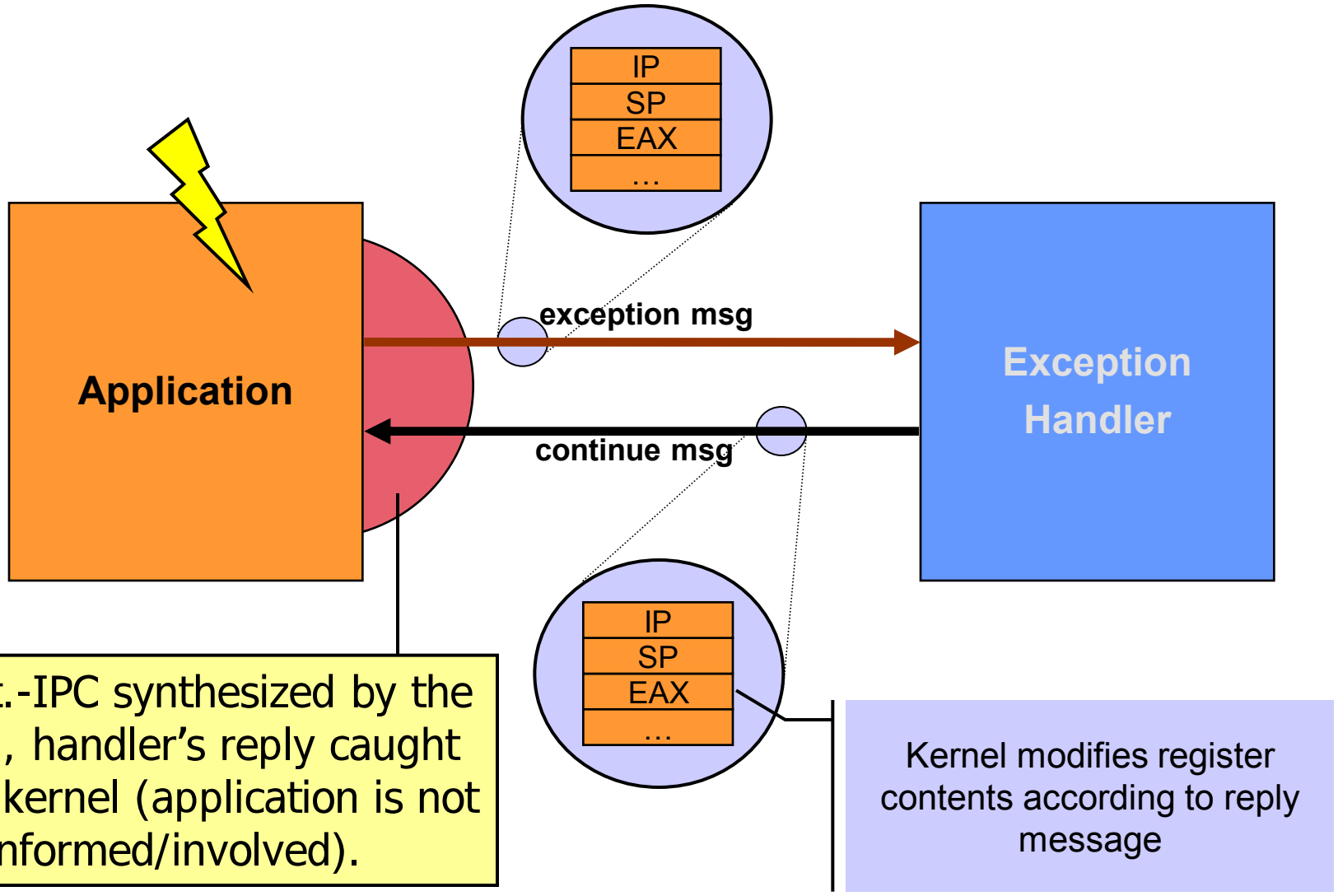
Everything the kernel needs to handle in a secure manner will either become invisible or be hidden behind an abstraction.

- Events that are not handled by the kernel itself will be posted to user land
  - Page faults
  - Hardware interrupts
  - Exceptions

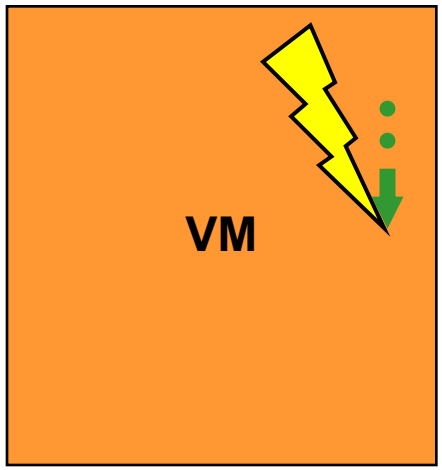
Old Exception Handling Model

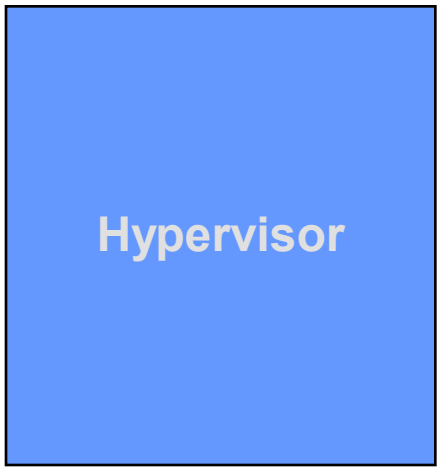
- Model
  - Create exception frame on user stack
  - Restart thread at a predefined exception handler
  - Return from exception handler using special instruction
- Problems
  - Very x86-ish, inconsistent
  - Requires a valid user stack
  - Poor performance for virtualization
    - Too many kernel entries
  - Recursive exception handling?
  - Safety?

New Exception Handling Model

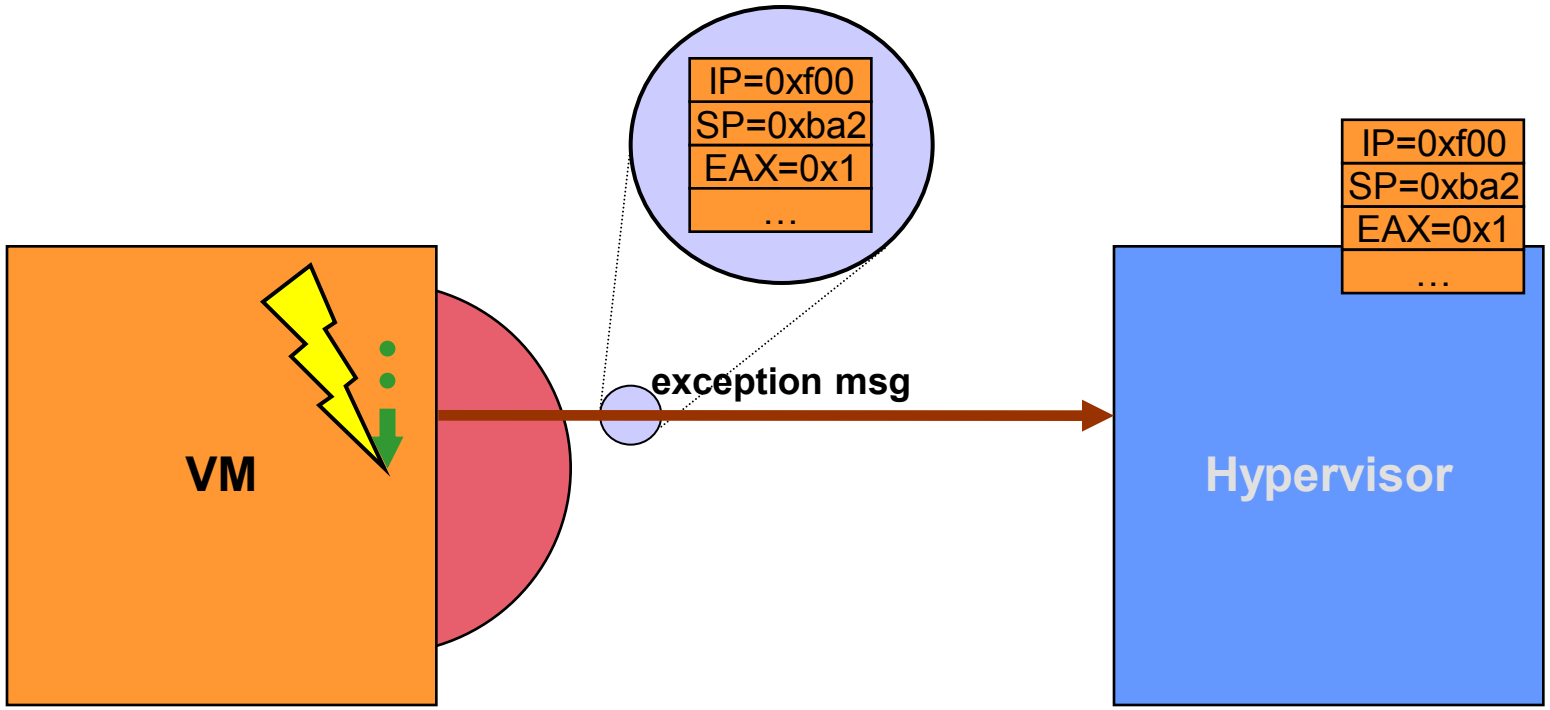


Example: Virtualization

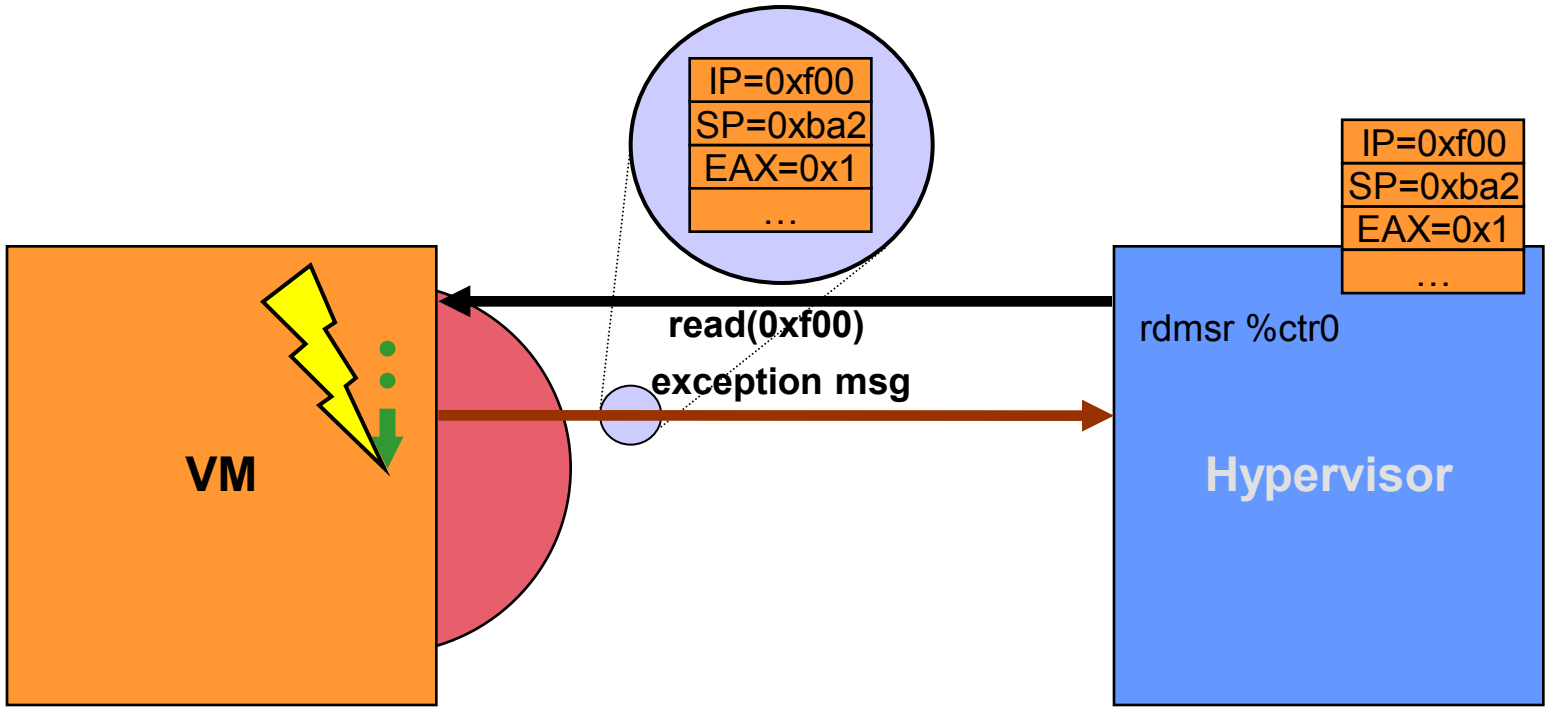




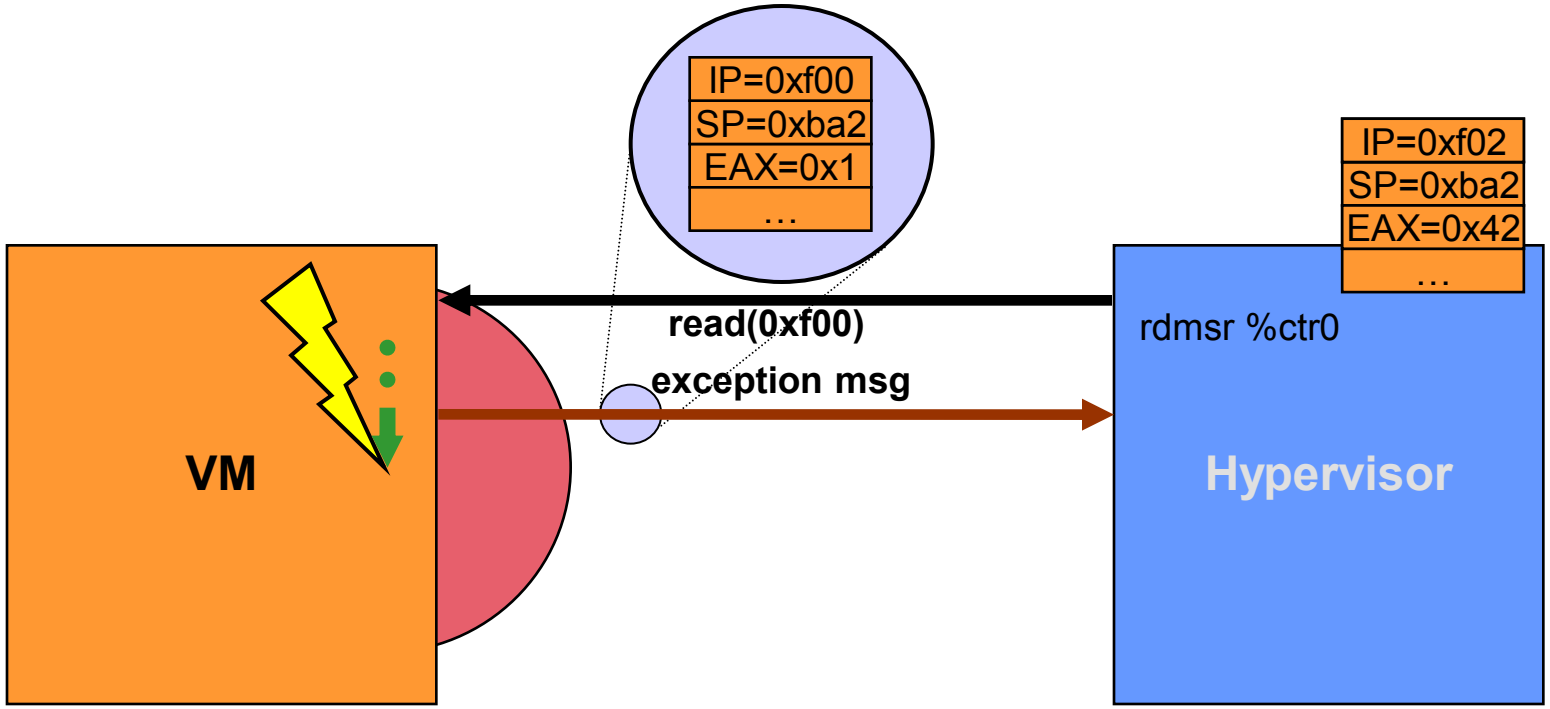
Example: Virtualization



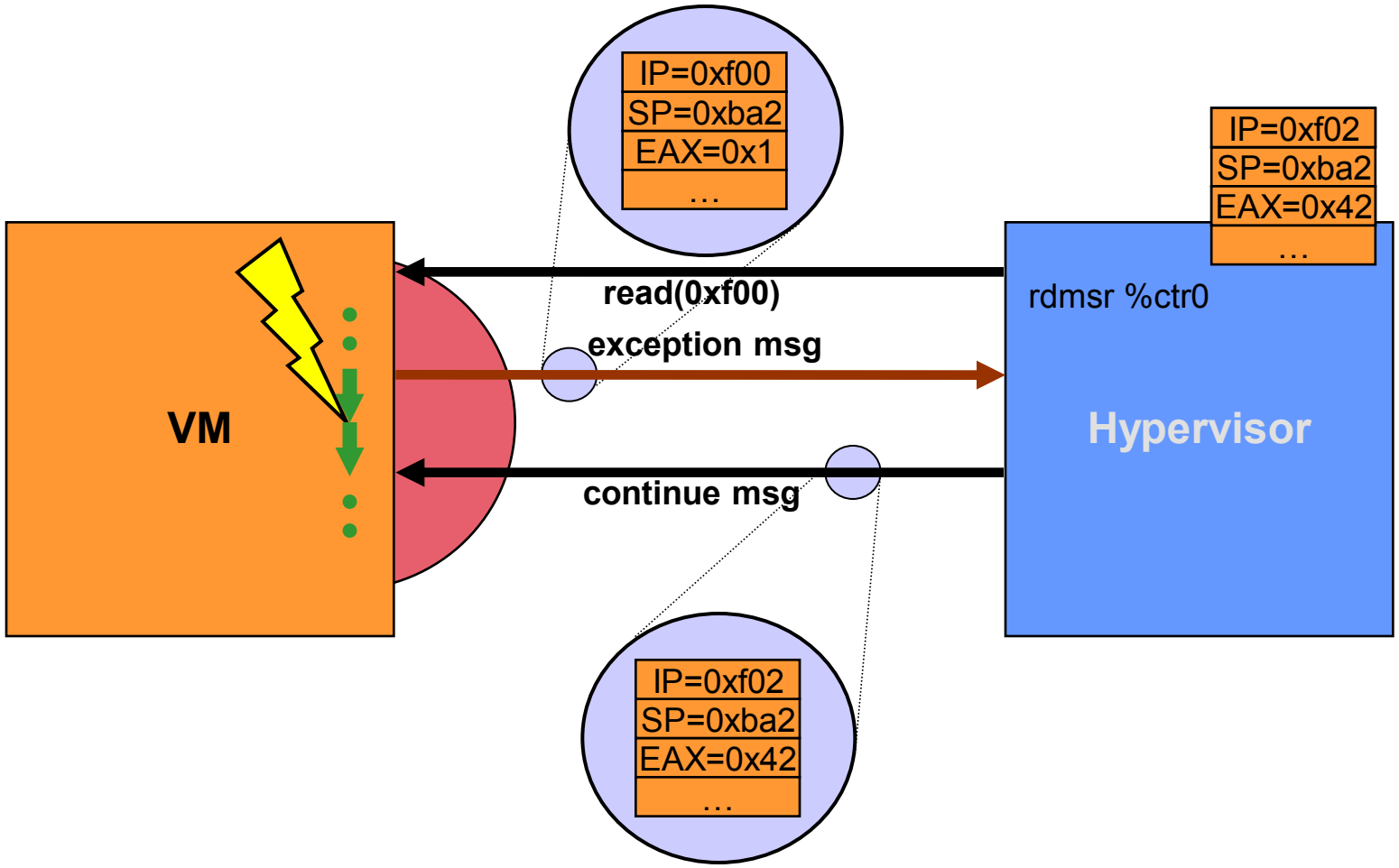
Example: Virtualization



Example: Virtualization



Example: Virtualization



Invisible Exceptions

- Kernel handles some exceptions internally
  - Coprocessor/FPU virtualization
  - Transparent small space extension
  - TLB misses with software-loaded TLBs
- Kernel debugger handles some exceptions
  - Breakpoints
  - Single-stepping
  - Branch tracing

Review: Processor Multiplexing

- Hardware model
  - One thread
  - One address space
  - Exclusive access to resources (such as FPU)
- Microkernel exports
  - Multiple threads
  - Multiple address spaces
  - Maintain threads' view of the world
    - Threads have exclusive access to resources
- Multiplex abstractions onto existing hardware
  - Switch register file contents at thread switch
  - Potentially switch MMU state at thread switch
  - Switch FPU content etc. at thread switch

FPU Virtualization

- Strict switching

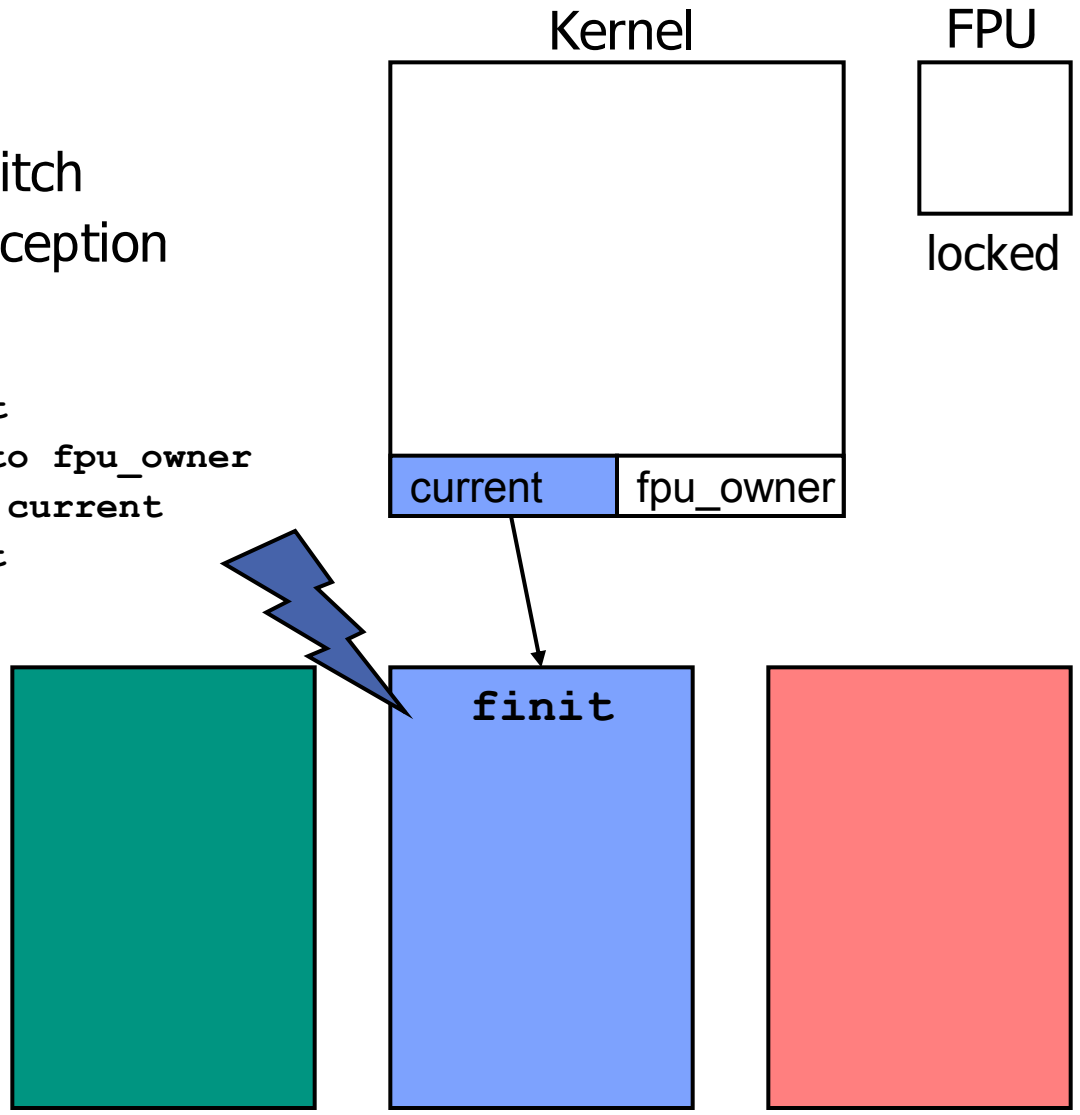
```
Thread switch:
    Store current thread's FPU state
    Load new thread's FPU state
```

  - Extremely expensive
    - IA-32's full SSE2 state is 512 Bytes
    - IA-64's floating point state is ~1.5 KiB
  - May not even be required
    - Threads do not always use FPU

Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

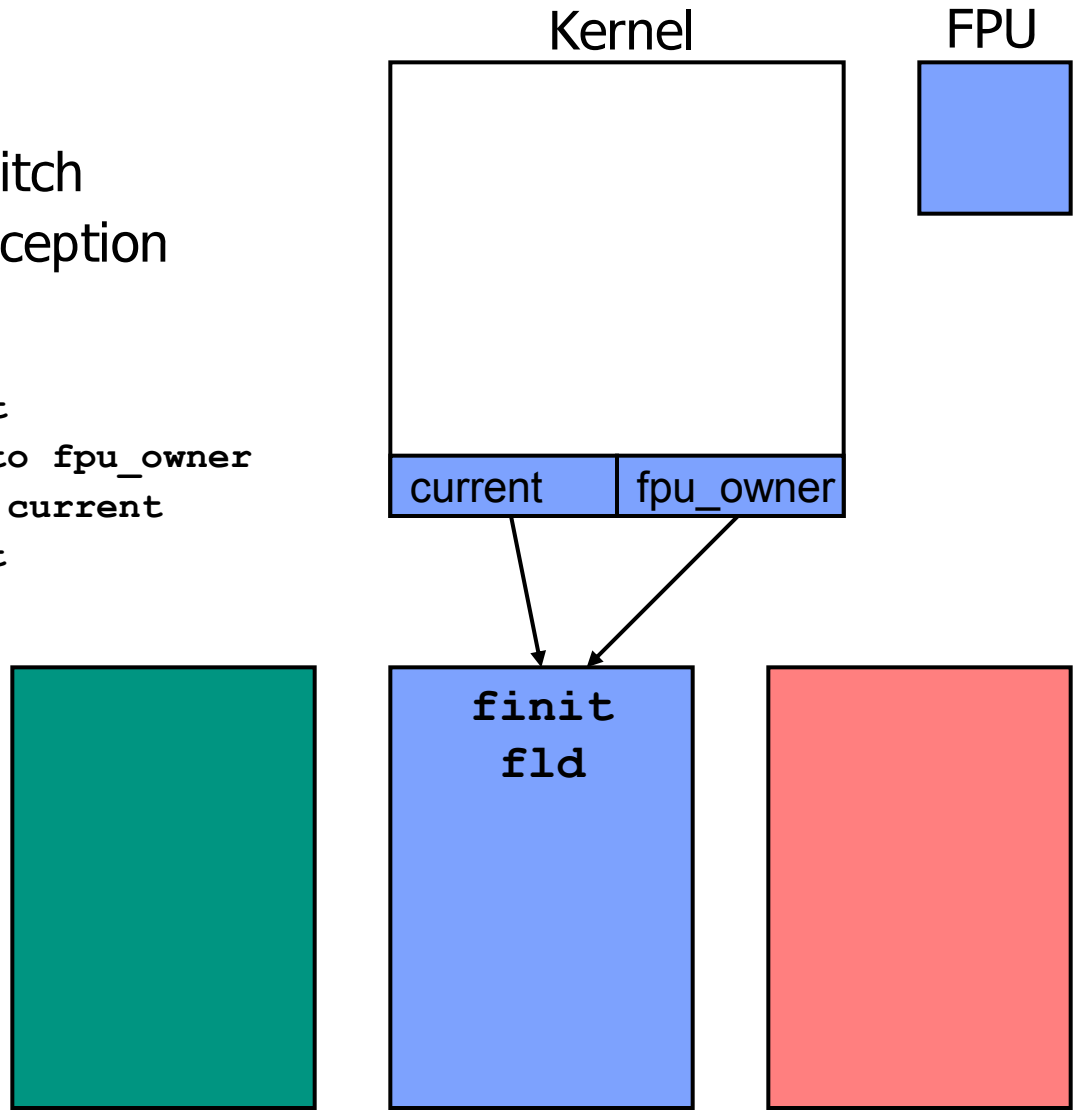
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

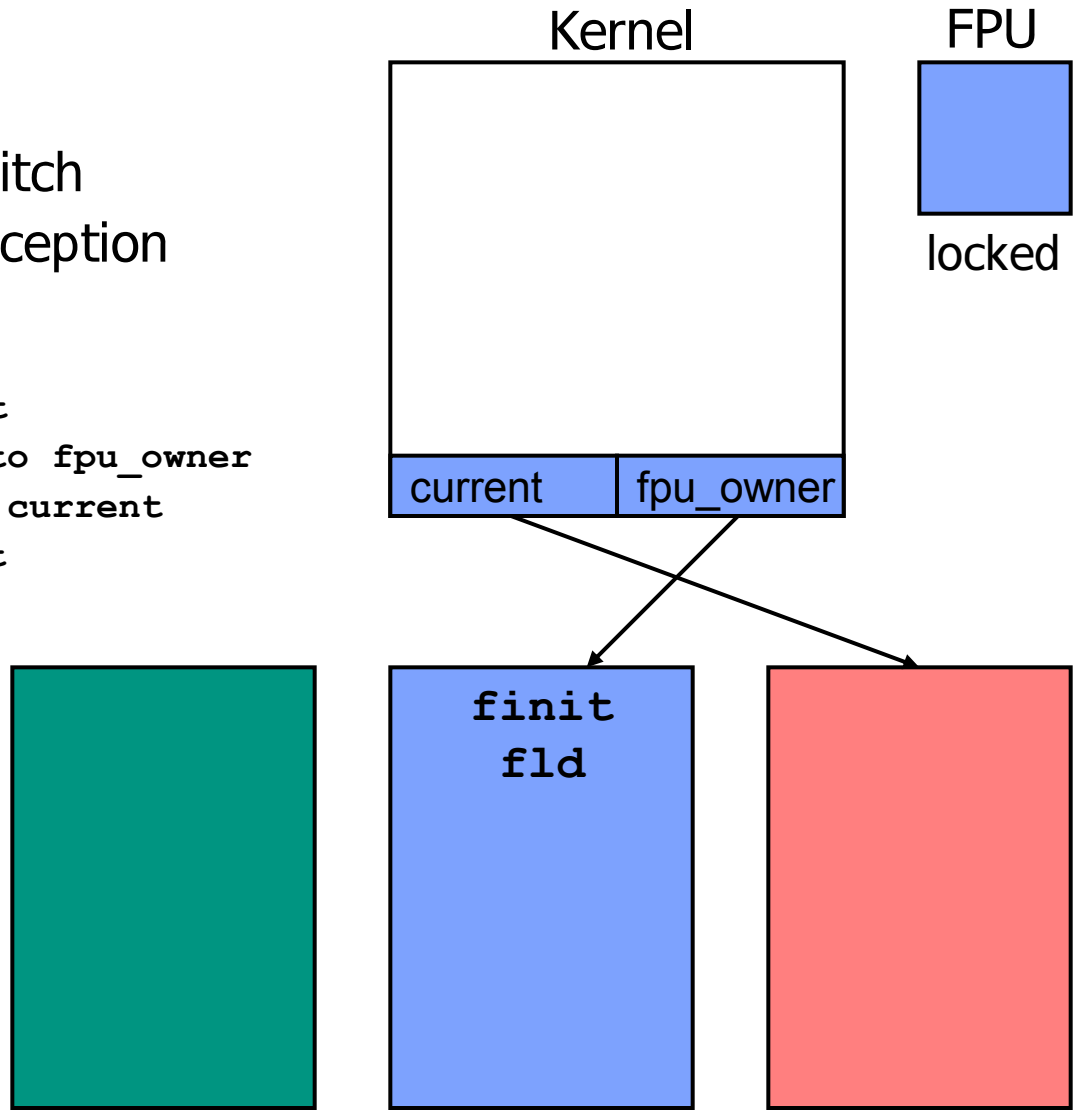
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

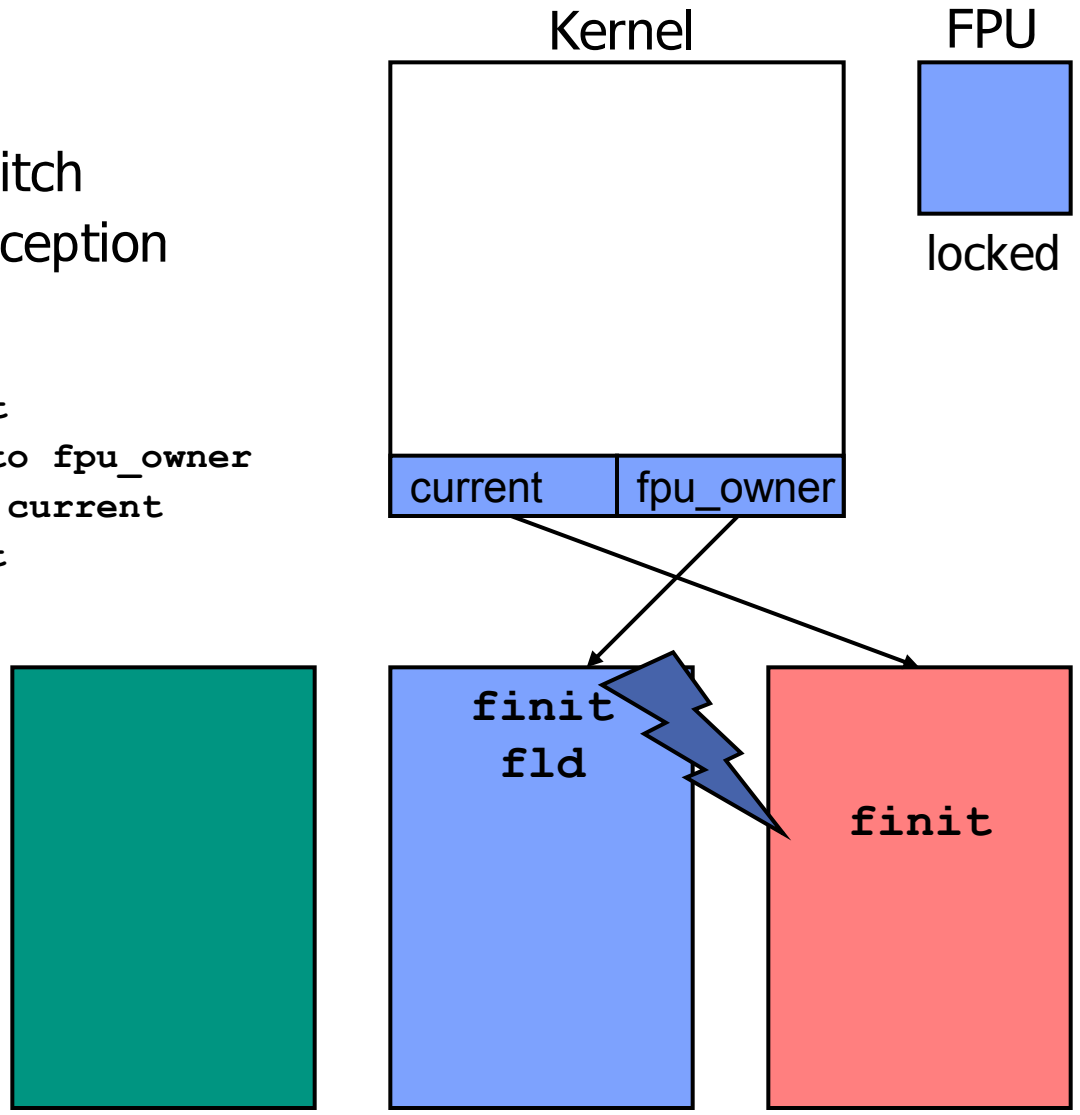
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

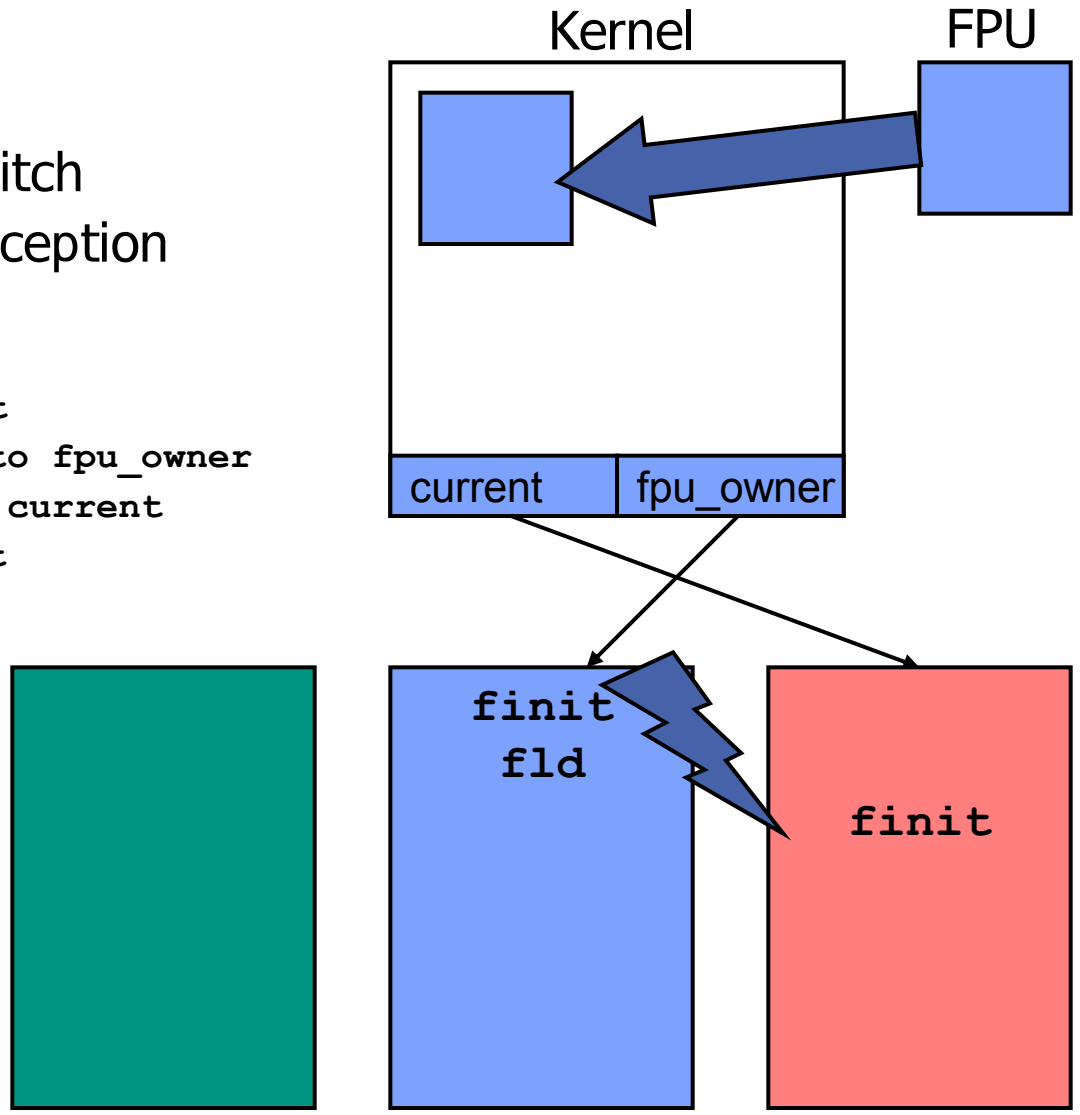
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

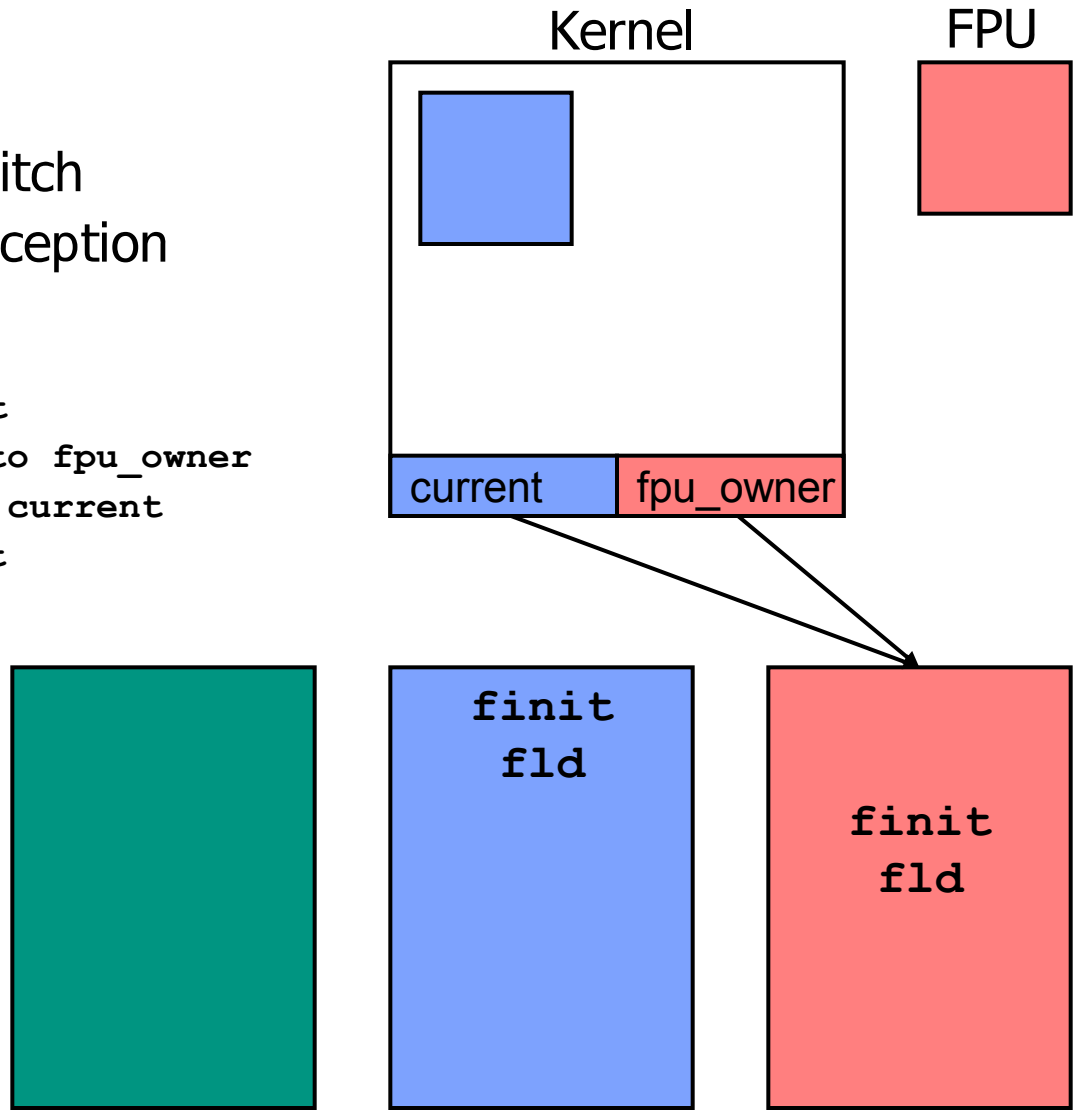
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

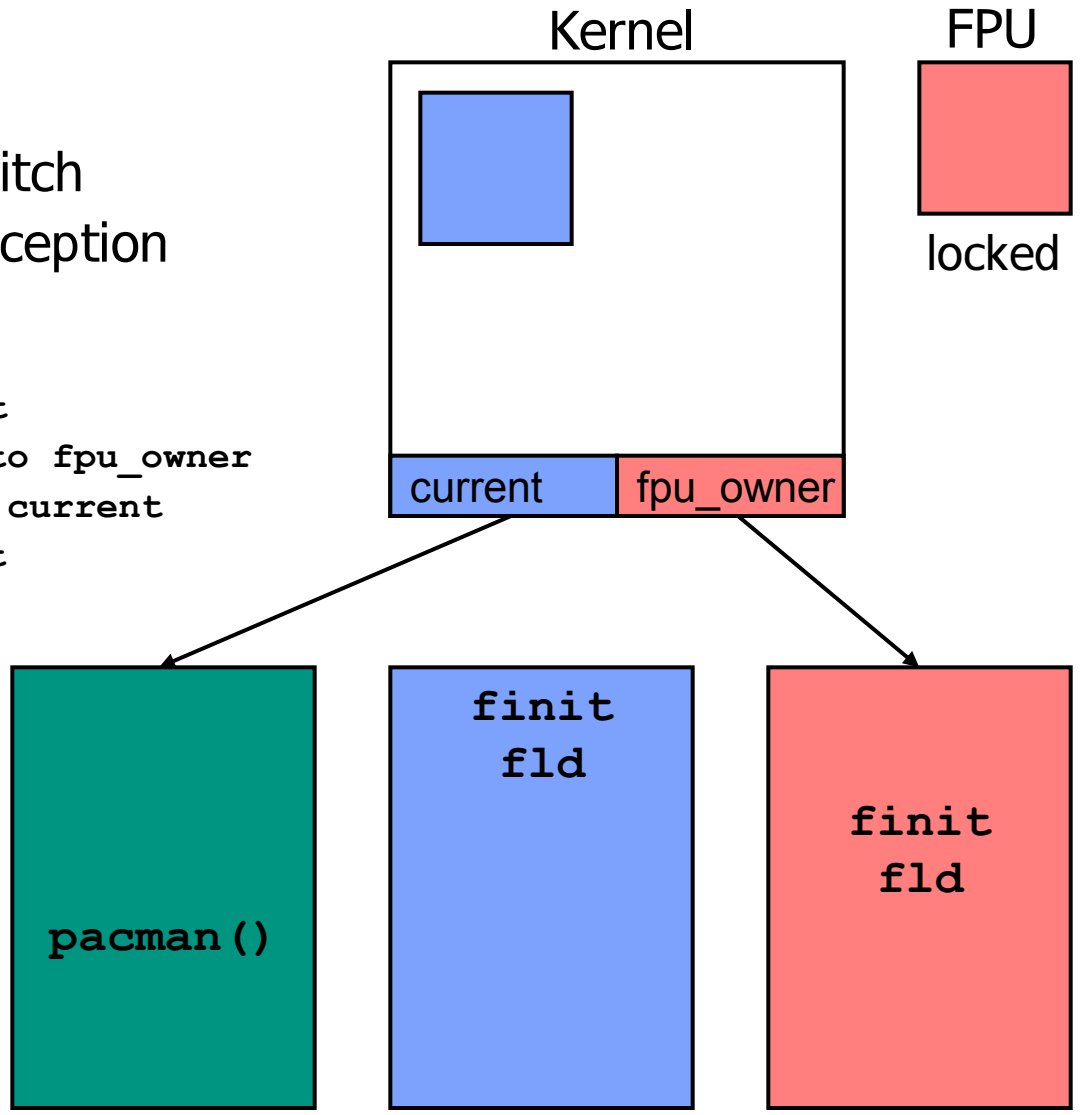
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

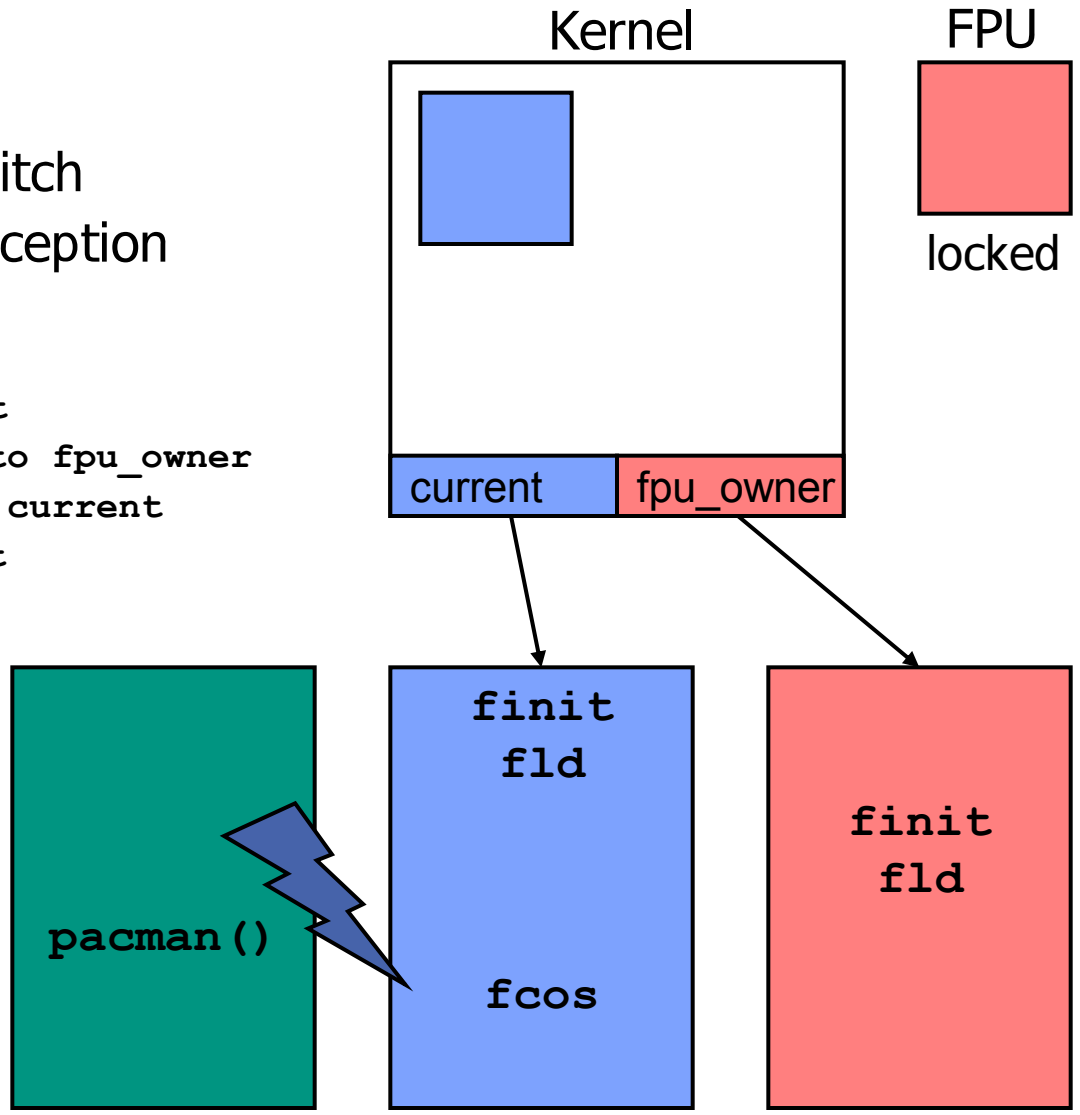
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

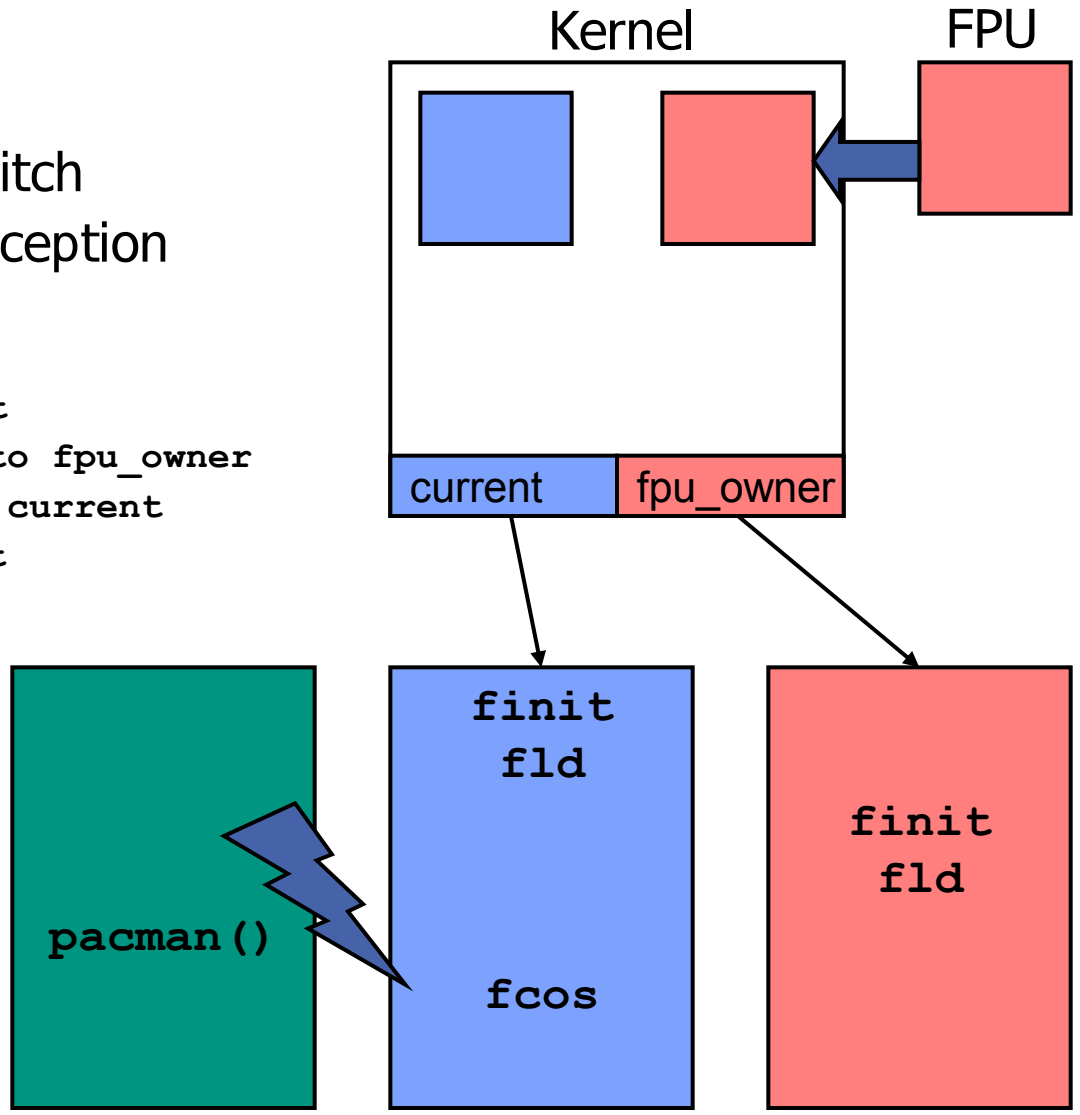
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

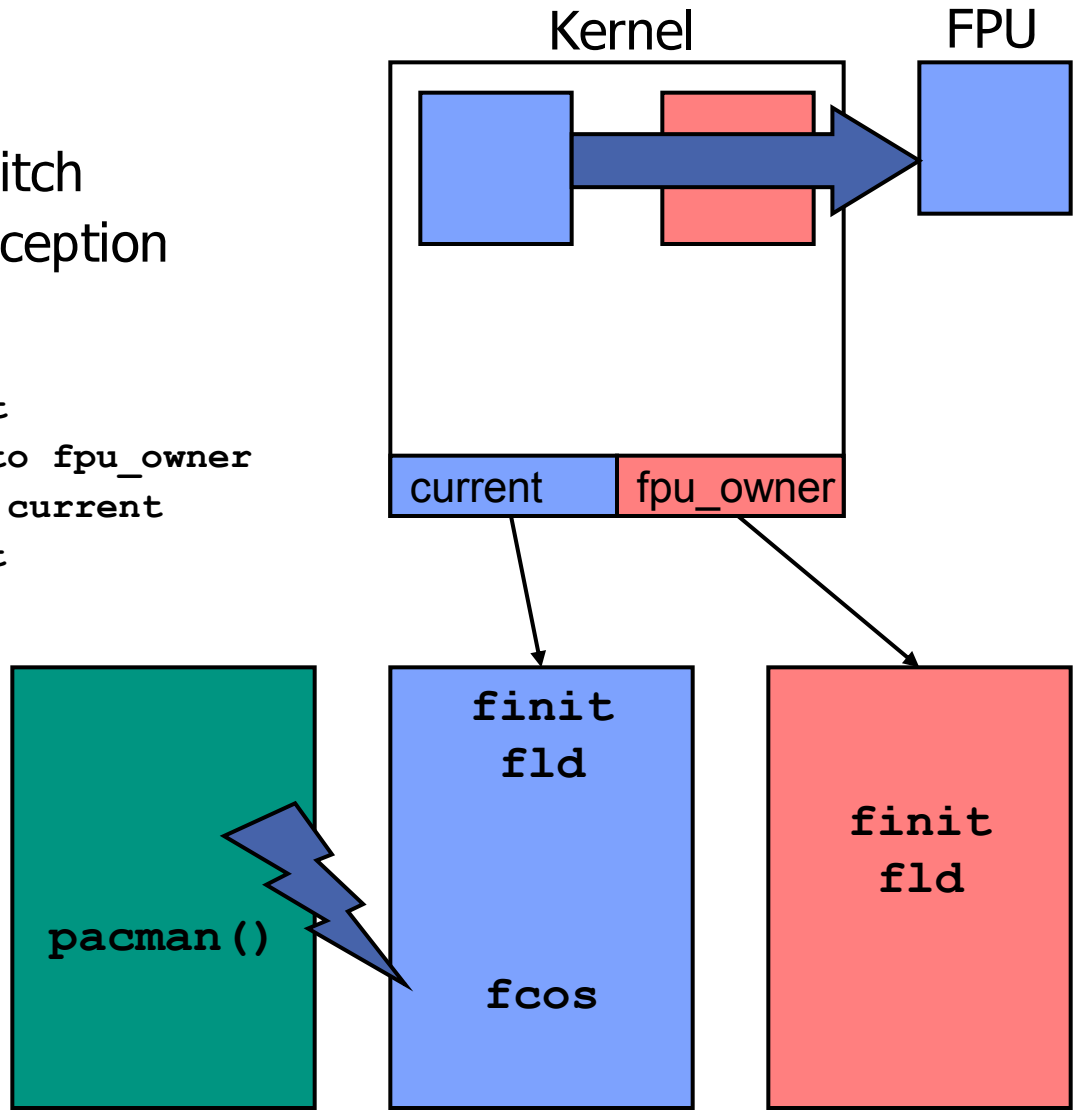
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

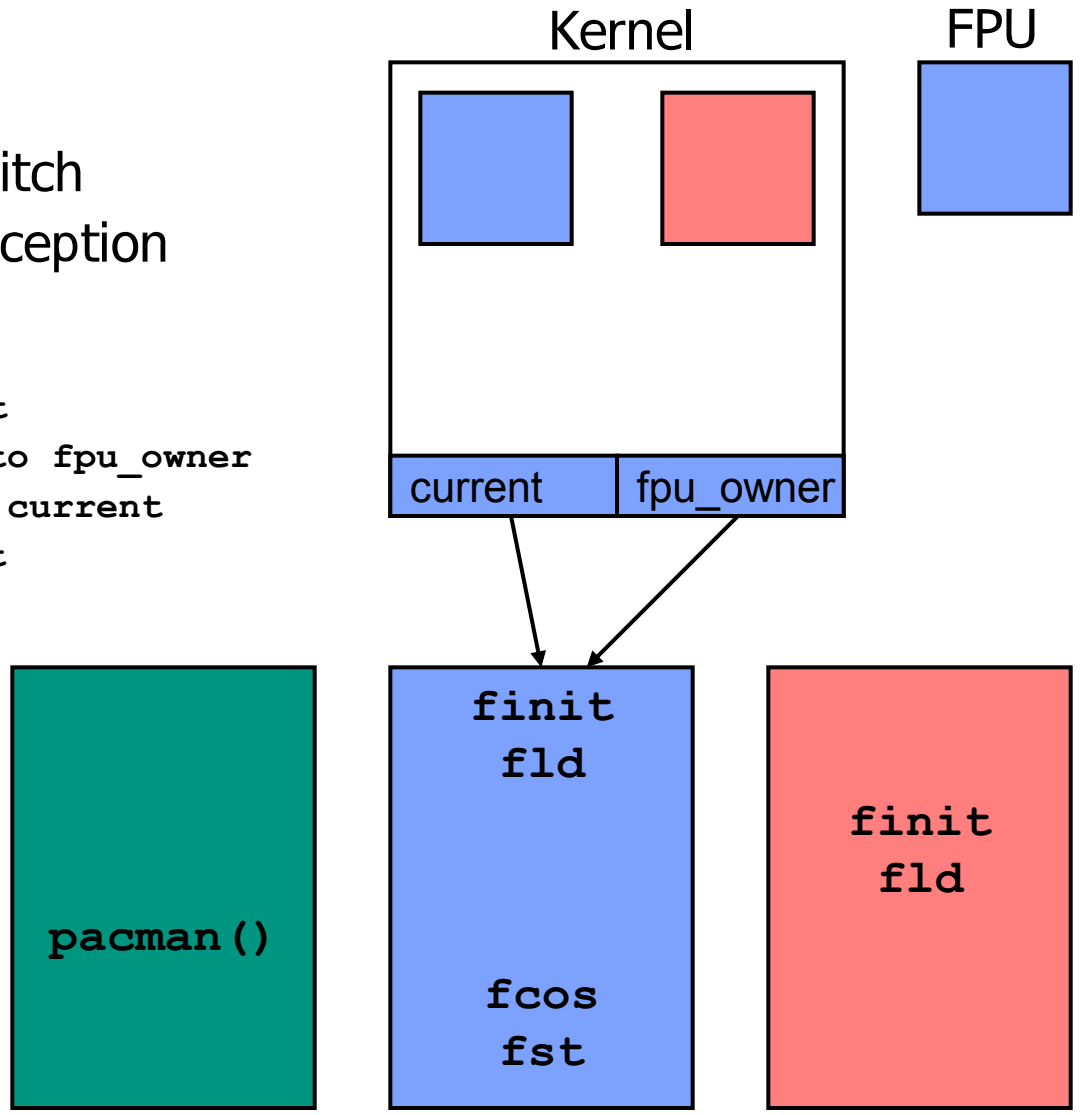
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



Lazy FPU Switching

- Lock FPU on thread switch
- Unlock at first use – exception handled by kernel

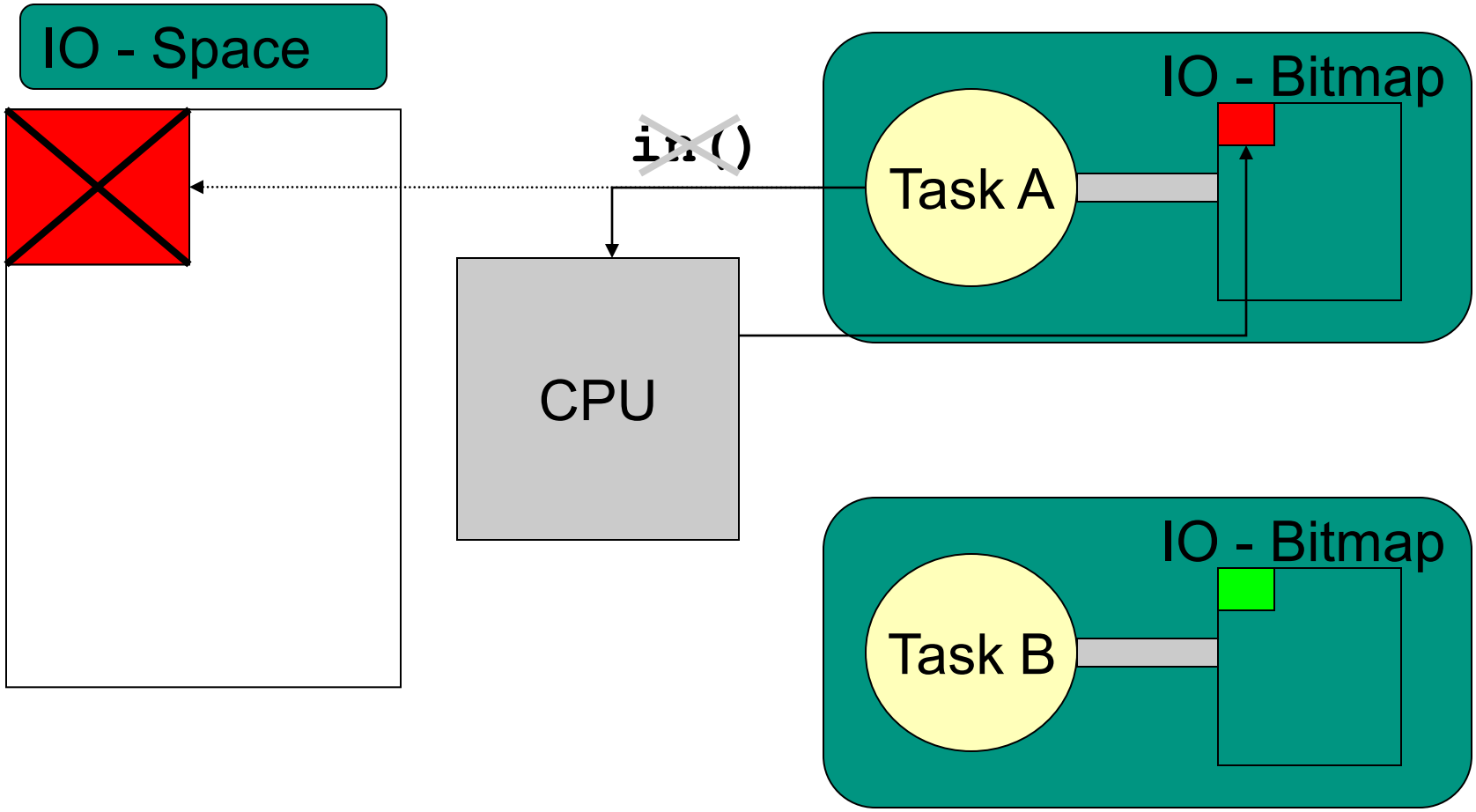
```
Unlock FPU
If fpu_owner != current
   Save current state to fpu_owner
   Load new state from current
   fpu_owner := current
```



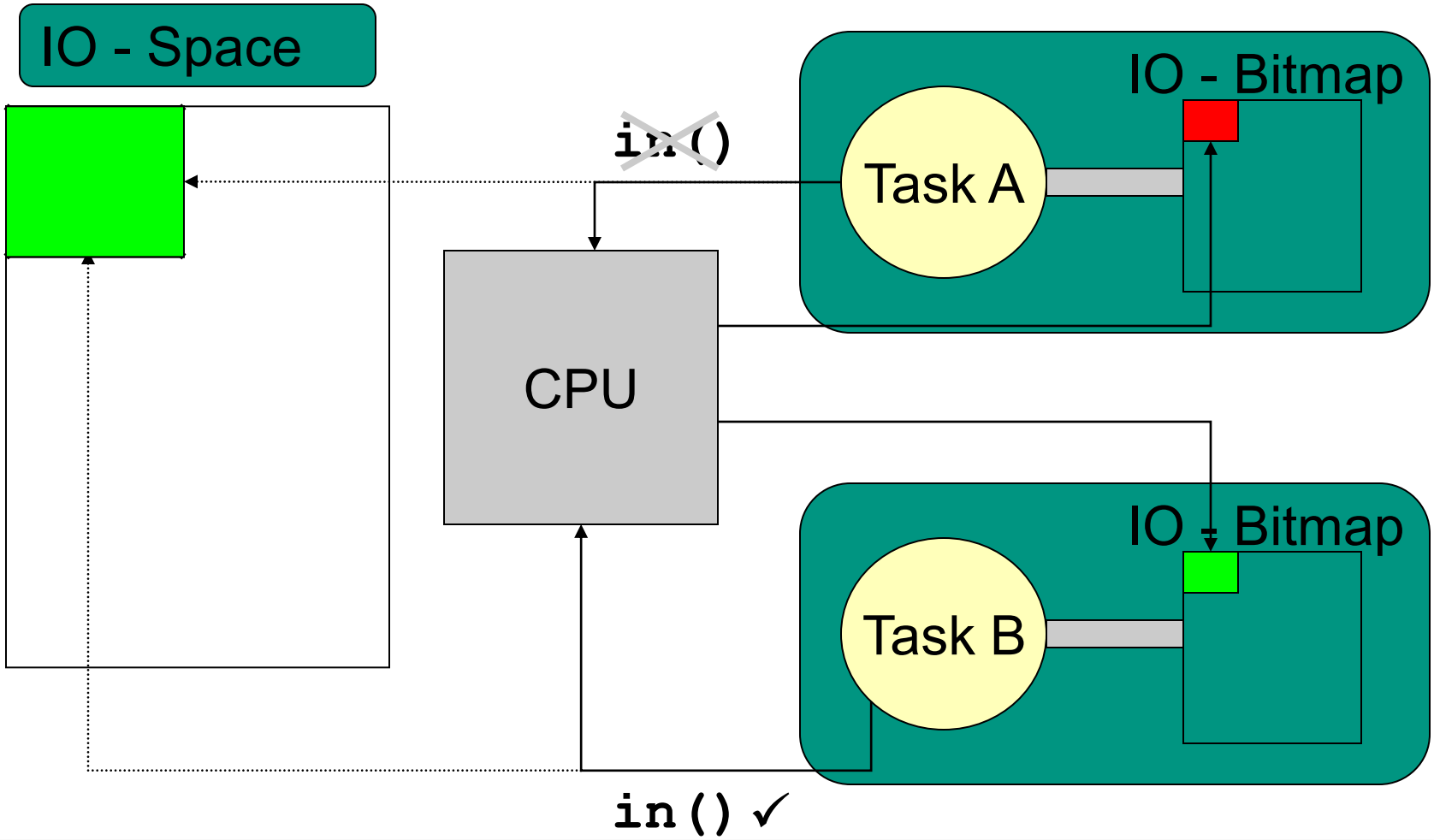
Privileged IA-32 Instructions

- Privileged instructions
  - `lidt` – Load interrupt descriptor table
  - `rdmsr`, `wrmsr` – Access model-specific registers
  - `wbinvd` – Write back and invalidate caches
  - `lgdt`, `lldt`, `ltr`, …
- IOPL-sensitive
  - `cli`/`sti` – Disable/enable interrupts
  - `in`, `out`, `ins`, `outs` – Access I/O address space

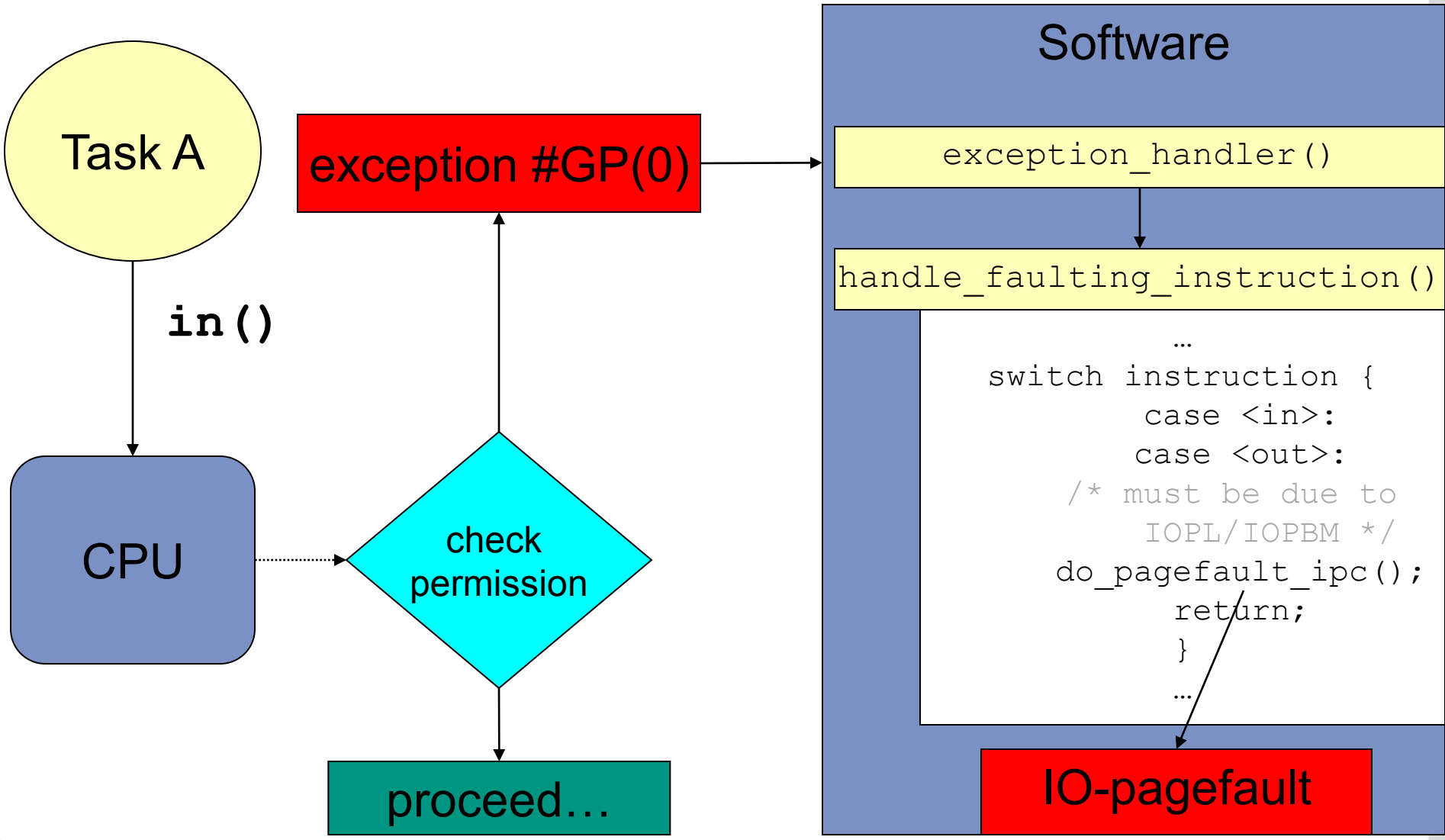
I/O Permission Bitmap



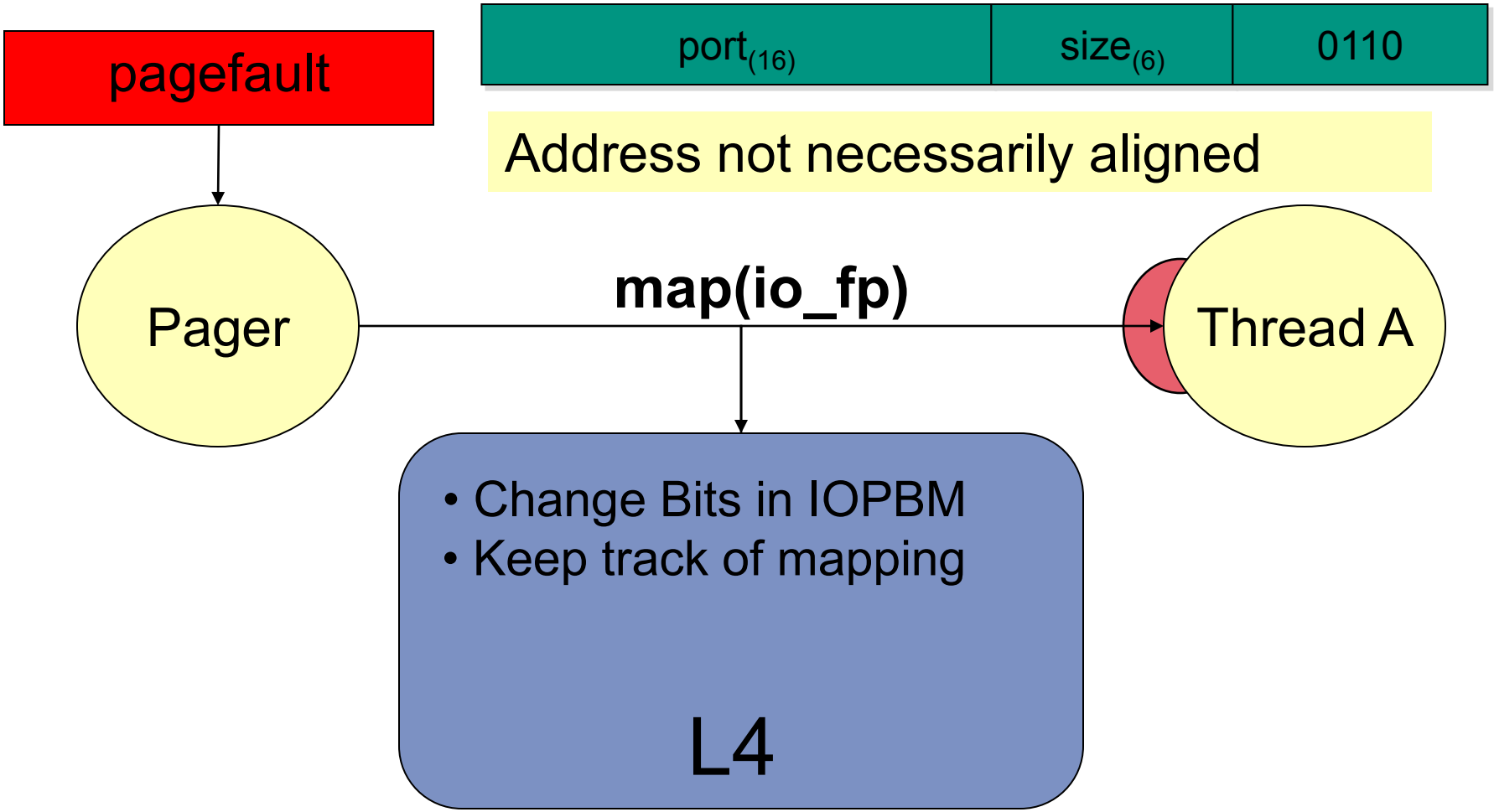
I/O Permission Bitmap



L4 I/O Port Access Control – IO Faults



L4 I/O Port Access Control – IO Mapping



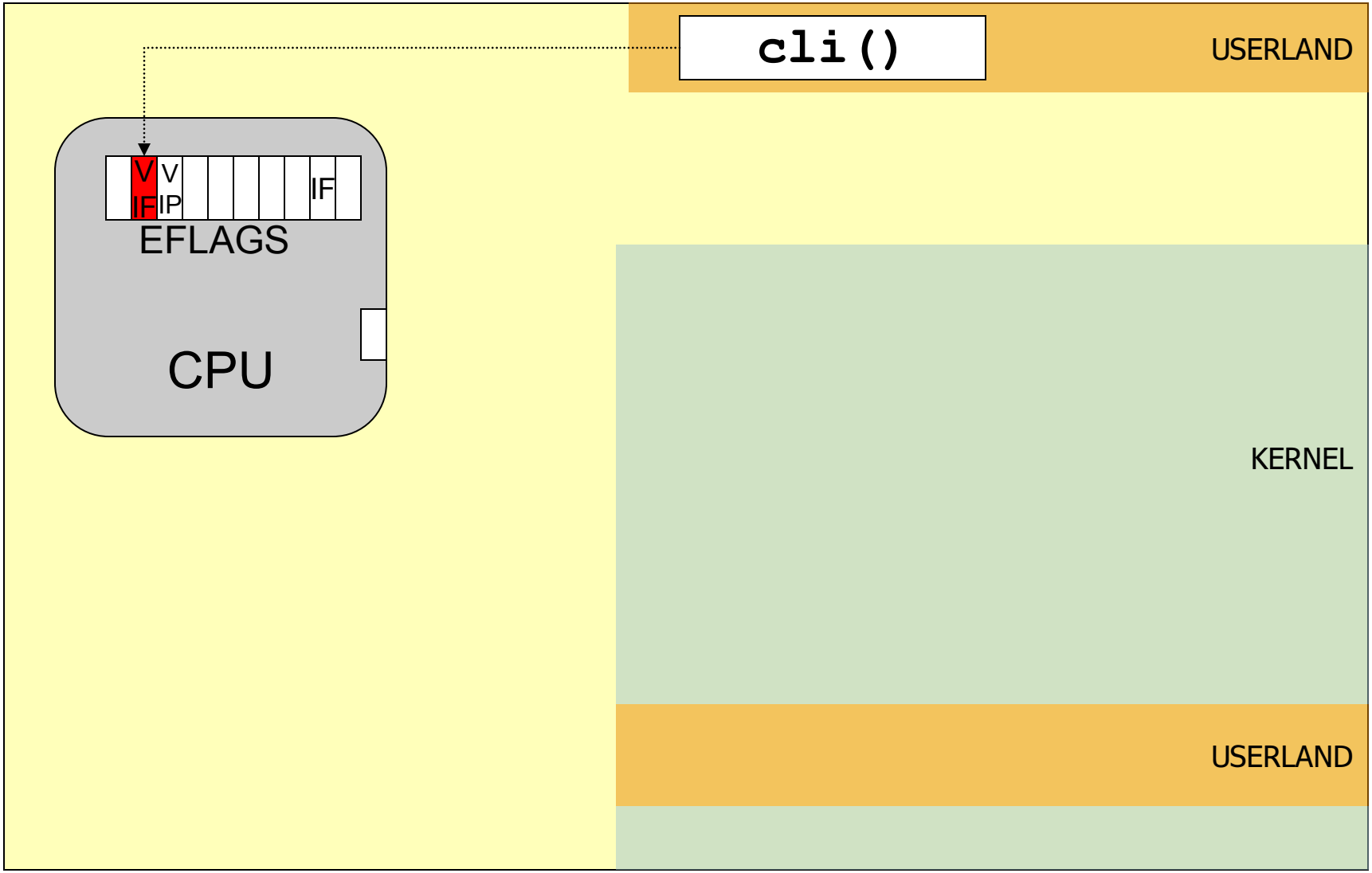
Virtualizing the Interrupt Flag

- Interrupt enable flag in EFLAGS
  - But user mode cannot modify IF directly
- `cli`/`sti` cause exception (#GP)
  - Analyze faulting instruction
  - Flip user's IF
  - Per-thread IF
- But … expensive
  - Unusable for implementing critical sections

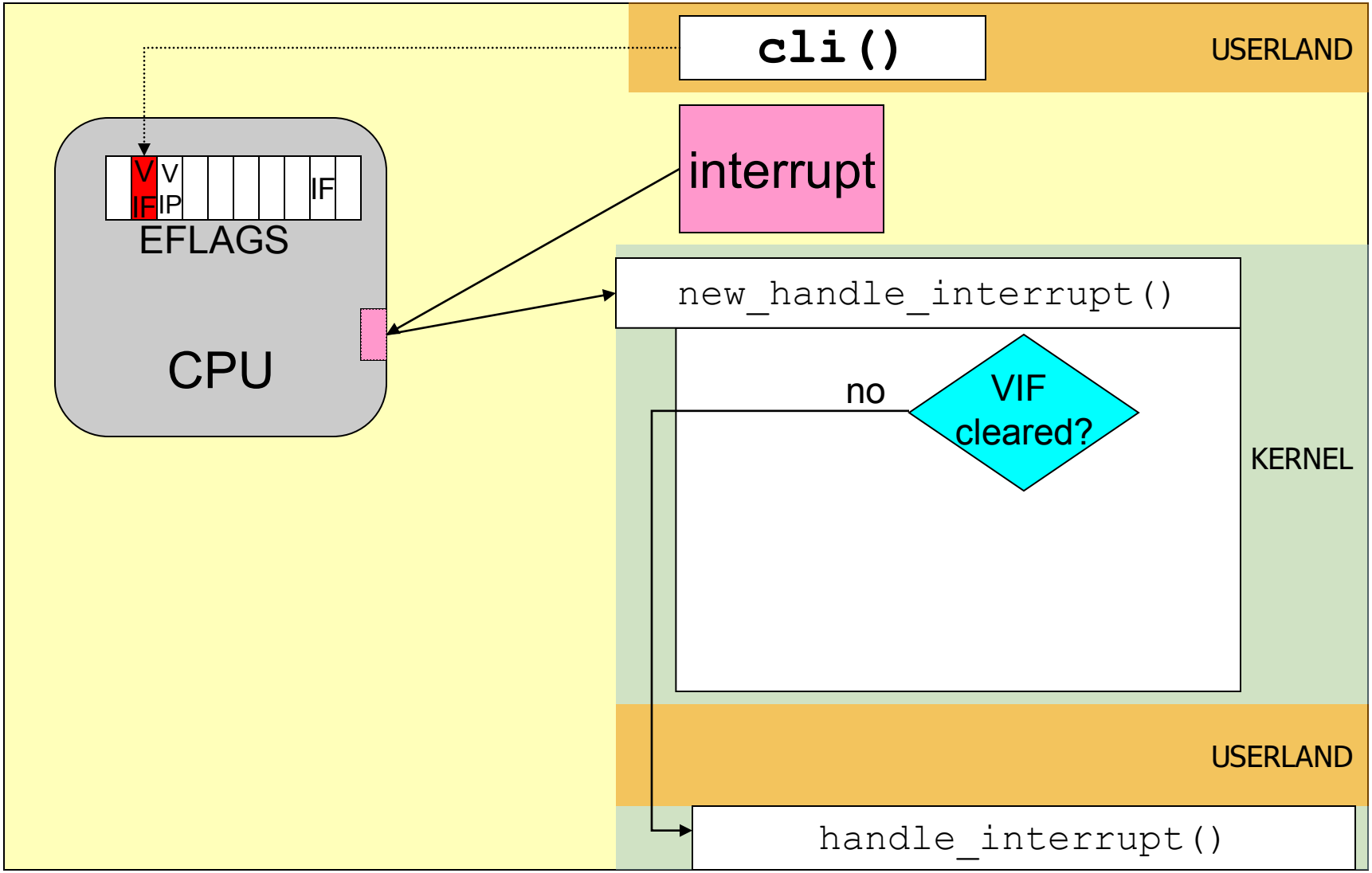
Protected Mode Virtual Interrupts

- Hardware support
  - Allows enforcing maximum interrupt latency
  - Two new flags in EFLAGS register (VIF, VIP)
- **`cli`**/**`sti`** in user mode updates Virtual IF
  - Less costly – no exception
- Hardware interrupts still subject to real IF
  - Deliver interrupts immediately or
  - Postpone delivery
- Kernel can set VIP flag
  - Indicates pending interrupt
  - Next **`sti`** will cause #GP
  - Kernel can deliver pending interrupts

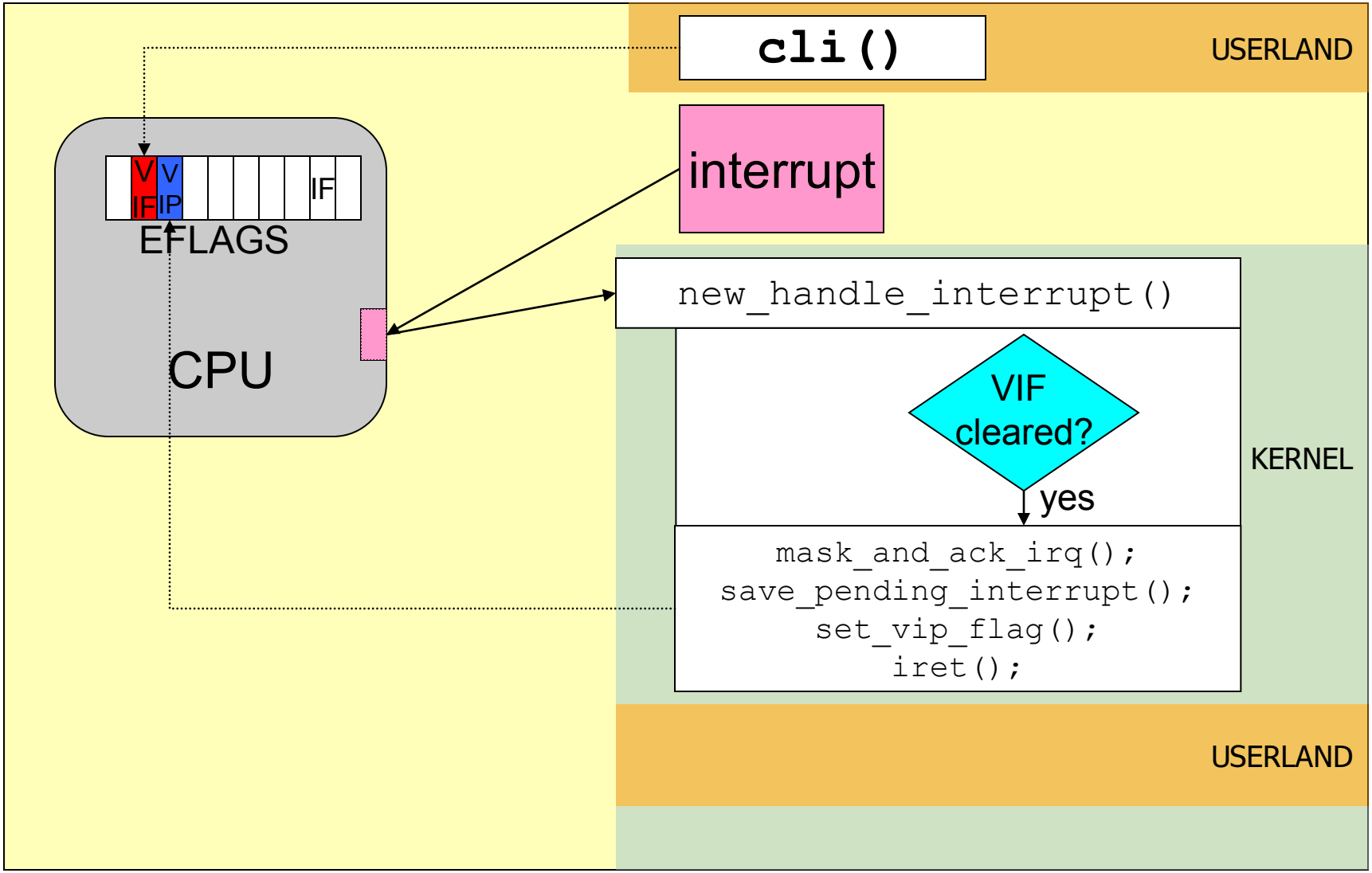
Protected Mode Virtual Interrupts



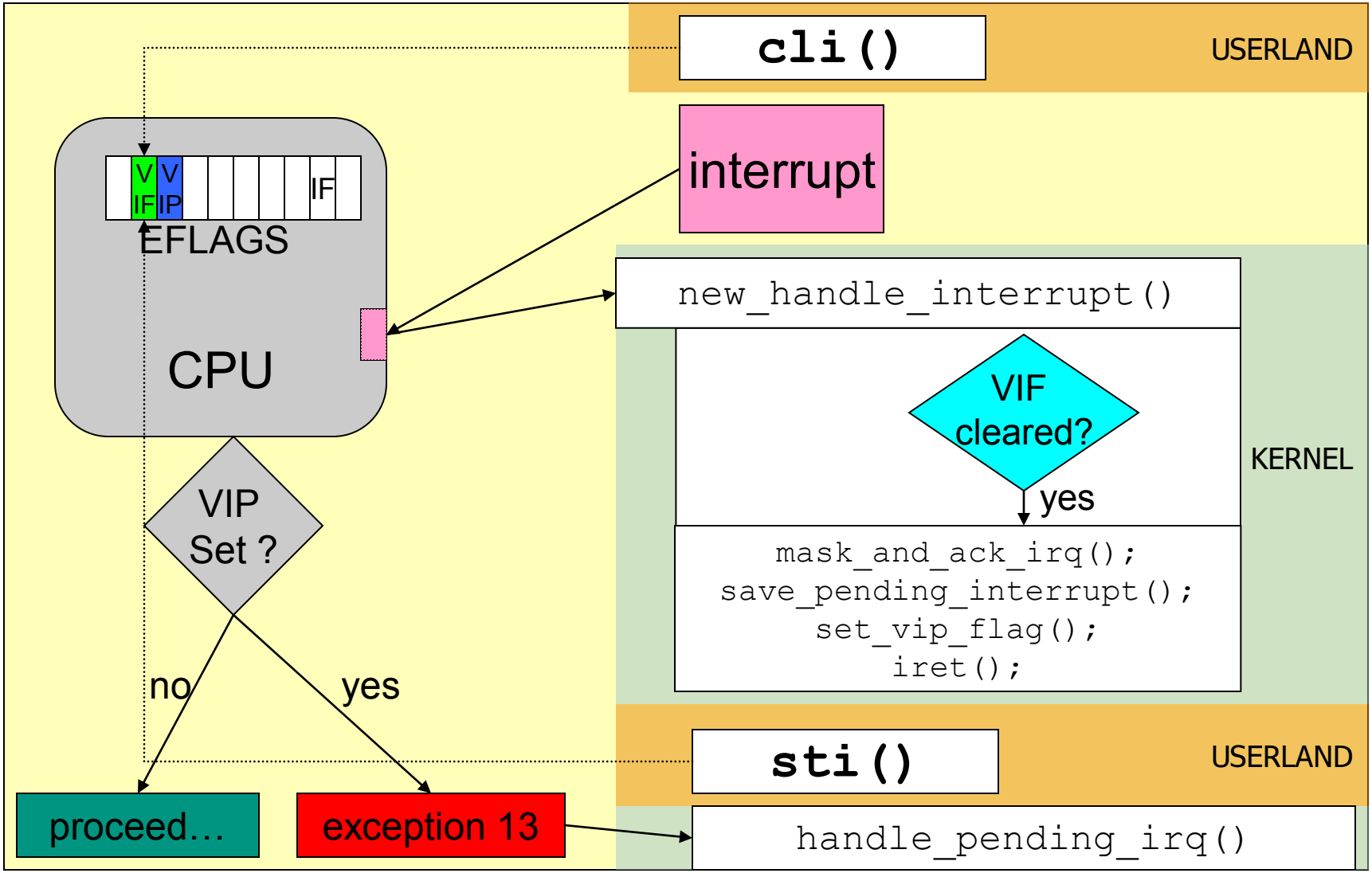
Protected Mode Virtual Interrupts



Protected Mode Virtual Interrupts



Protected Mode Virtual Interrupts



Summary

- Most events processed in userspace
  - Page faults, interrupts, exceptions
  - Handler sends and receives IPC
  - IPC invisible to faulting thread
- Some events handled by the kernel internally
  - Kernel page faults
  - Timer interrupt, IPI
  - FPU/Interrupt flag virtualization