Operating System Design and Implementation

OSDI overview

Charles Tsao

Outline

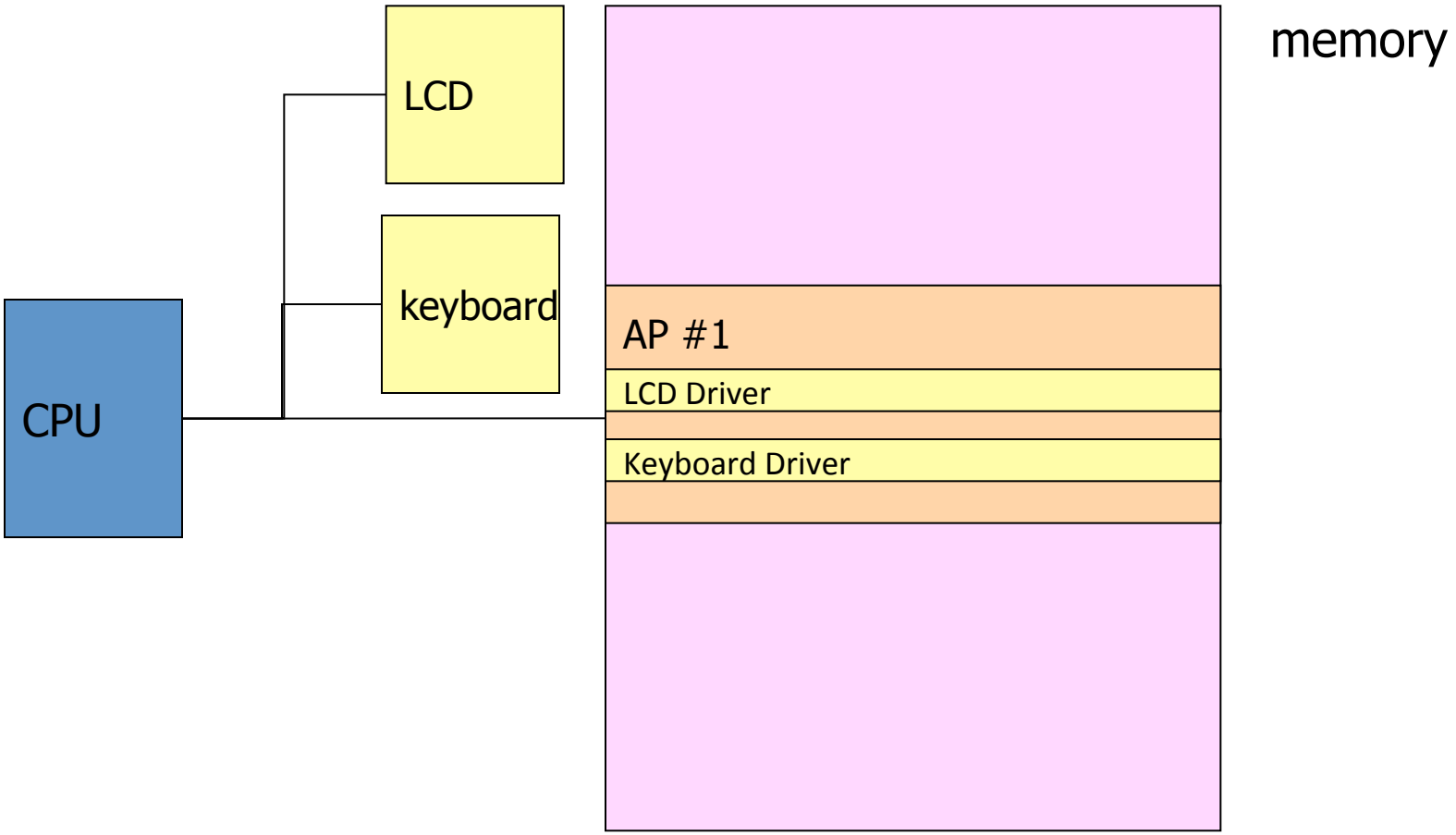
- OS Basics
- Process Management
- Kernel Synchronization
- Memory Management
- I/O Management

What is OS?

- Special computer program
- A middleware between application software and hardware
  - What kind of hardware you want to drive
  - What kind of services you have to provide
- Assistant applications to manage physical hardware more efficiently, …

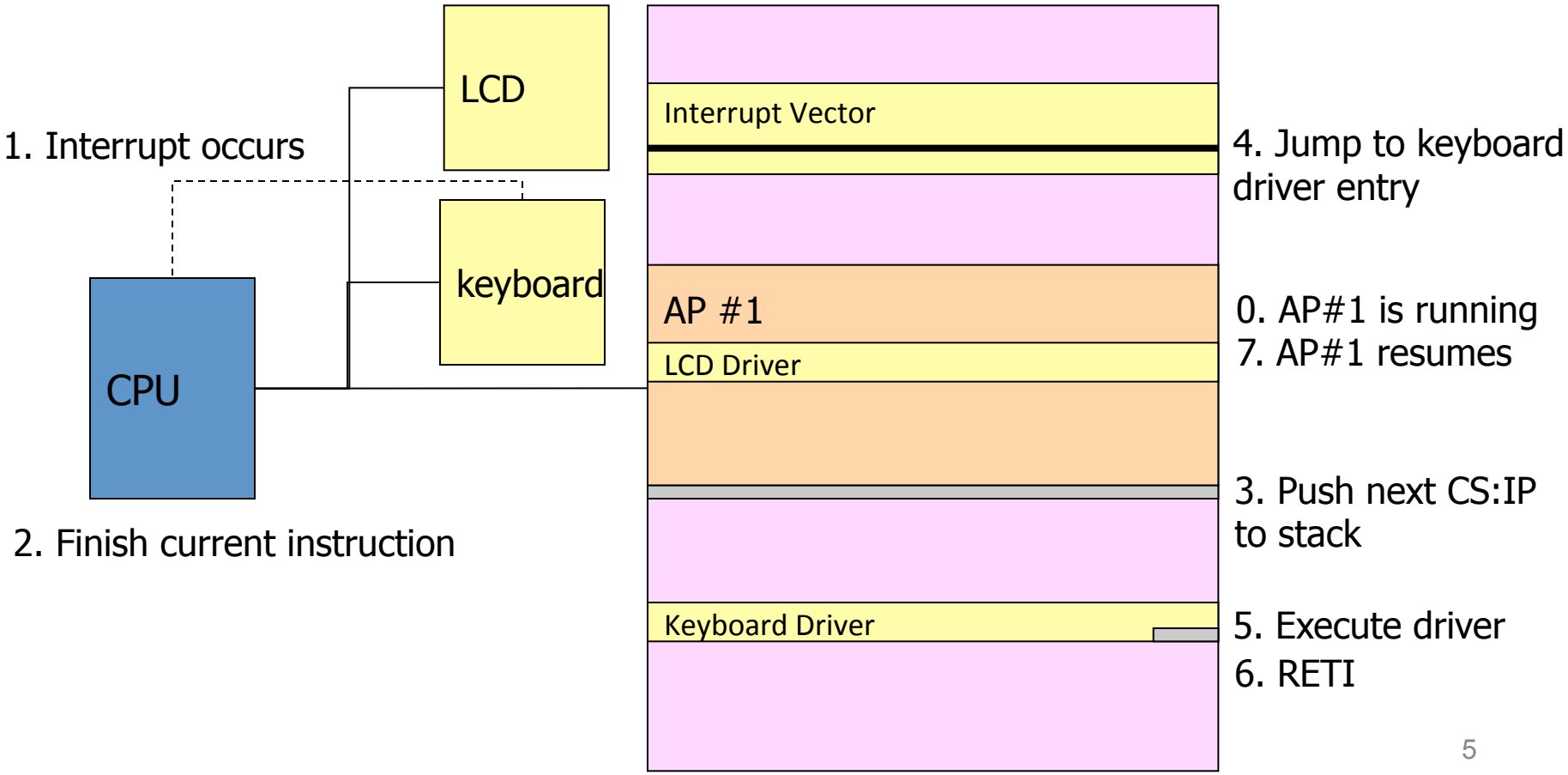
Let's build an OS together

- Step 1: drive the hardware



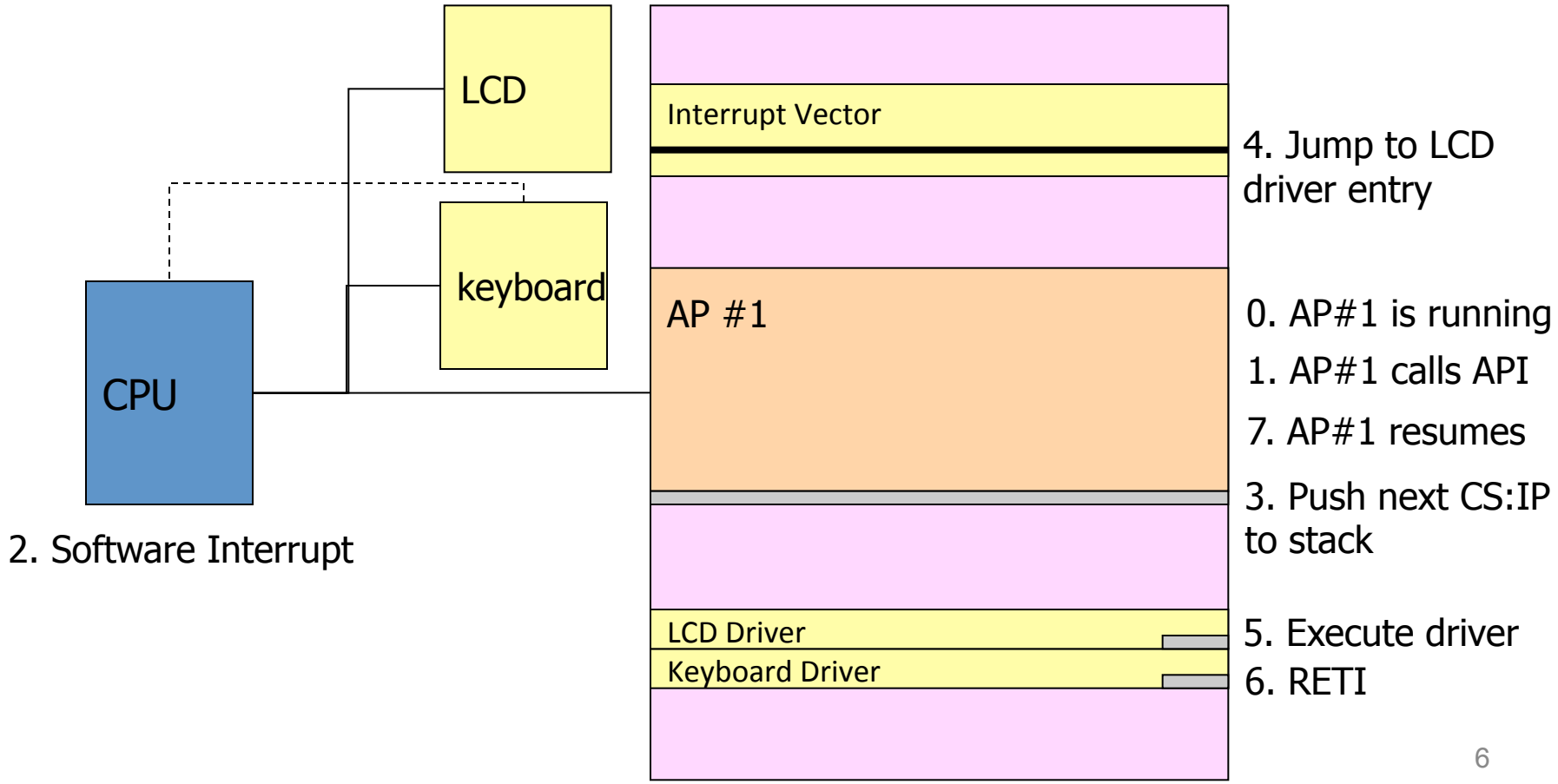
Let's build an OS together

- Step 2: handle interrupt



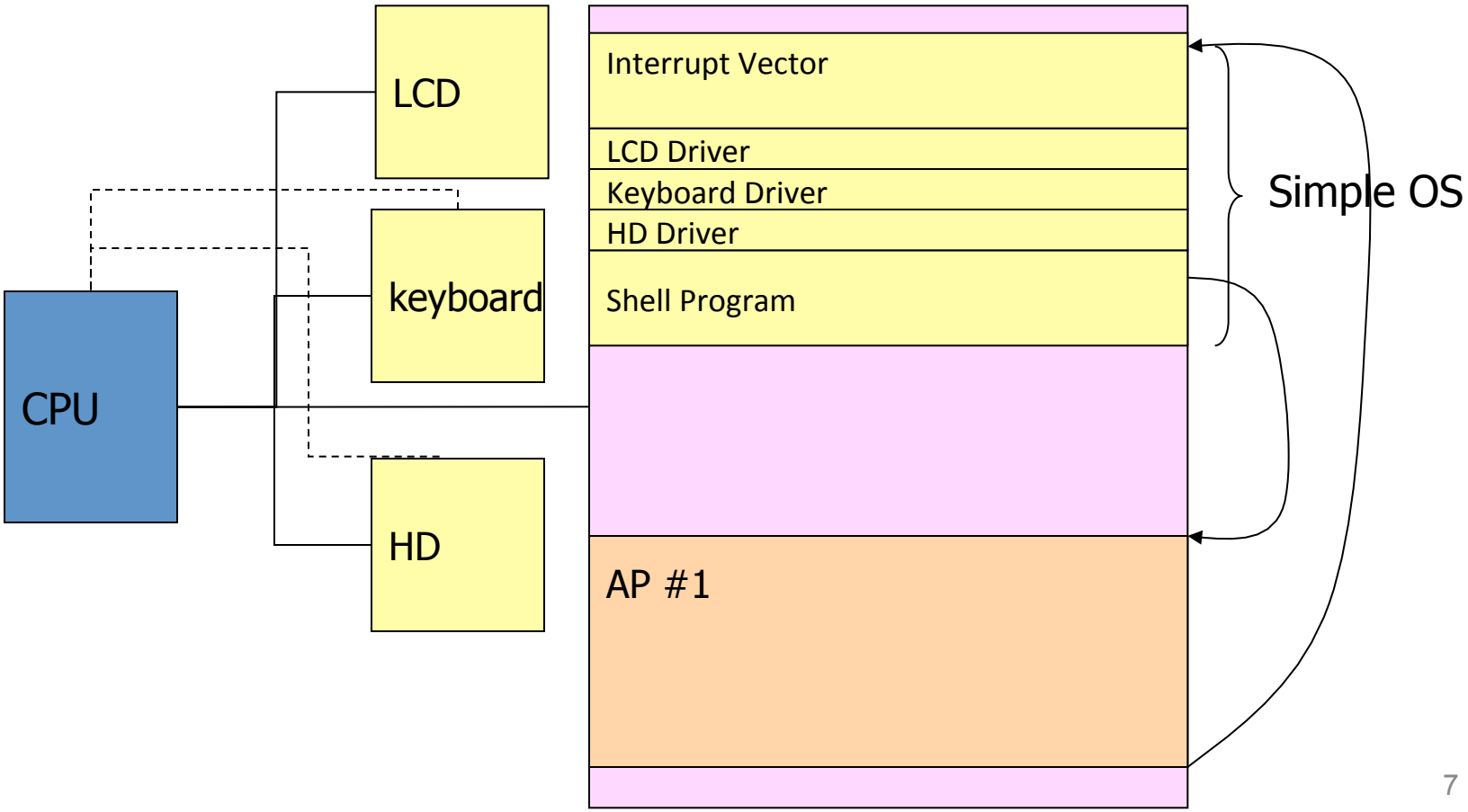
Let's build an OS together

- Step 3: handle software interrupt



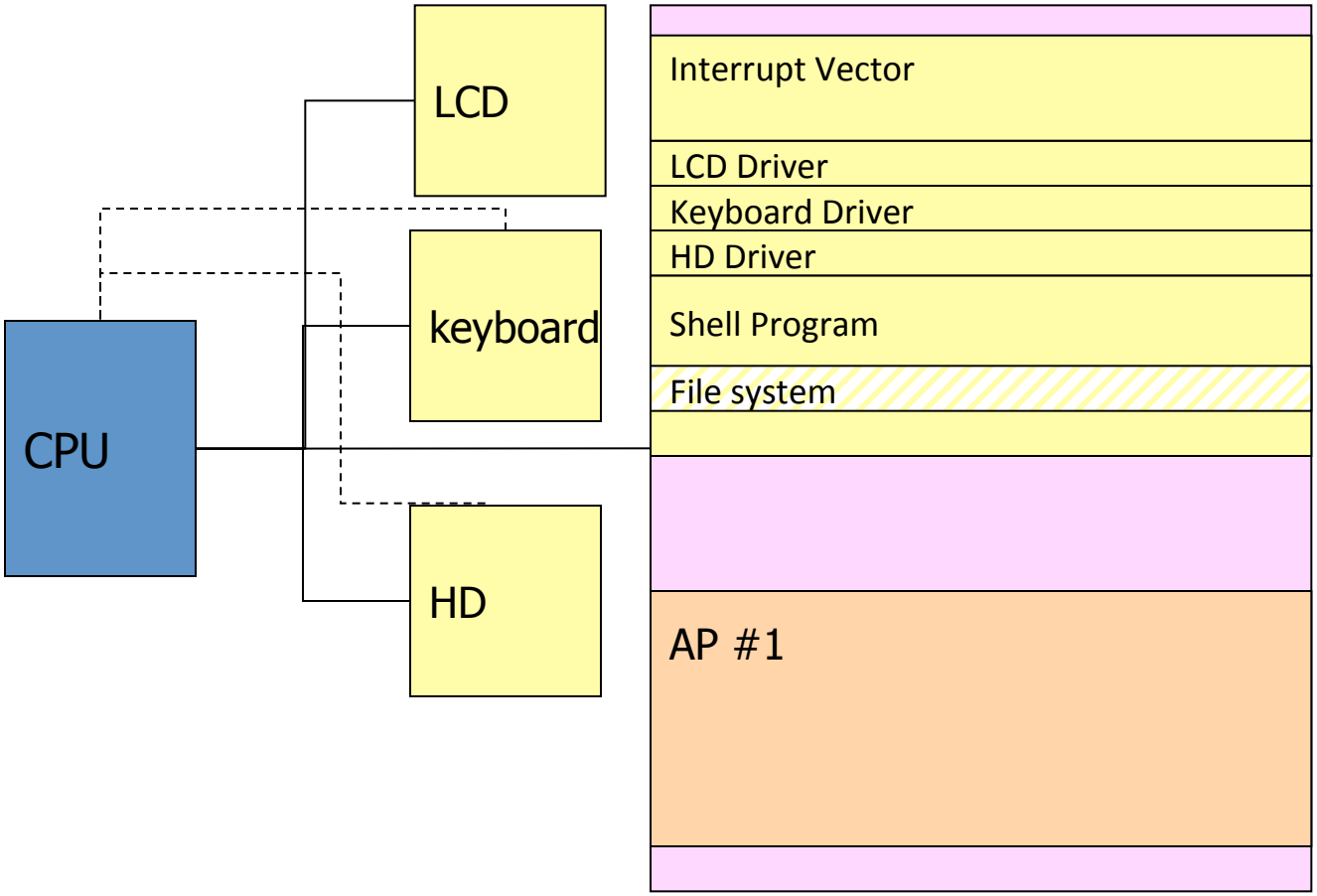
Let’s build an OS together

- Step 4: shell



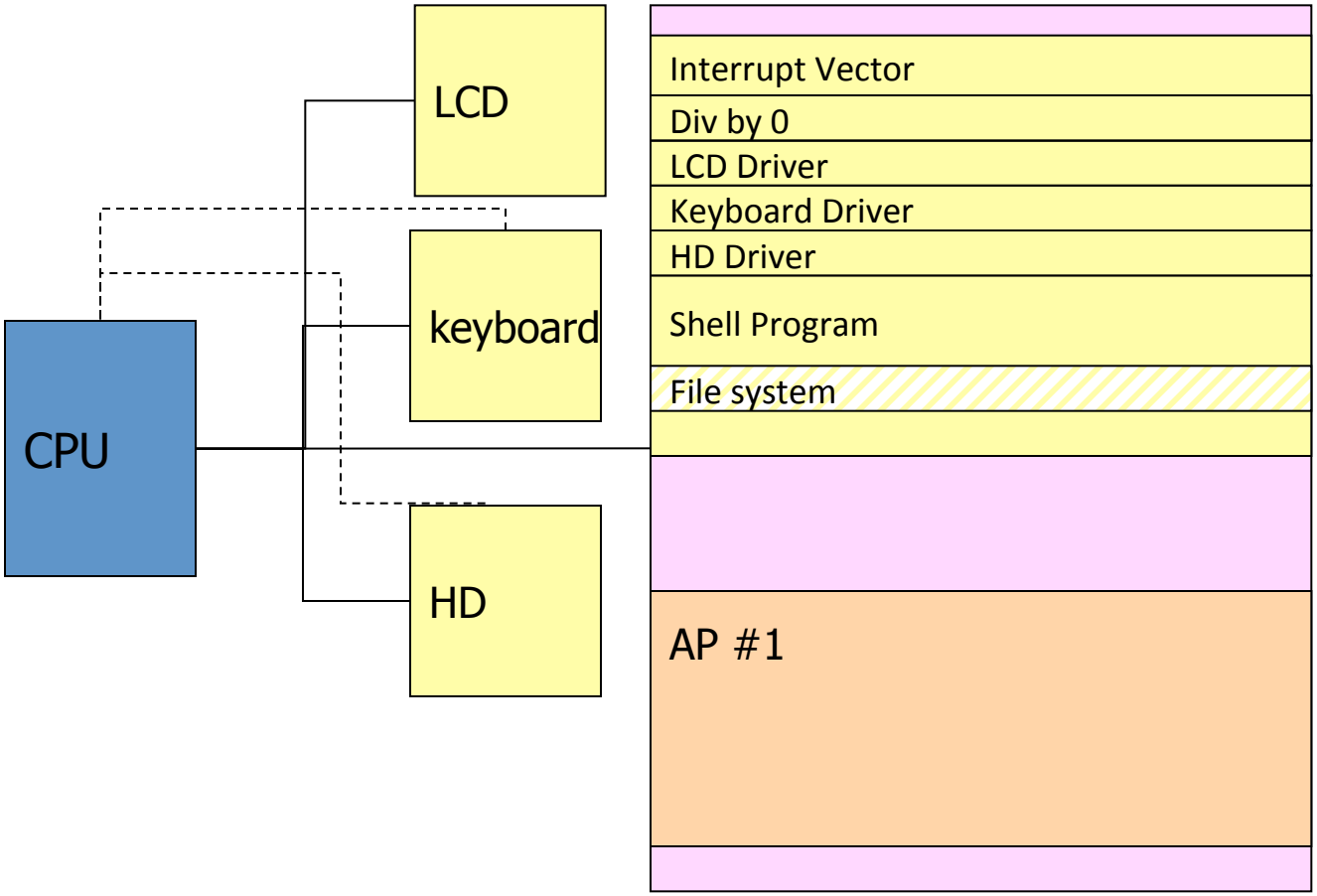
Let's build an OS together

- Step 5: file system



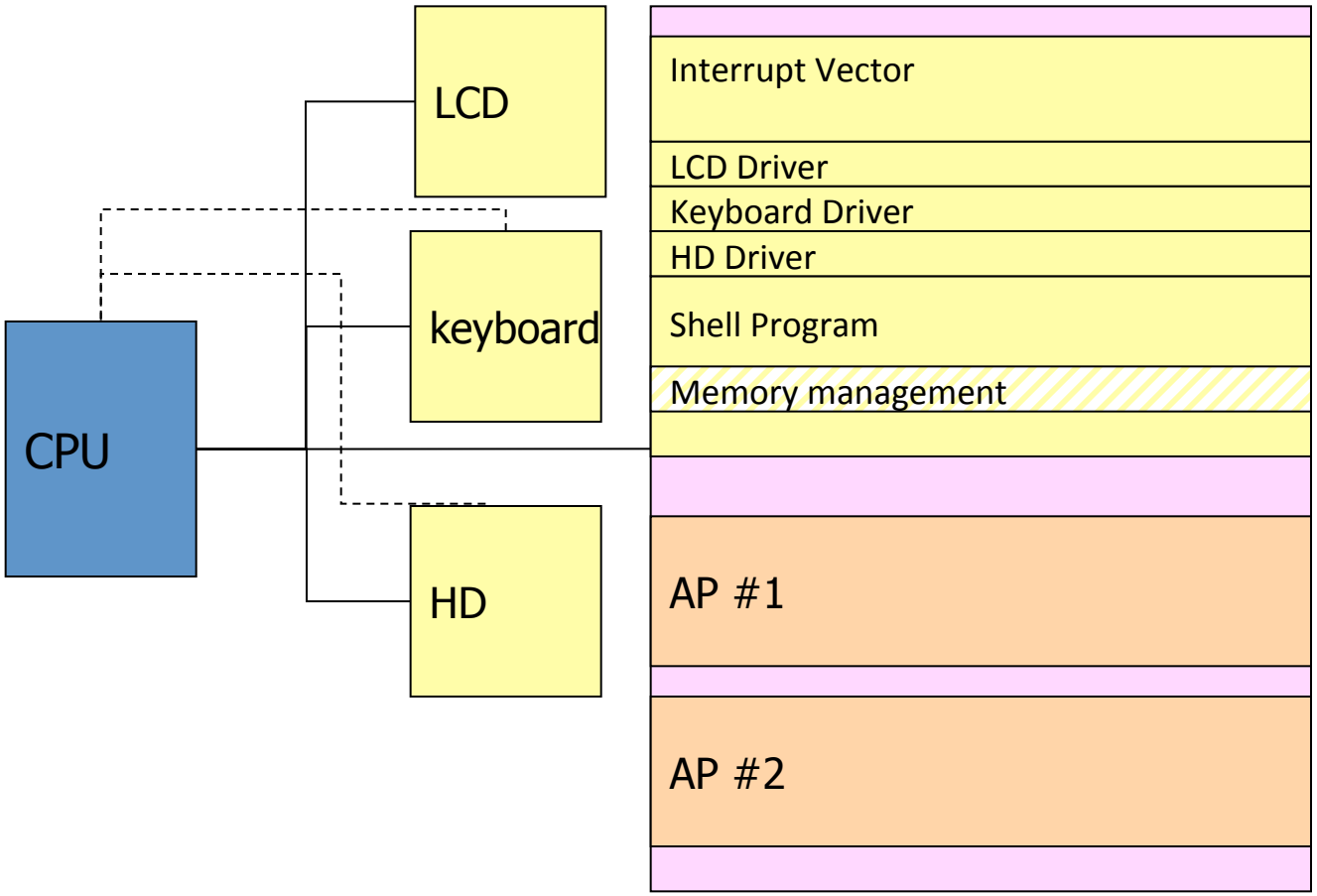
Let's build an OS together

- Step 6: handling exception



Let's build an OS together

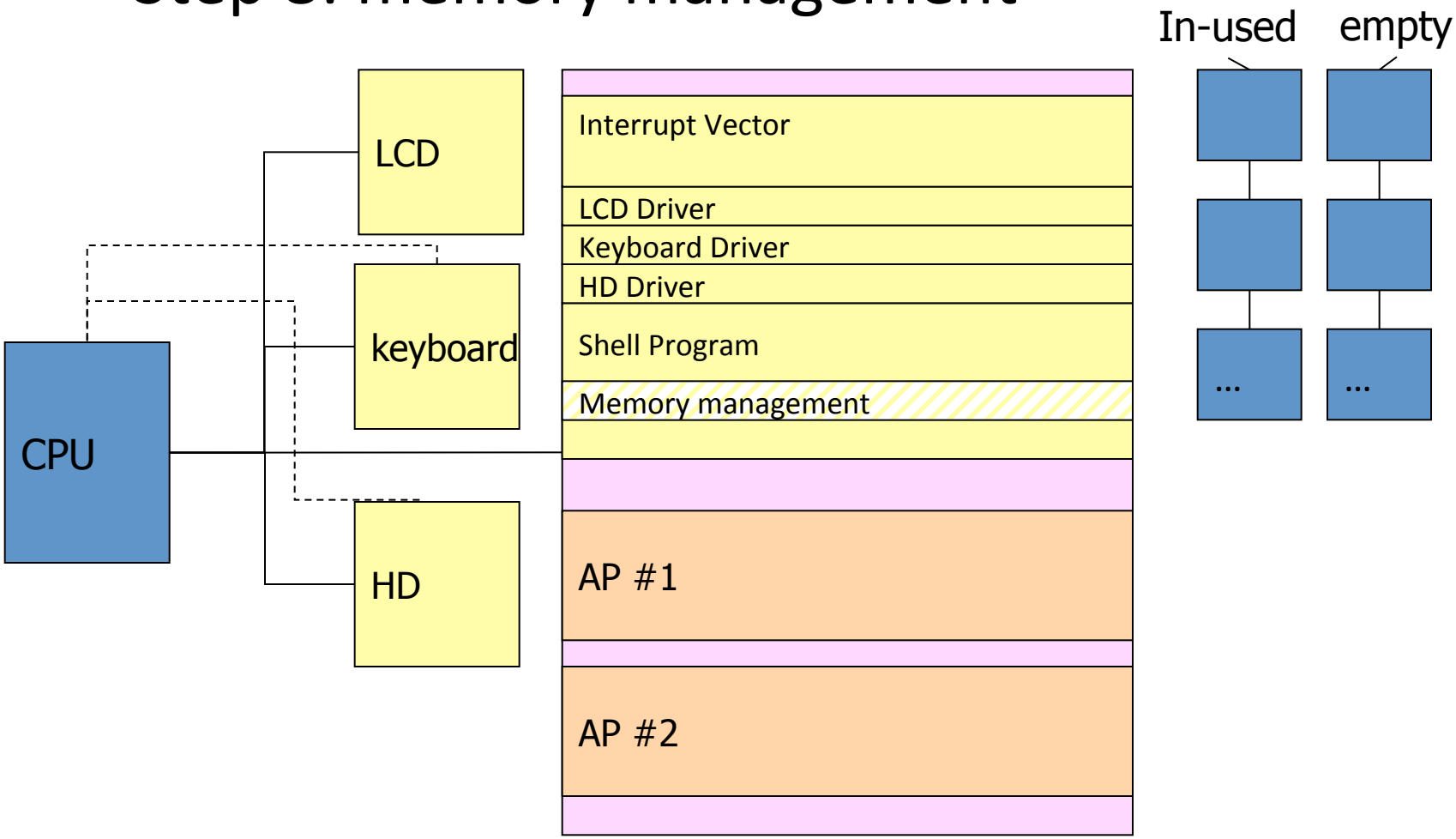
- Step 7: multi-programming



- Memory management?
- Protection?
- Current CS:IP?

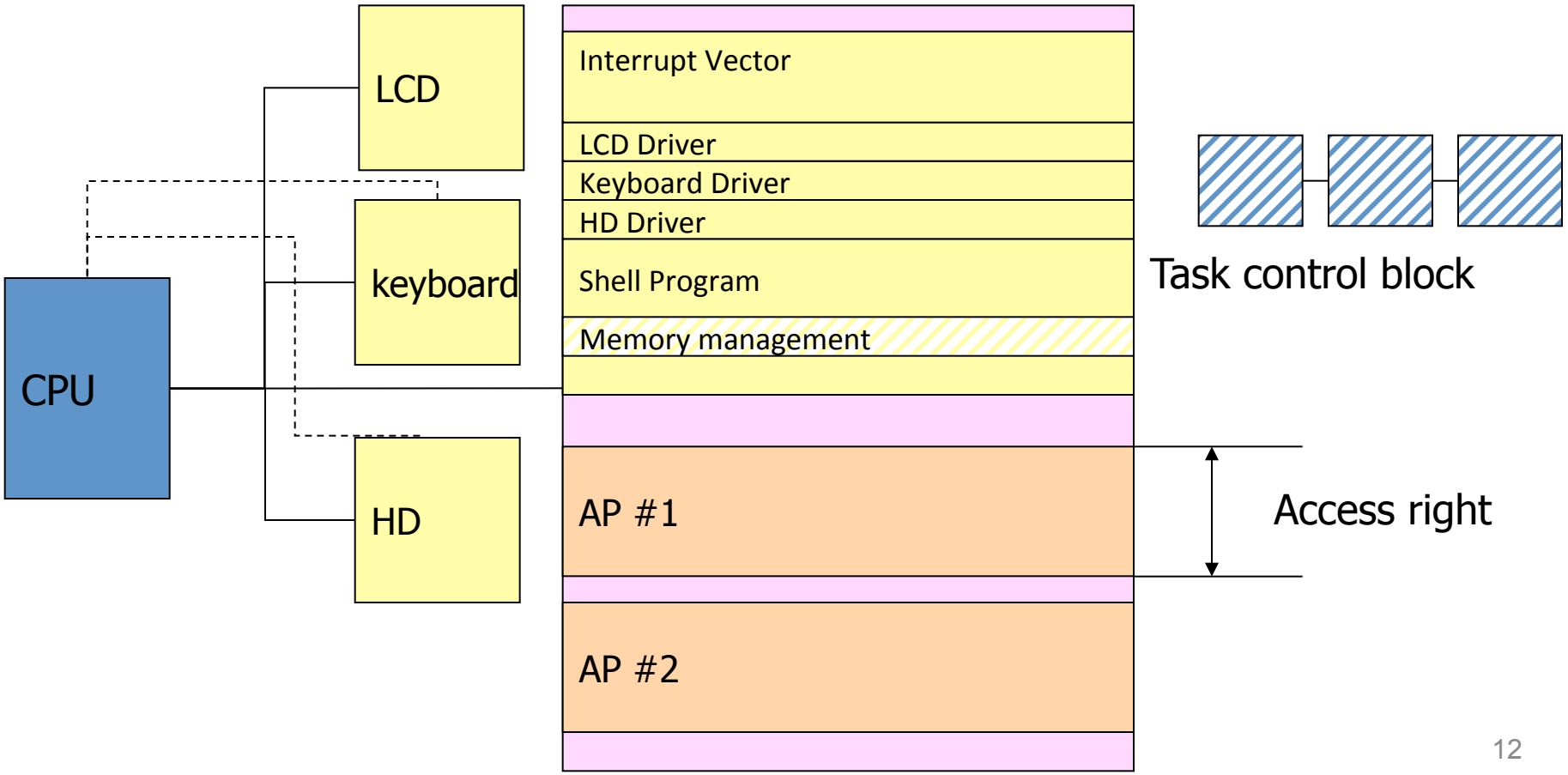
Let’s build an OS together

- Step 8: memory management



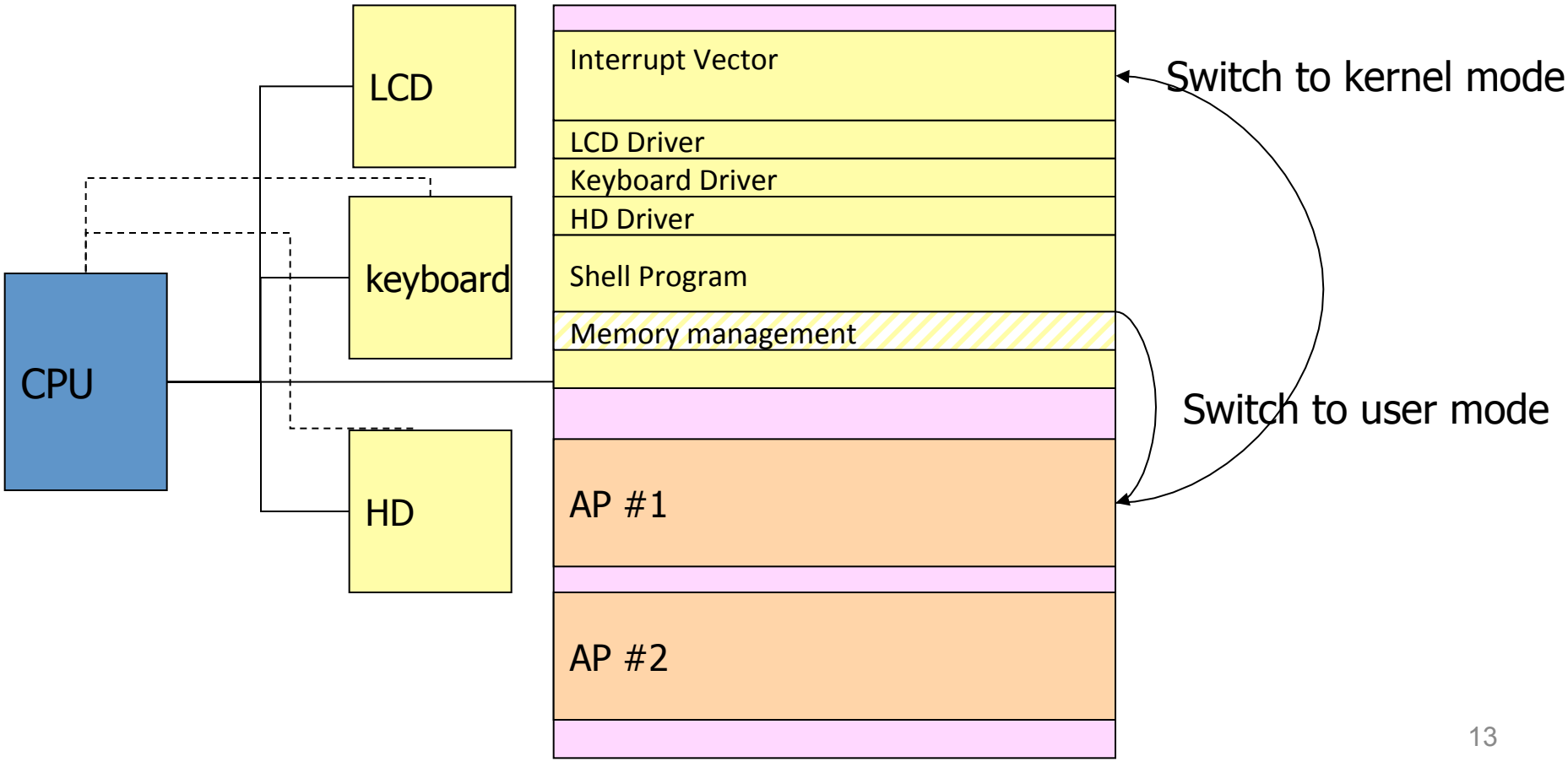
Let's build an OS together

- Step 9: protection



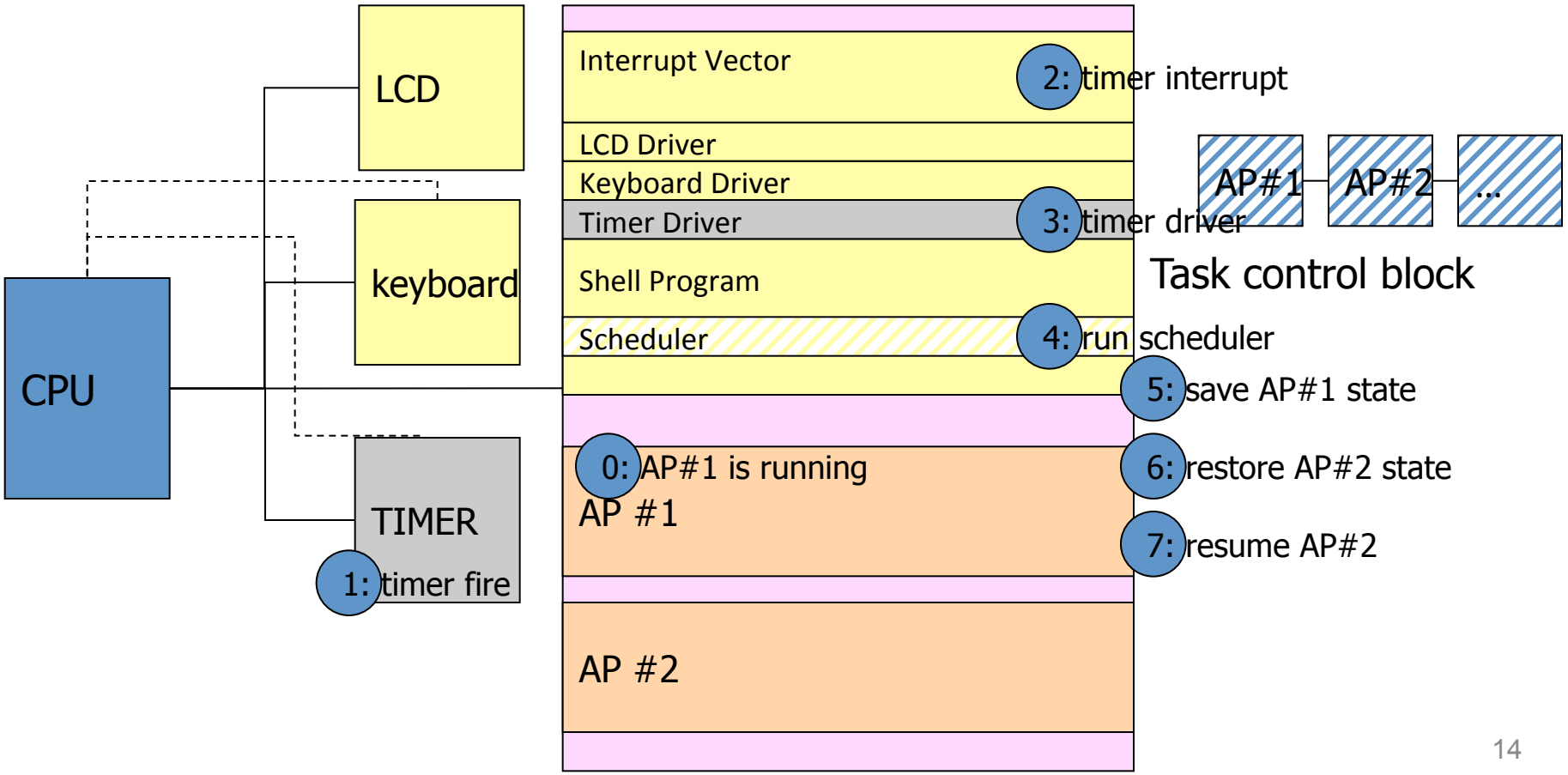
Let's build an OS together

- Step 10: how to protect OS kernel



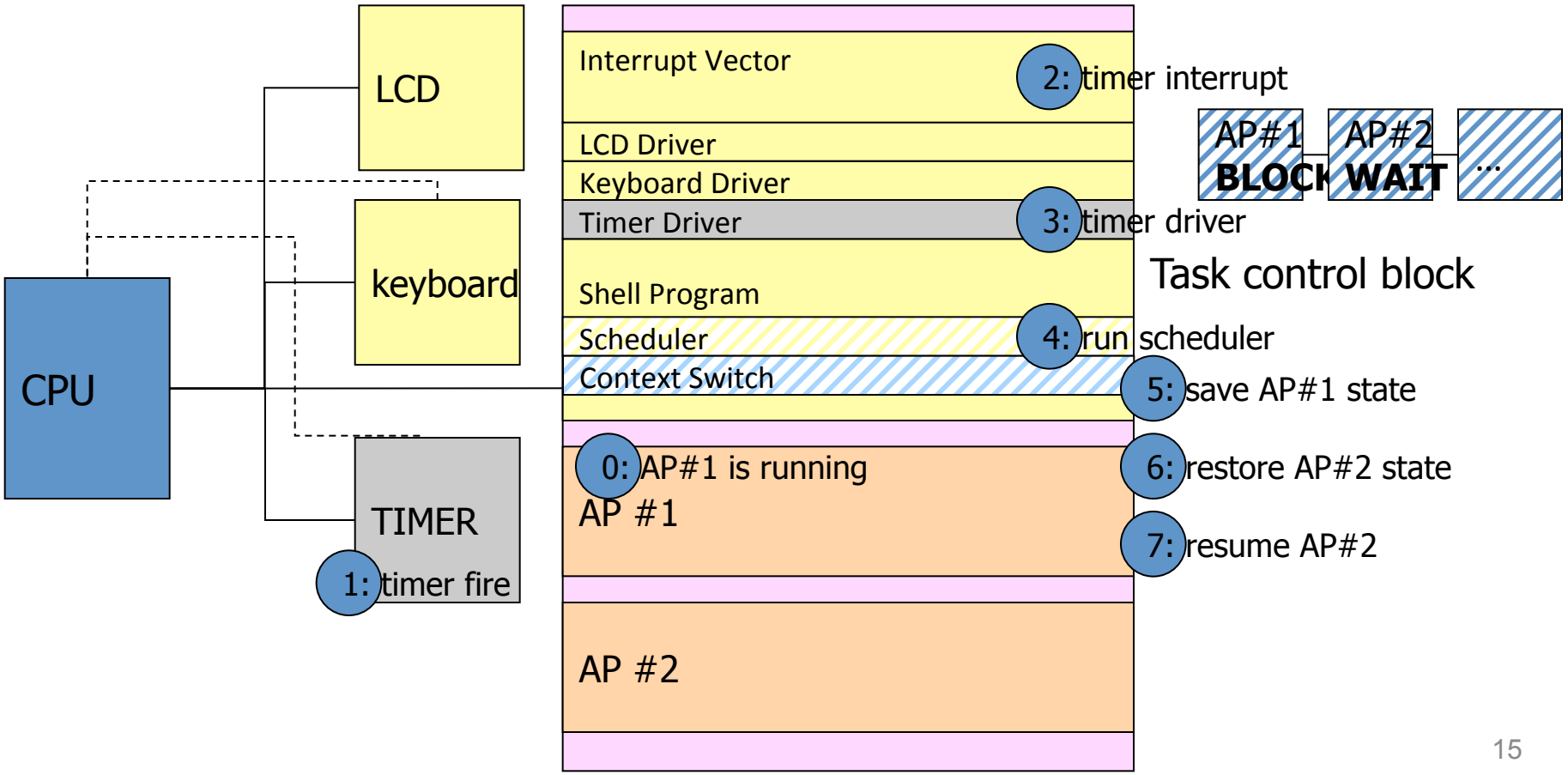
Let's build an OS together

- Step 11: let's go multi-tasking



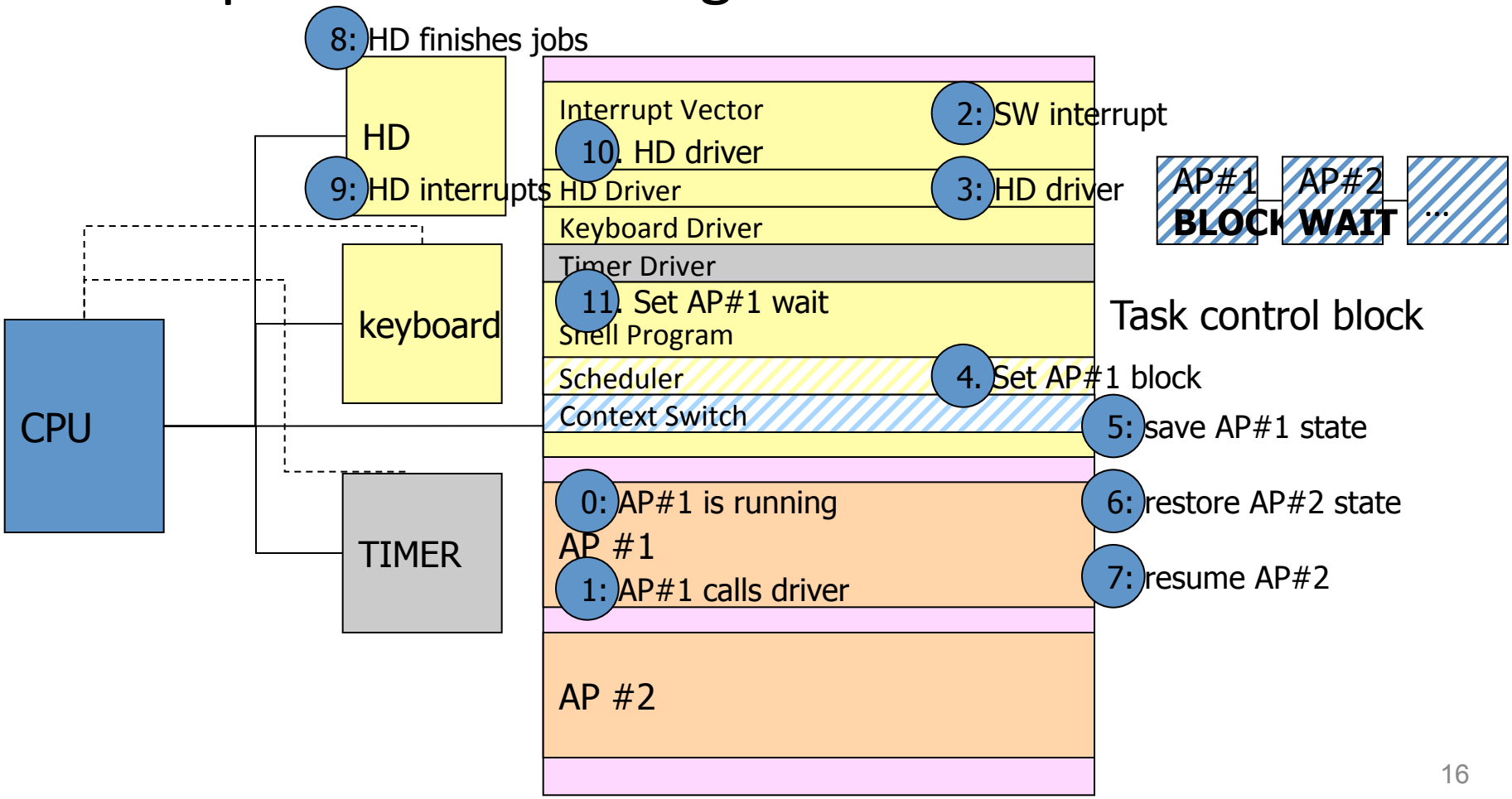
Let's build an OS together

- Step 12: process management



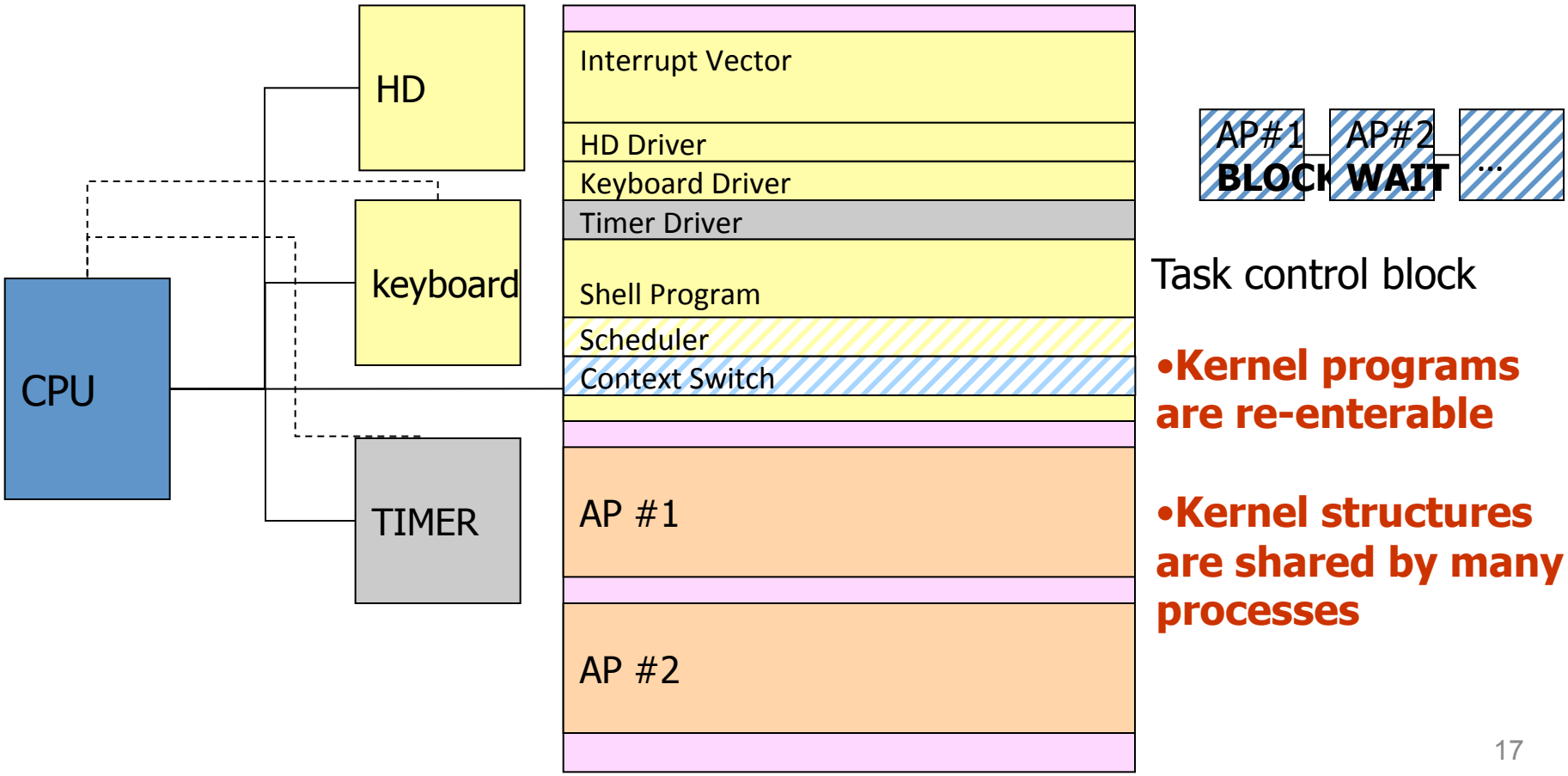
Let's build an OS together

- Step 13: multitasking and IO



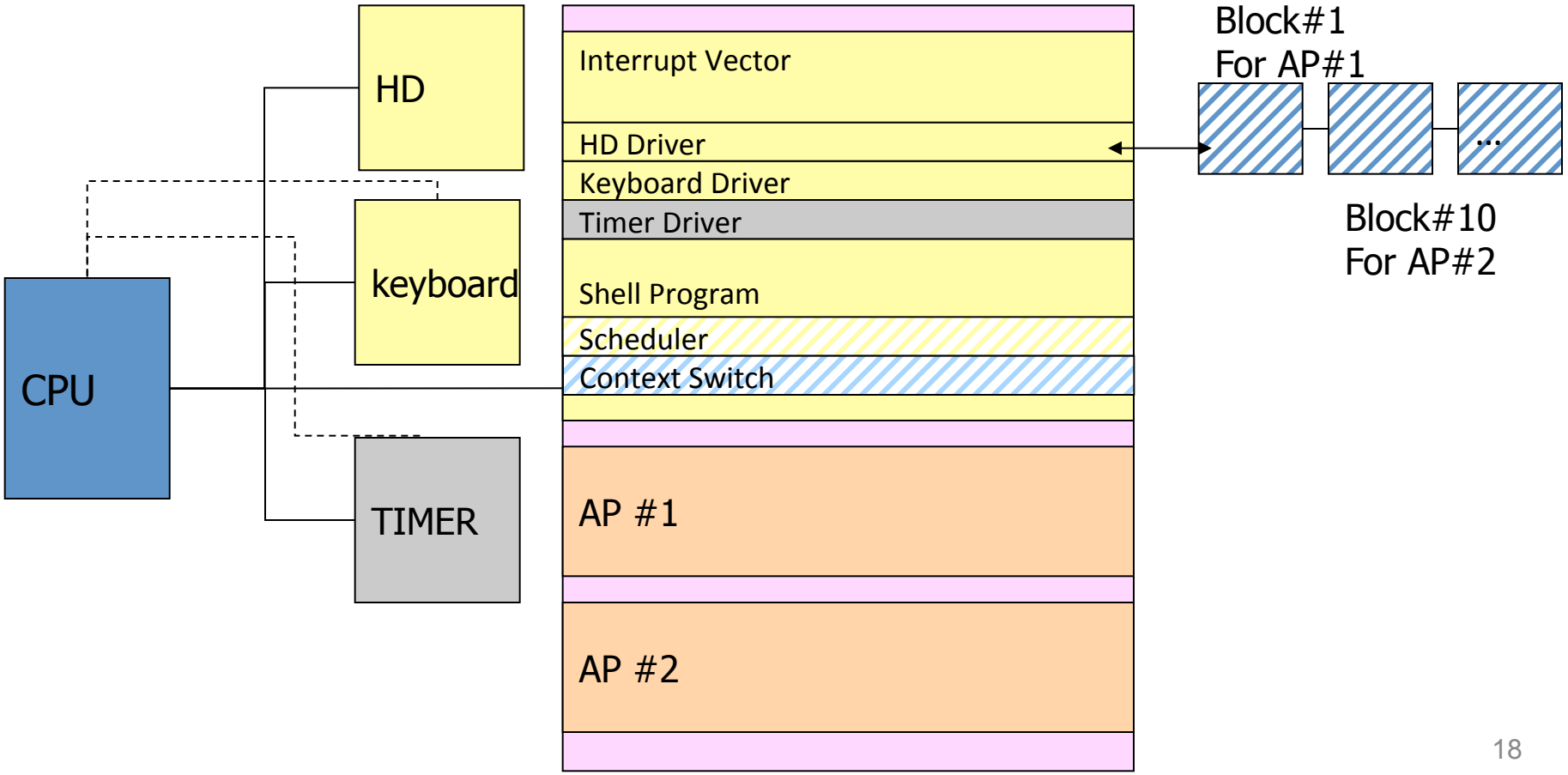
Let's build an OS together

- Step 14: kernel synchronization

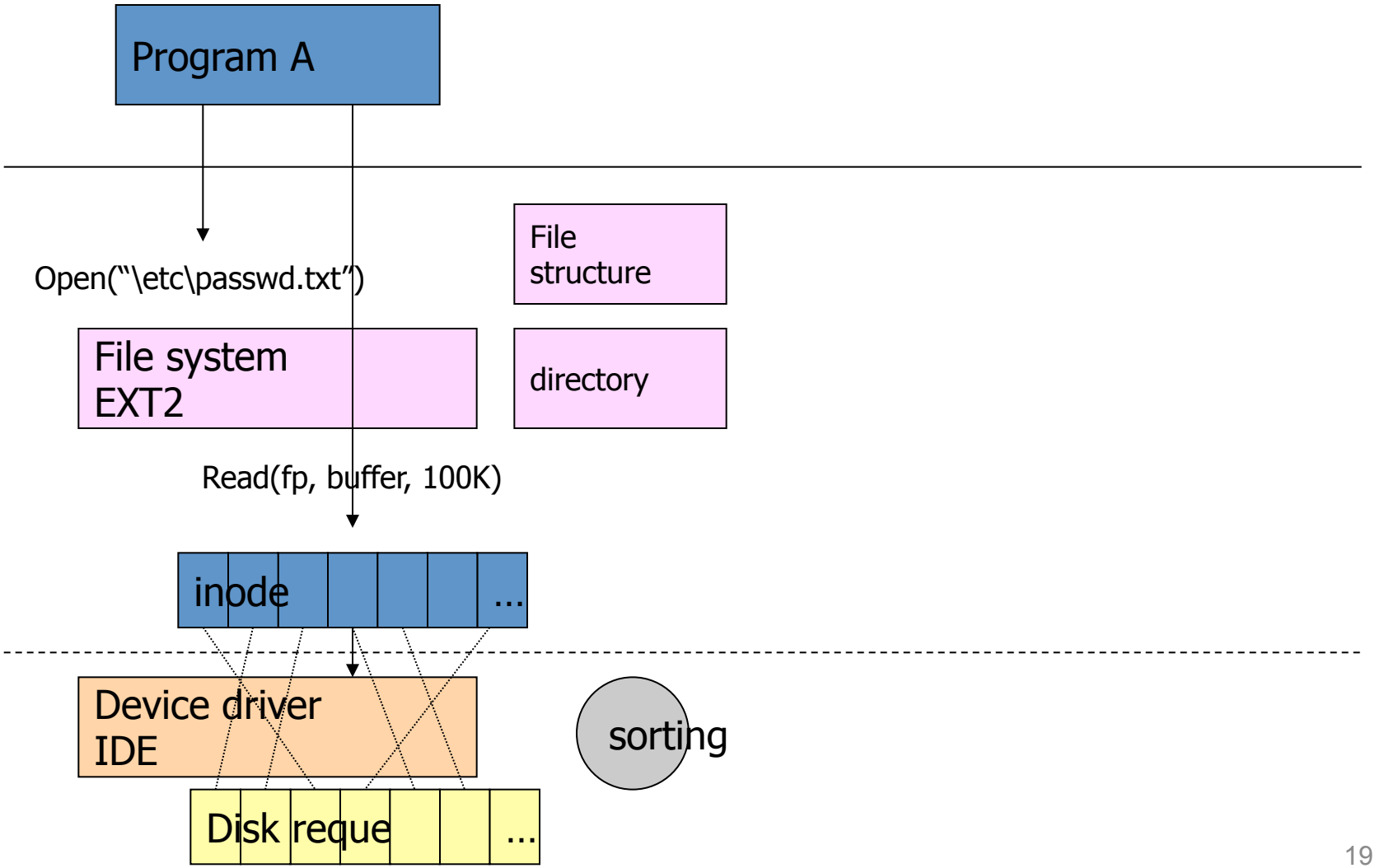


Let's build an OS together

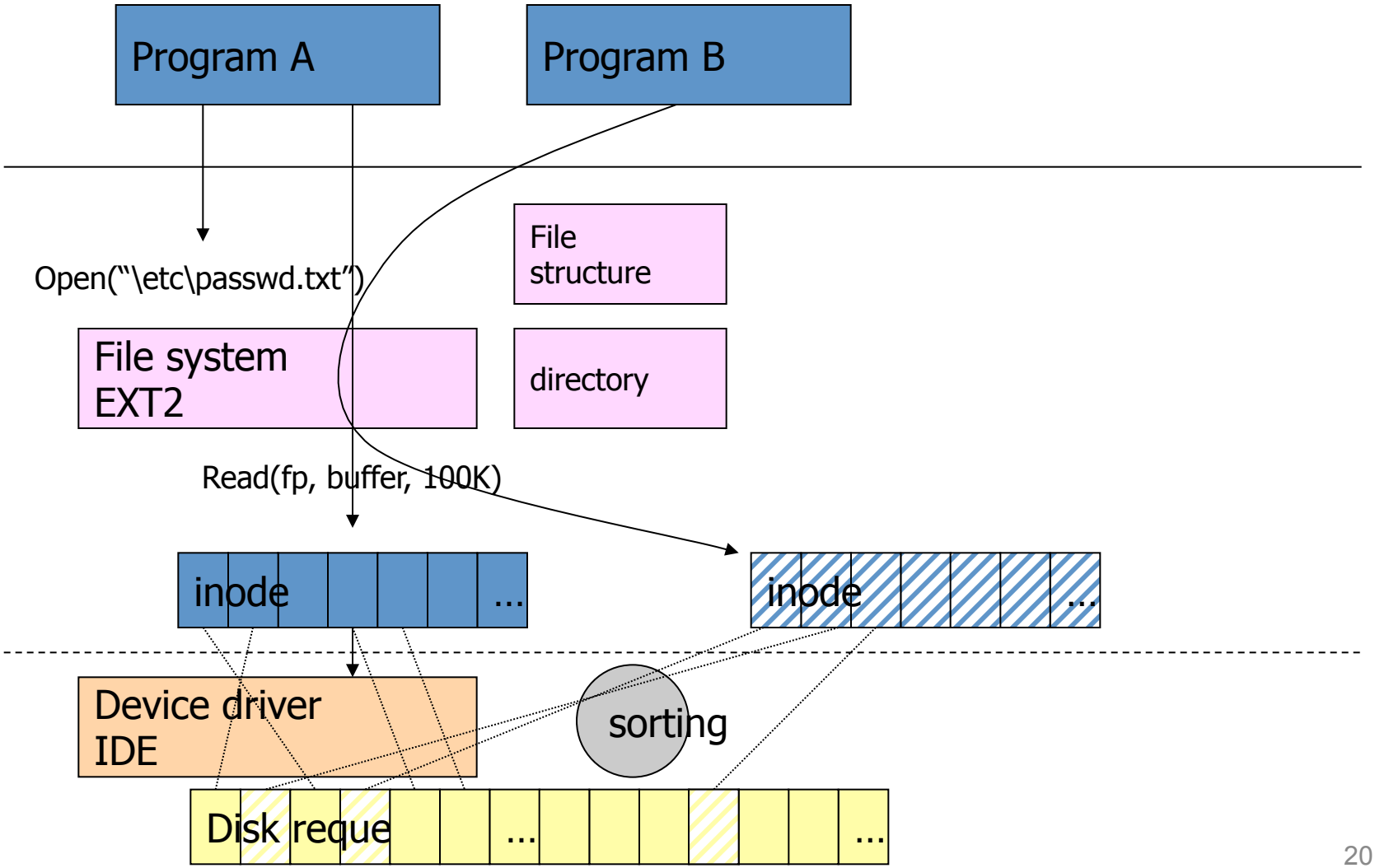
- Step 15: device driver



Process Schedule and I/O Queue

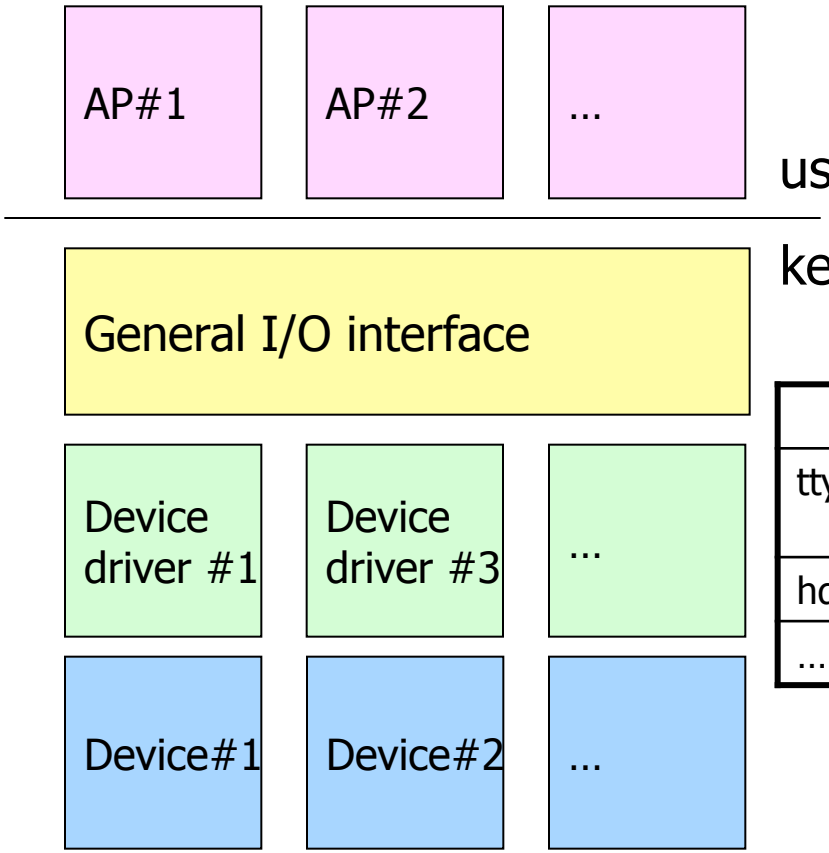


Process Schedule and I/O Queue



Let's build an OS together

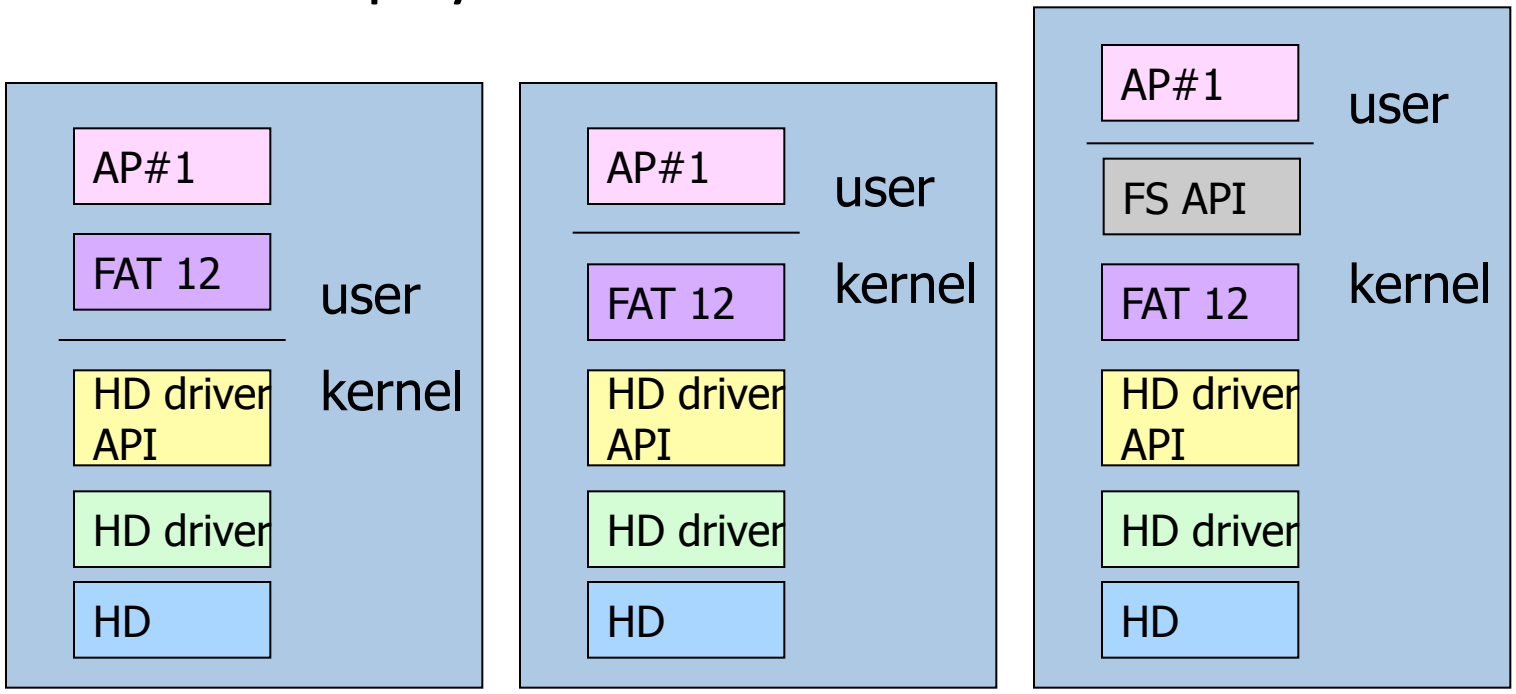
- Step 16: I/O subsystem



	open	close	read	...
tty0	tty0_open()	tty0_close()	tty0_read()	
hda1	...	...	...	
...				

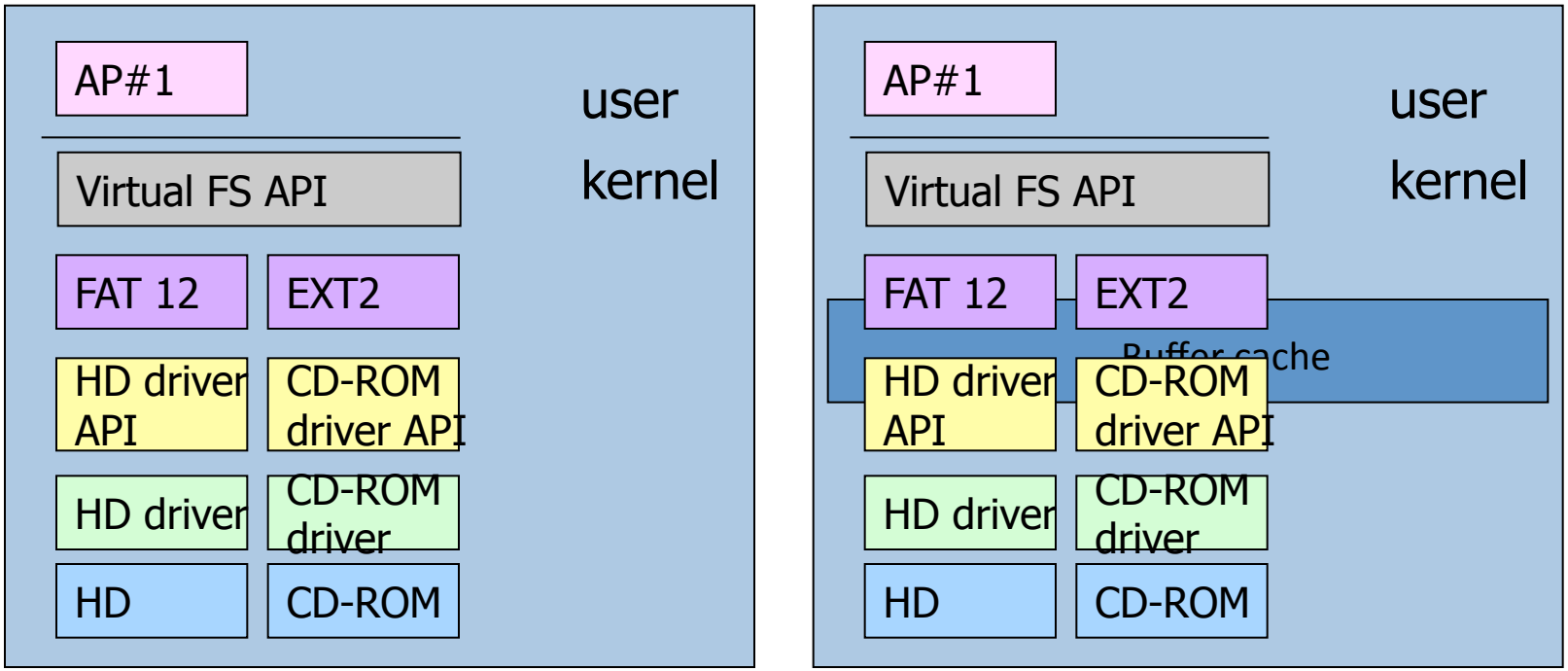
Let's build an OS together

- Step 17: file system
  - A kernel program to map the file directory/file name to physical blocks



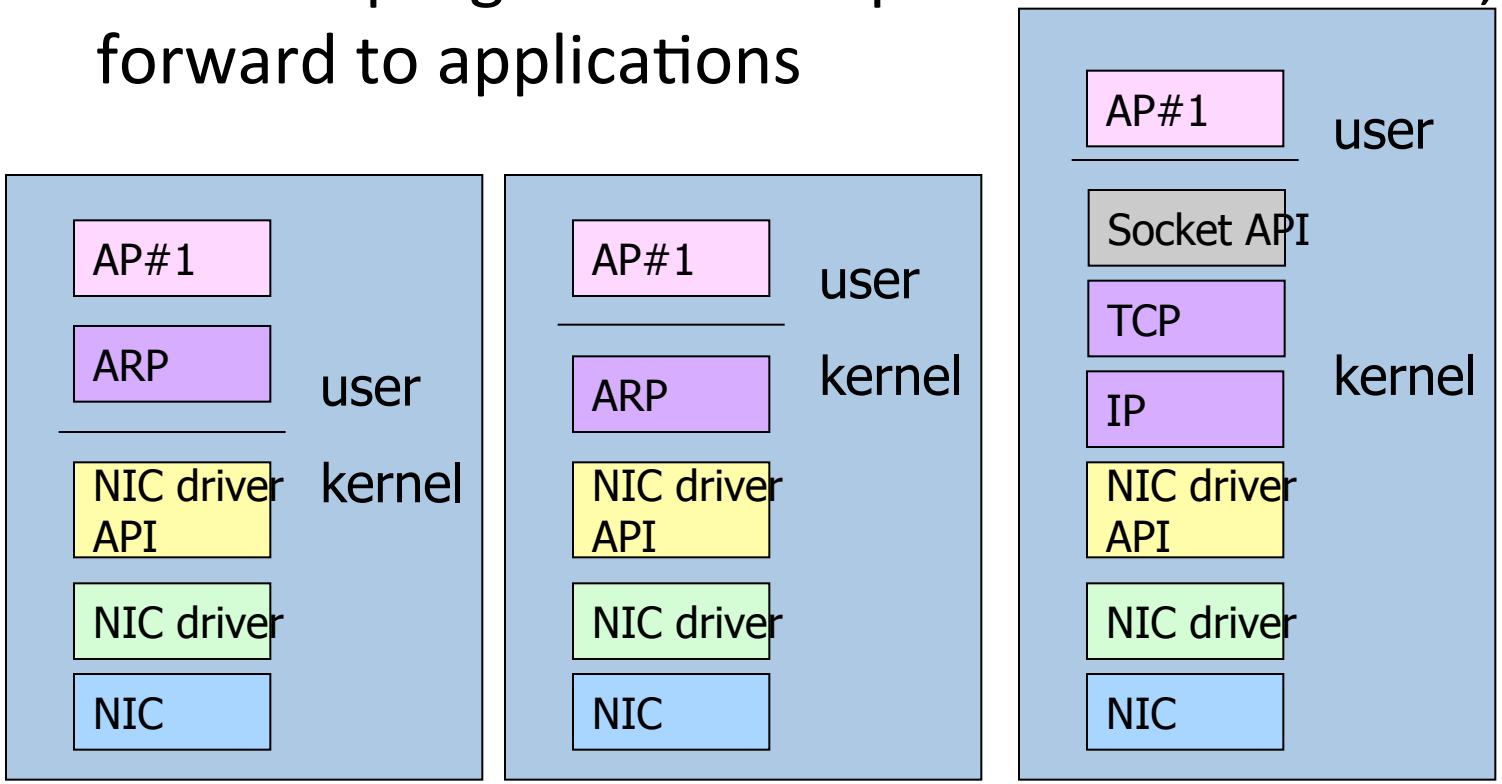
Let's build an OS together

- Step 17: file system

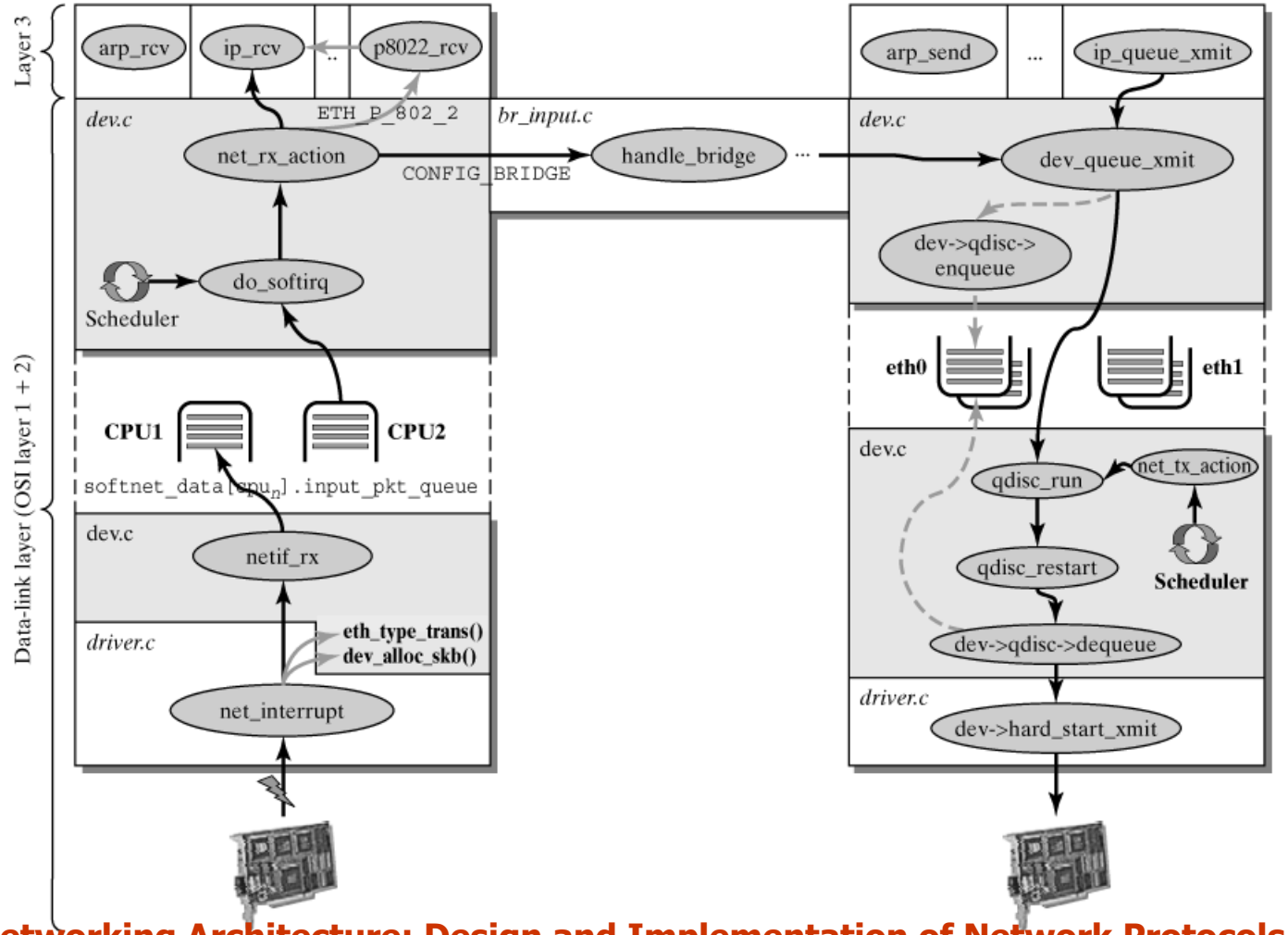


Let's build an OS together

- Step 18: network protocol stack
  - A kernel program to interpret received blocks, and forward to applications

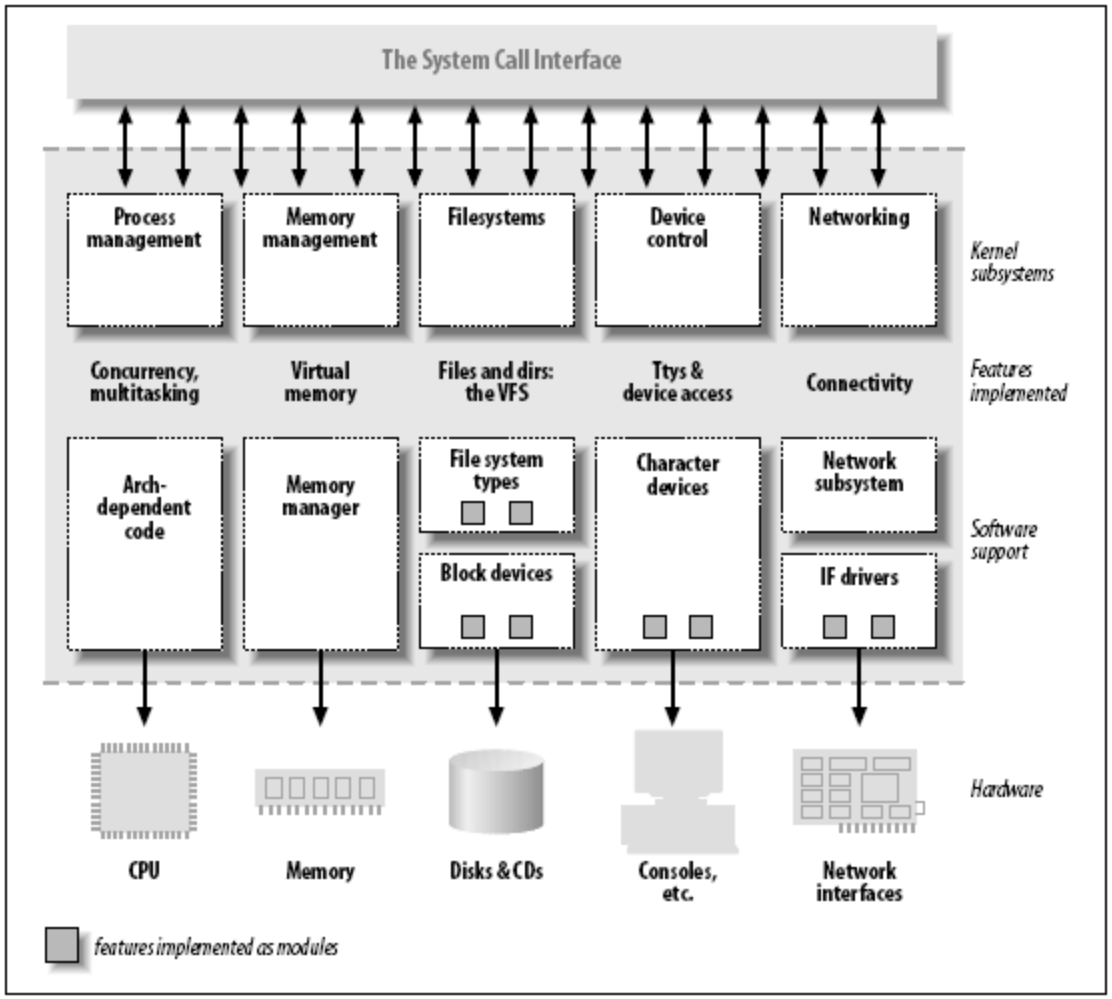


Let's build an OS together



The Linux® Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel By Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, Marc Bechler

Linux

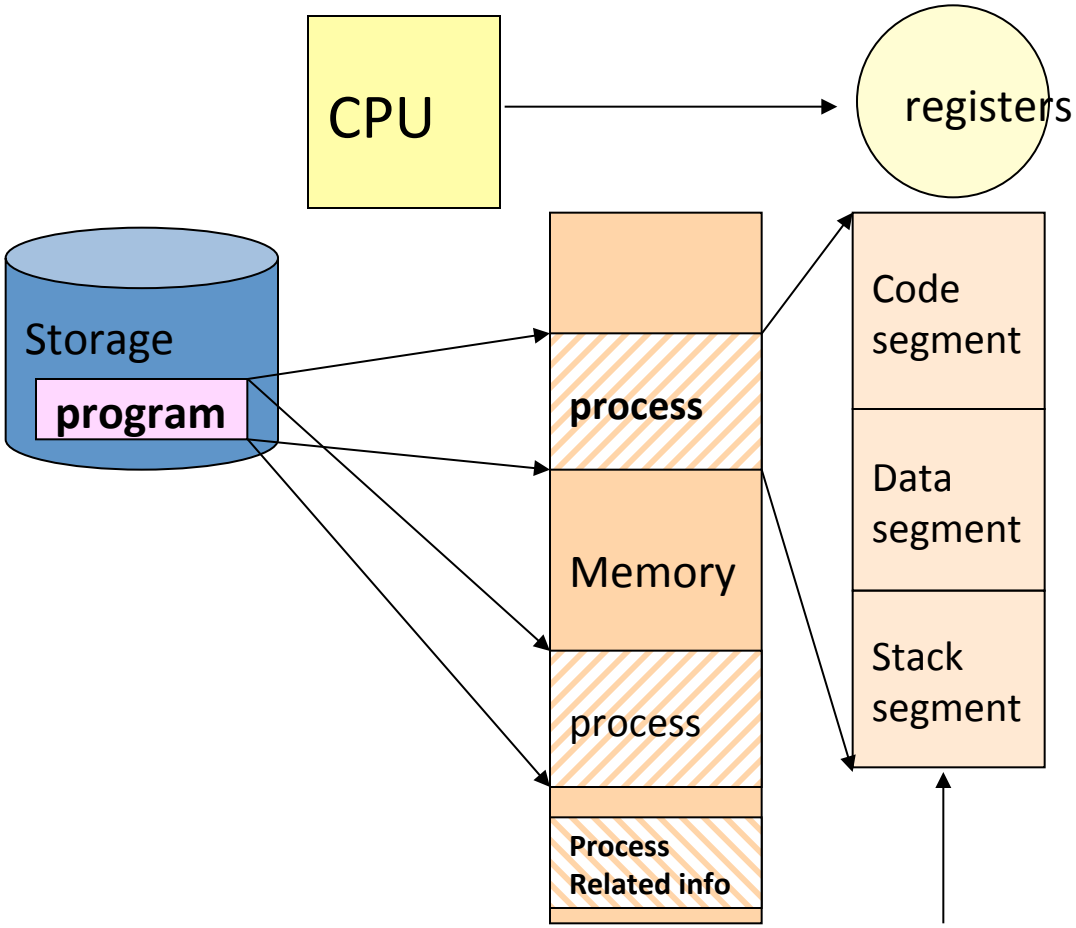

The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

Process Management

Process related terms

- Program
- Process (Task)
  - Lightweight process
- Thread
- Job

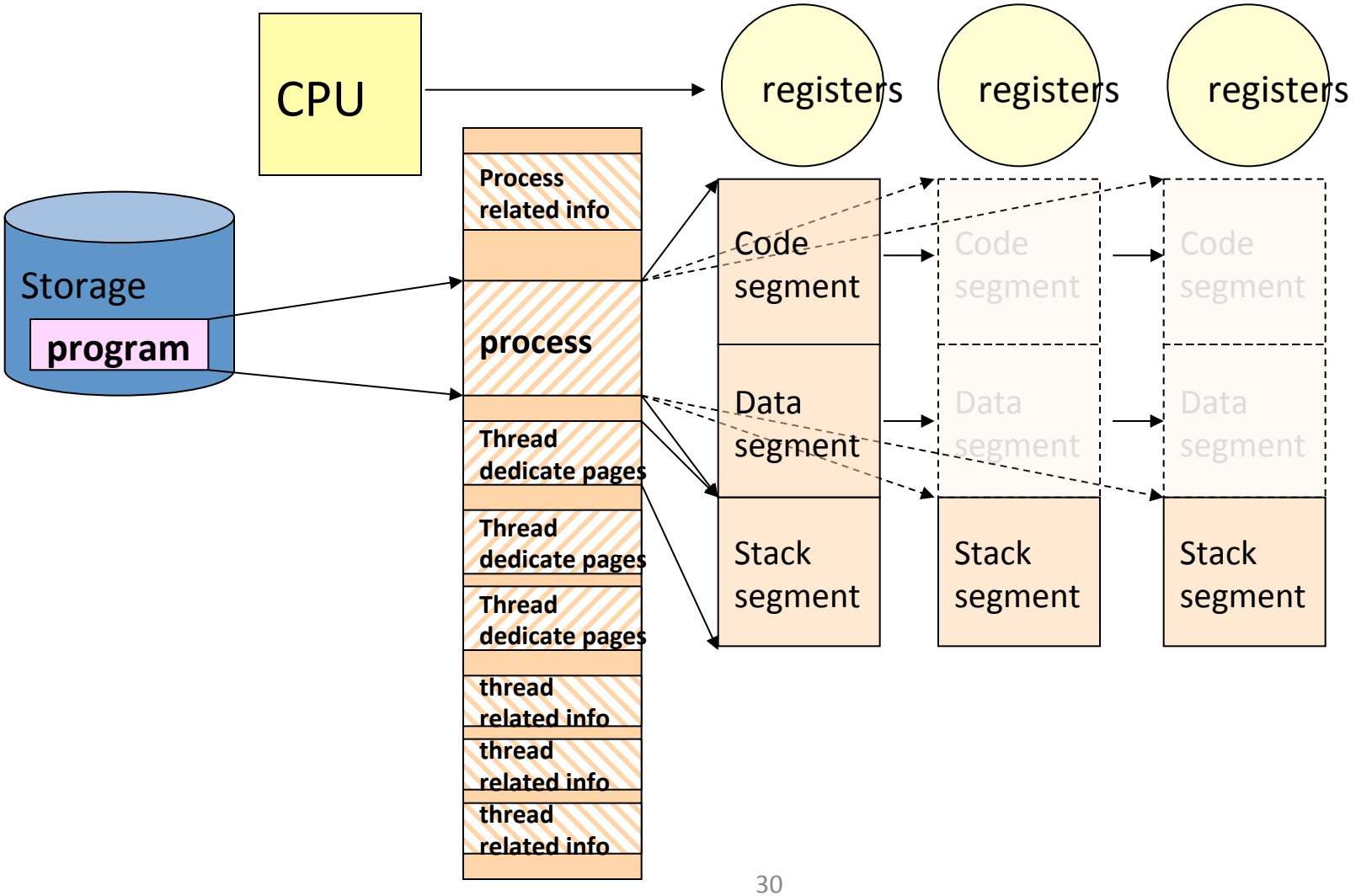
Process related terms



Is that good enough ?
If not, why ?

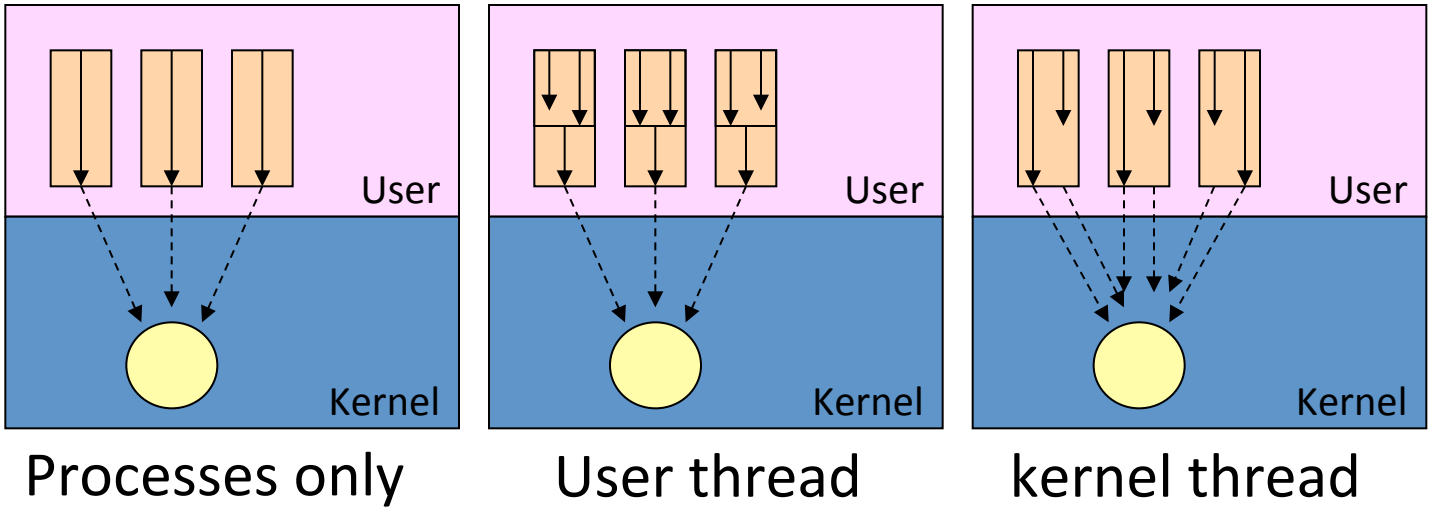
Physical memory might be discontinuous

Process related terms (Cont.)



Process related terms (Cont.)

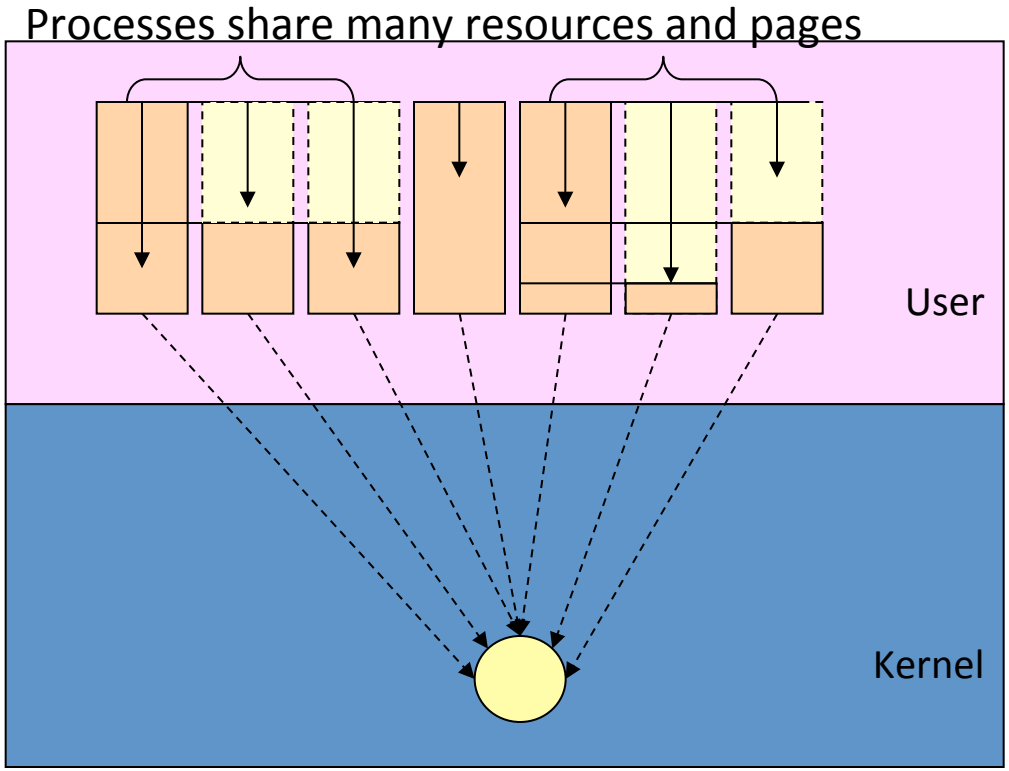
- Depending on OS designs



Processes only User thread kernel thread

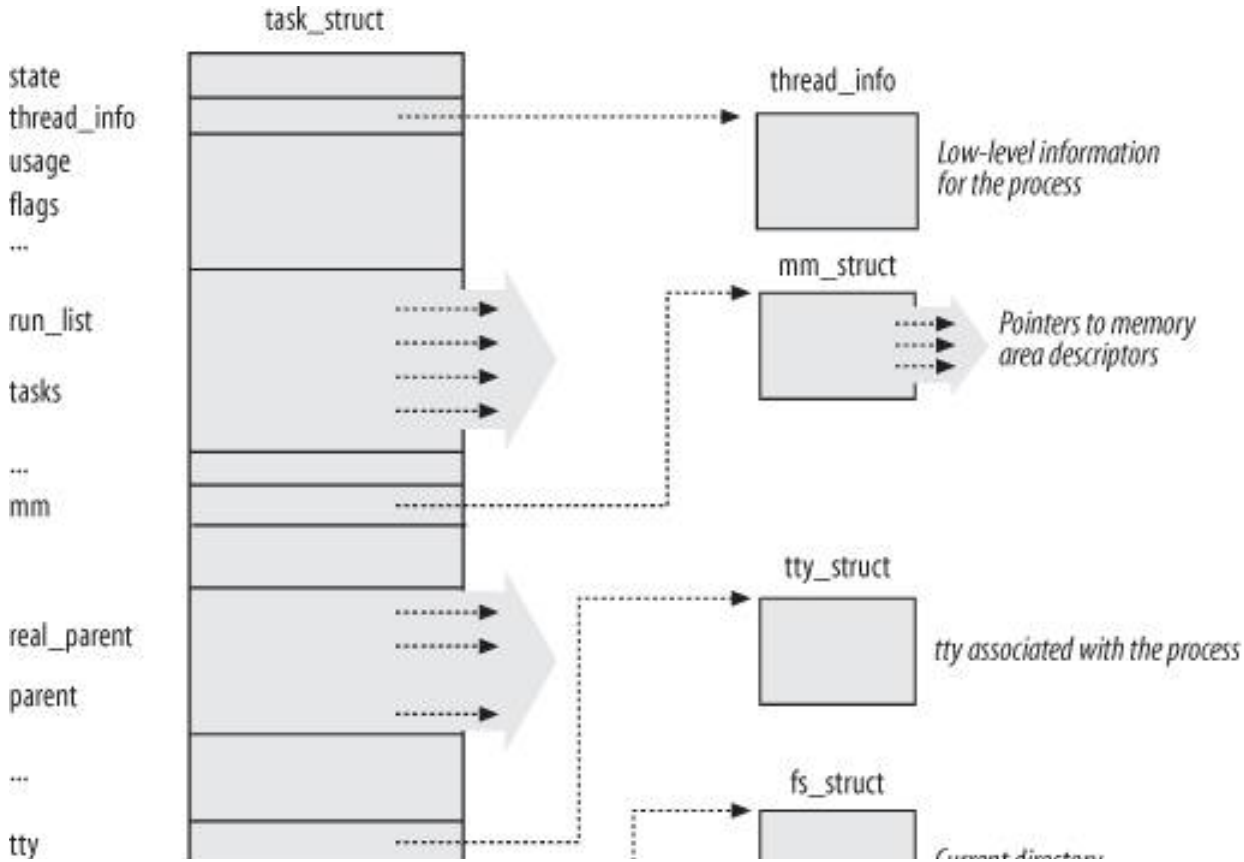
Process related terms (Cont.)

- Linux lightweight process



Process descriptor in Linux

- Linux process descriptor



Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 3rd Edition

Process descriptor in Linux

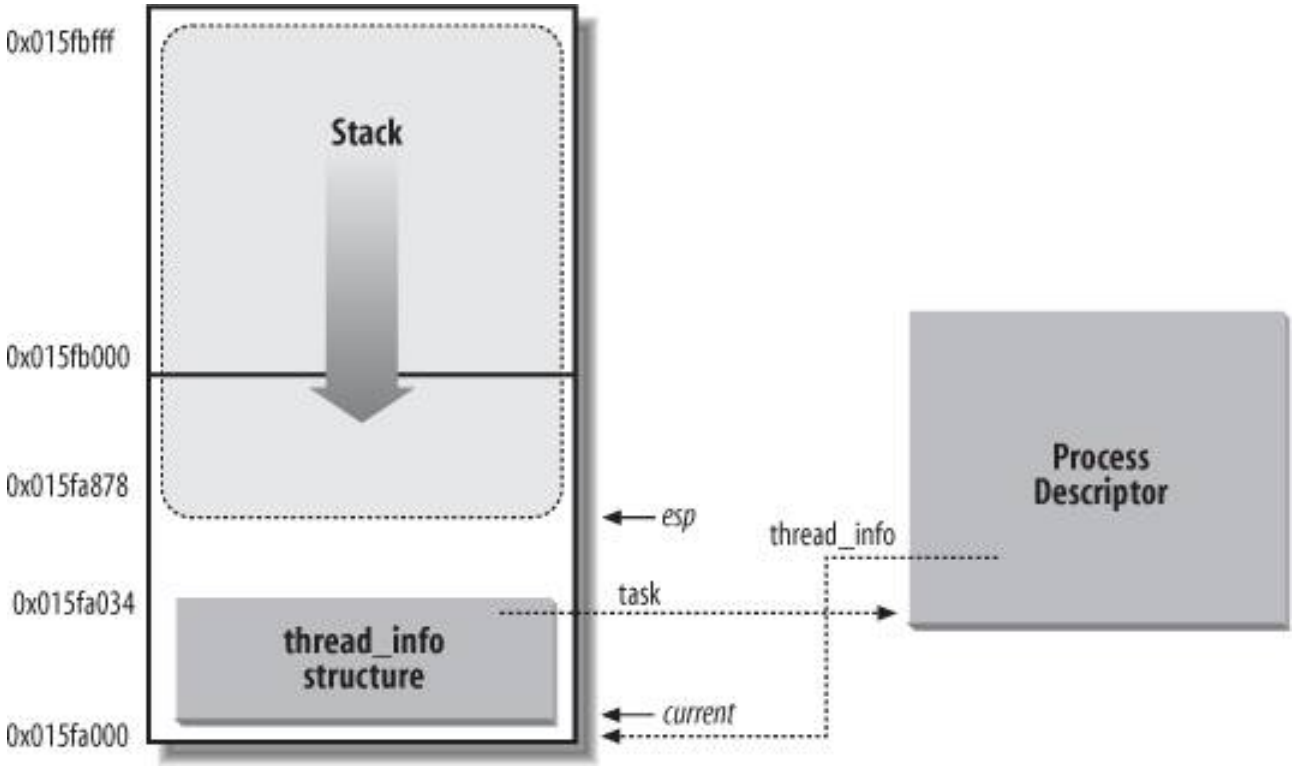
- Where Linux store the process descriptor
  - Of course memory
  - In kernel space – why ?
  - Per process (lightweight process)
  - Allocated by slab allocator (will be described in MM) – why ?
  - Co-located with process kernel stack
    - What is process kernel stack?
    - Why co-located with process kernel stack?

Process descriptor in Linux

- What is process kernel stack?
  - A stack space used while process is running at kernel mode – why separated with stack used in user mode?
- Why co-located with process kernel stack
  - Easy to access by stack pointer

Process descriptor in Linux

- Storing process descriptor and kernel stack



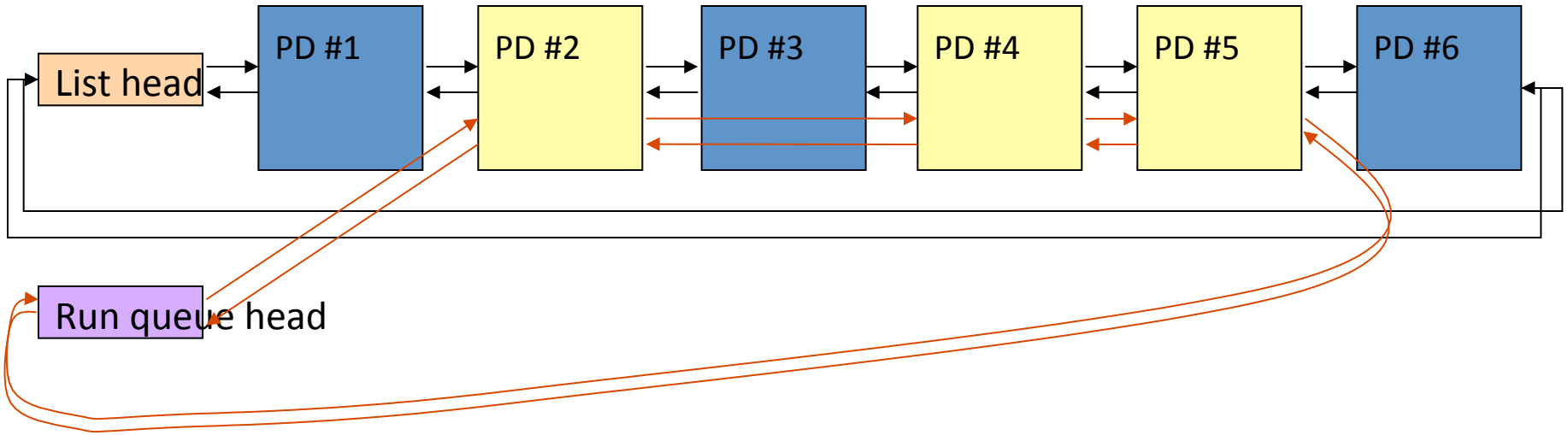
Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 3rd Edition

Process descriptor in Linux

- How these process descriptors are managed
  - Who & when create process descriptor?
    - “Parent” creates children & process descriptor
    - While (more precisely should be “before”) the process is created
  - Who & when destroy process descriptor?
    - People destroy themselves, “Parent” or “ancestor” destroy the process descriptor
    - While (more precisely should be “after”) the process is terminated
  - Who & when use process descriptor?
    - Scheduler
    - While scheduler is invoked

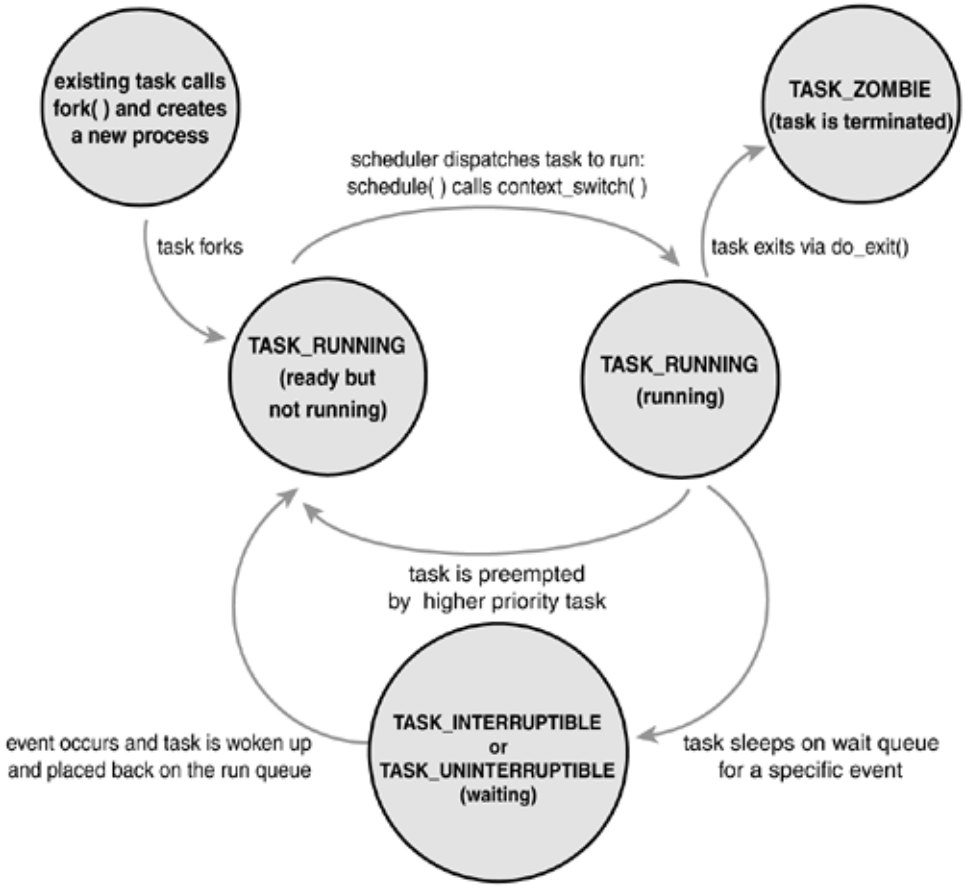
Process descriptor in Linux

- Process list
- Run queue(s) (per CPU per run queue)
- Parenthood relationships among processes
- The pidhash table



Process descriptor in Linux

- Linux process states



Robert Love, "Linux Kernel Development," Second Edition

Process creation and destroy in Linux

- User process/thread
  - create by parent
  - run alternatively in Kernel Mode and in User Mode
  - run kernel function through system calls
- Kernel process/thread
  - normally create while OS booting
  - run only in Kernel Mode
  - run specific kernel function
  - use only linear addresses (will be discussed later in MM)
  - process 0: **init** (executed during start_kernel())

Process creation and destroy in Linux

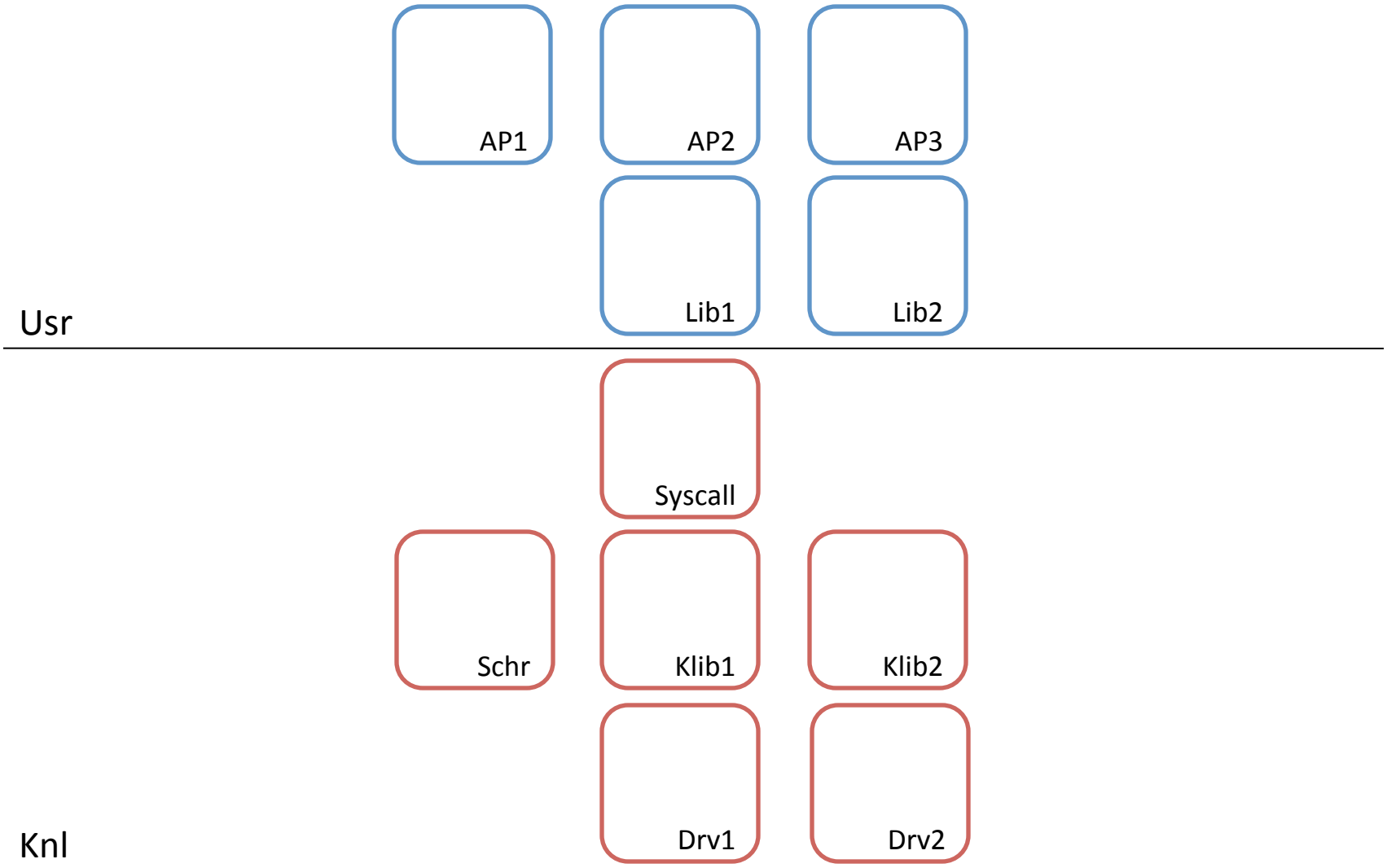
- Process creation flow
  - fork()
    - creates a child process that is a copy of the current task
    - it differs from the parent only in its PID, its PPID, and certain resources
    - copy-on-write
  - exec()
    - loads a new executable into the address space and begins executing it
  - fork()+exec()=spawn()

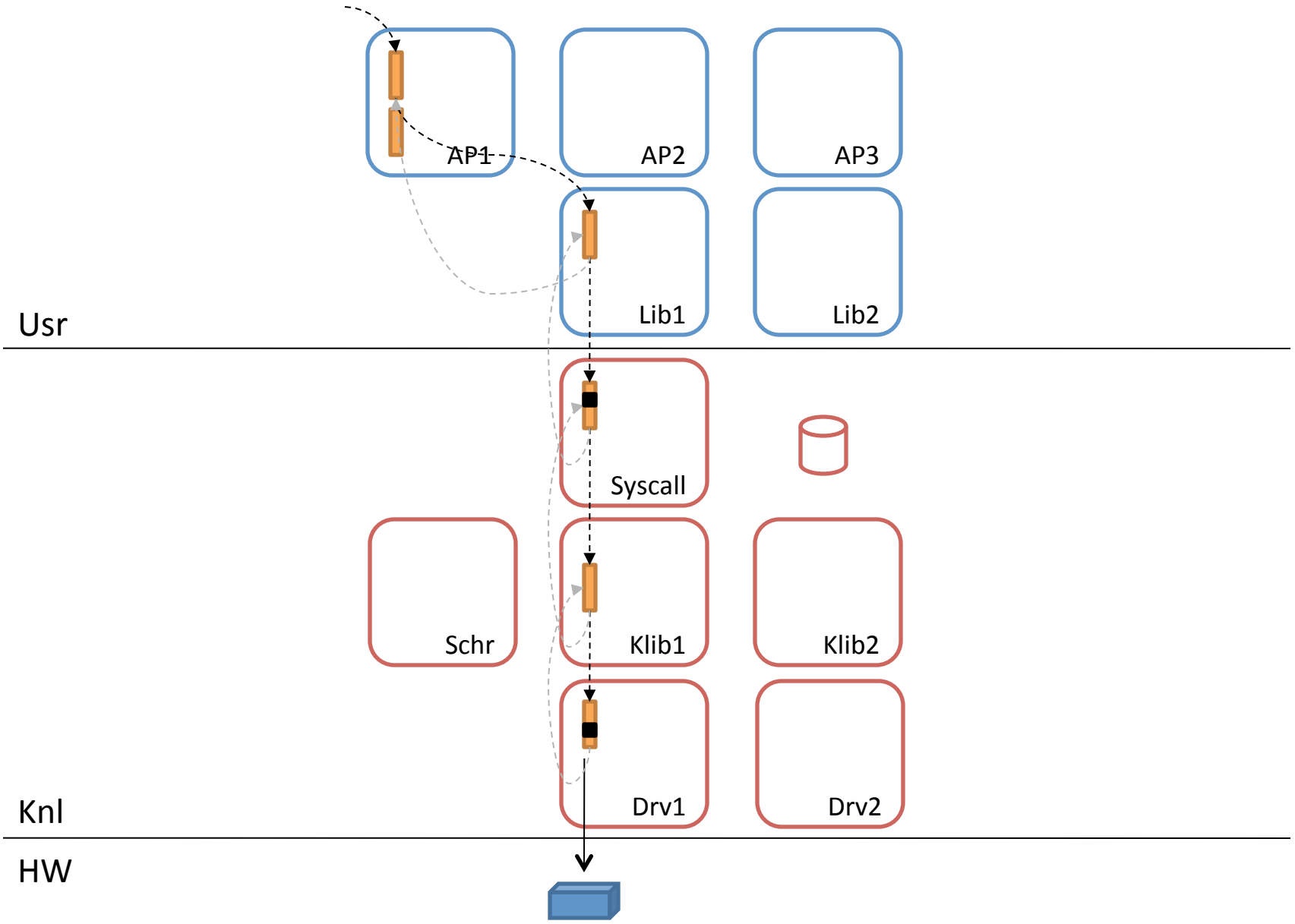
Process schedule and context switching in Linux

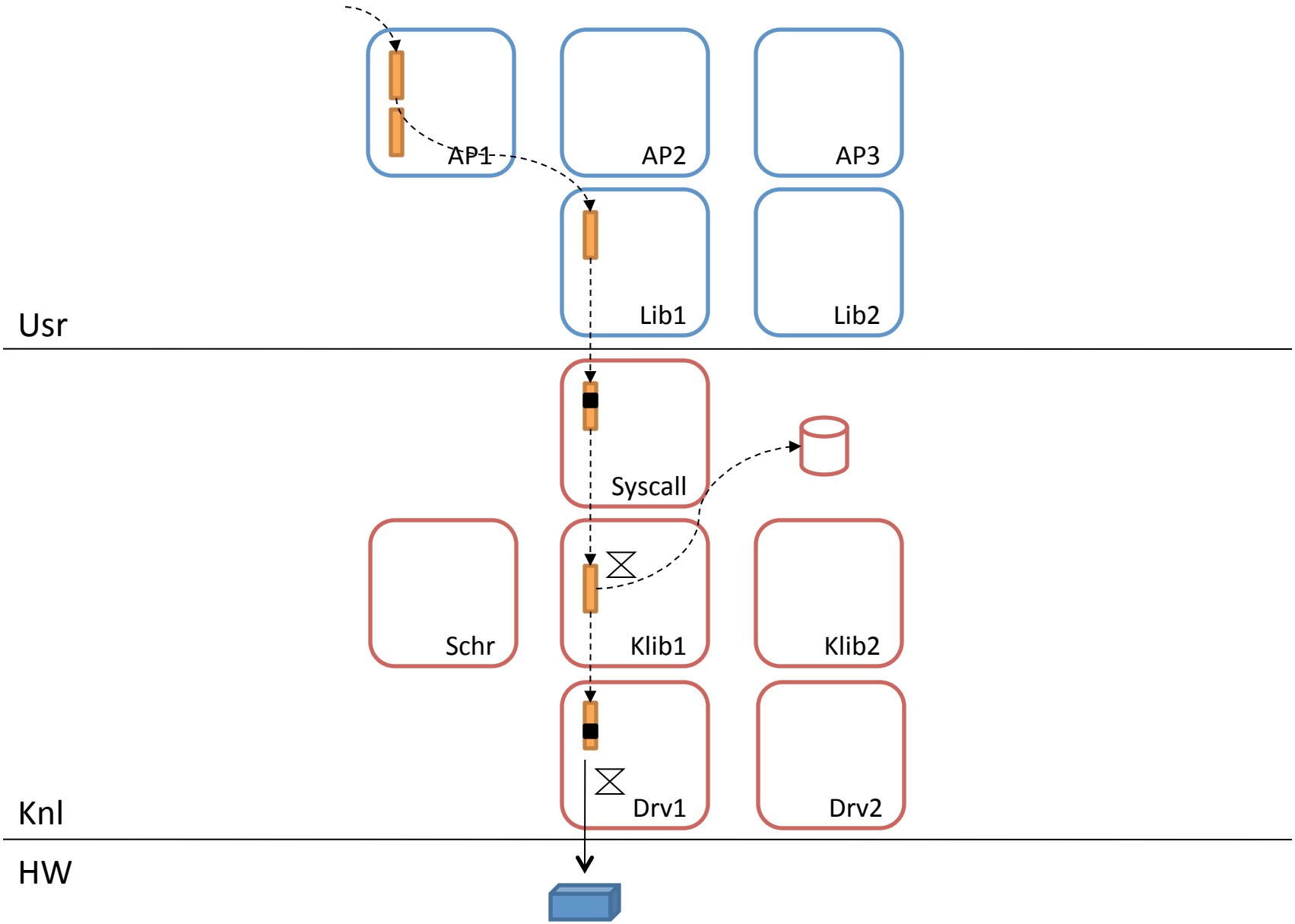
- Scheduling
  - Find the next suitable process to run
- Context switch
  - Store the context of the current process, restore the context of the next process

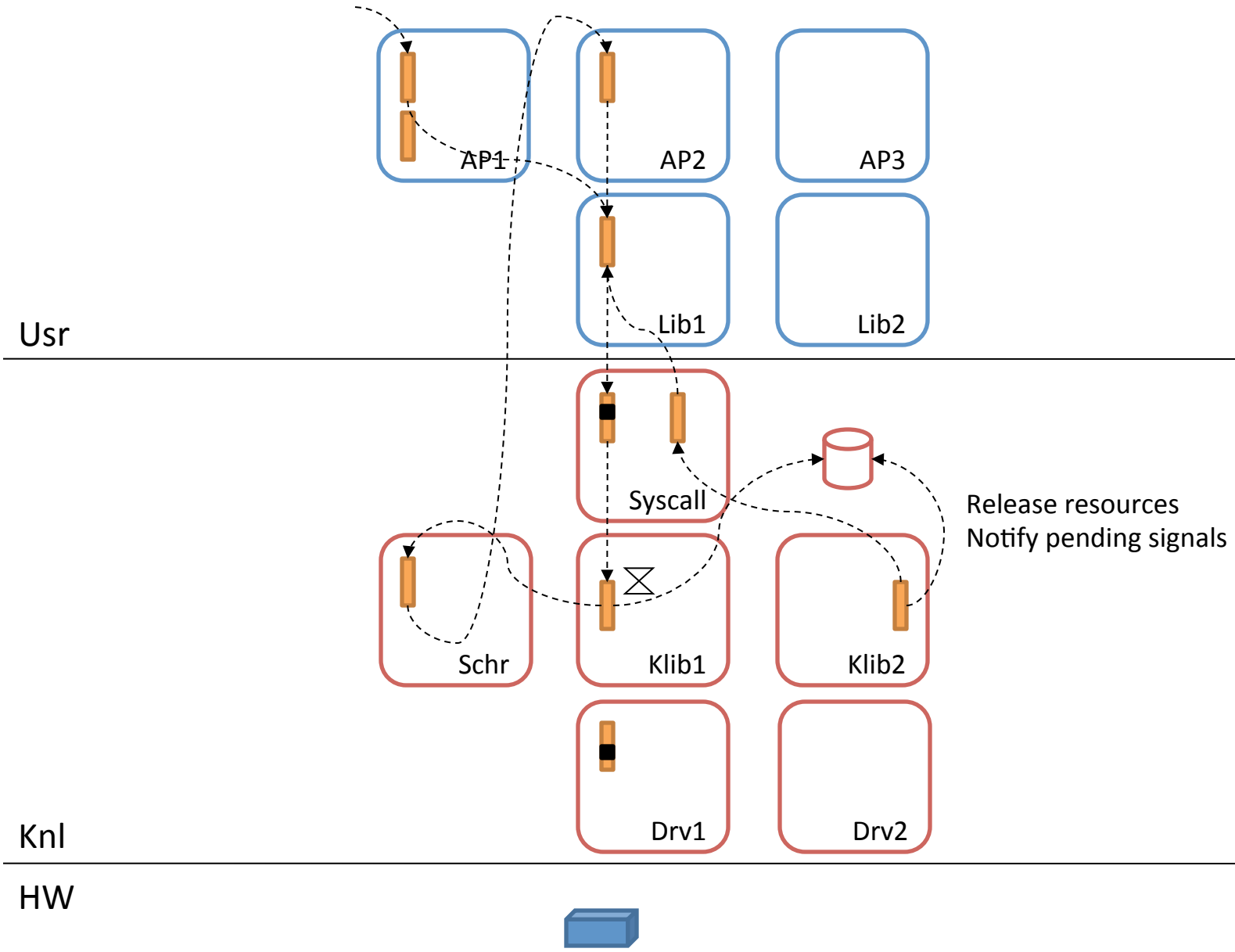
Process schedule and context switching in Linux

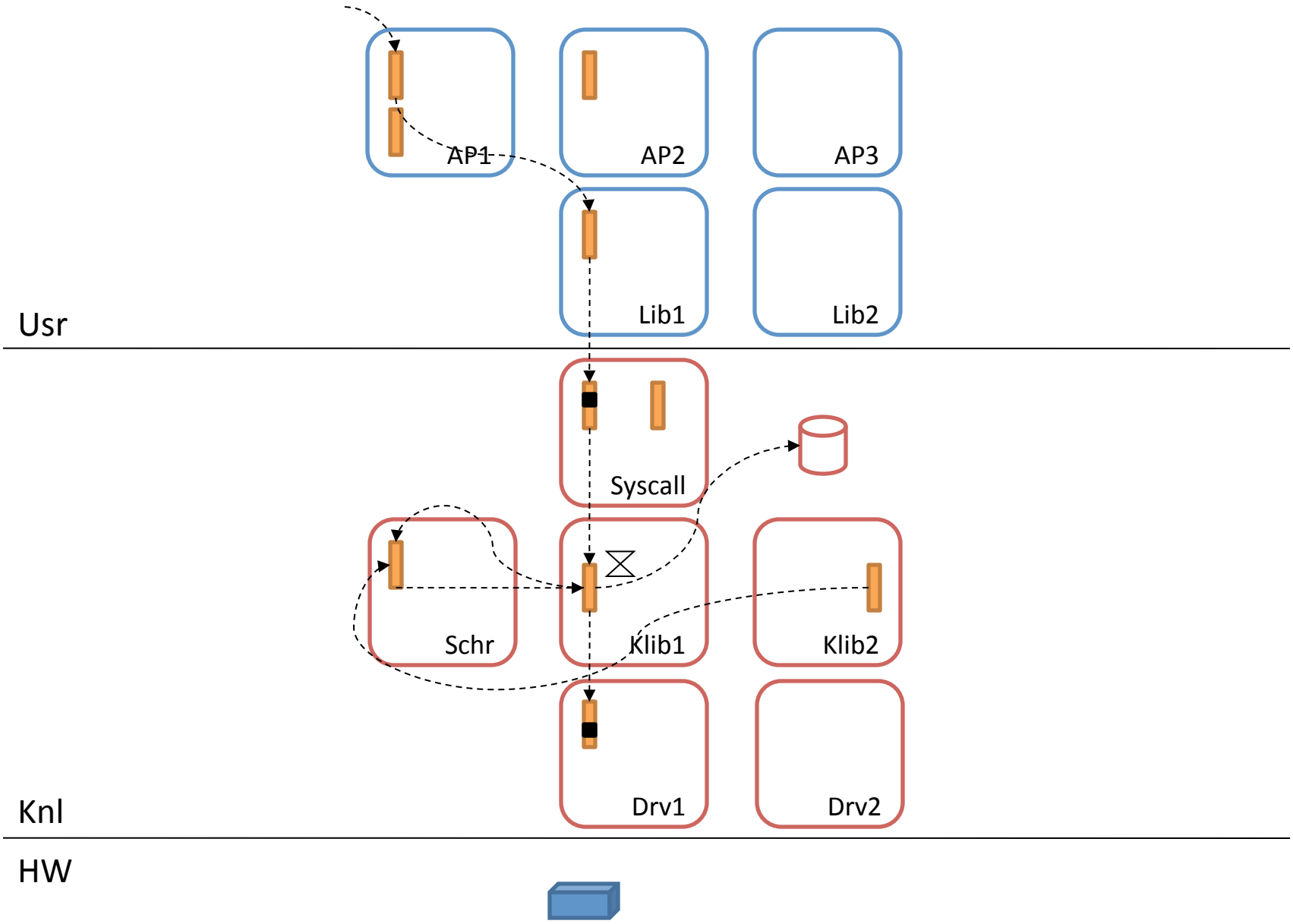
- When is the scheduler be invoked
  - Direct invocation vs. Lazy invocation
  - When returning to user-space from a system call
  - When returning to user-space from an interrupt handler
  - When an interrupt handler exits, before returning to kernel-space
  - If a task in the kernel explicitly calls schedule()
  - If a task in the kernel blocks (which results in a call to schedule())

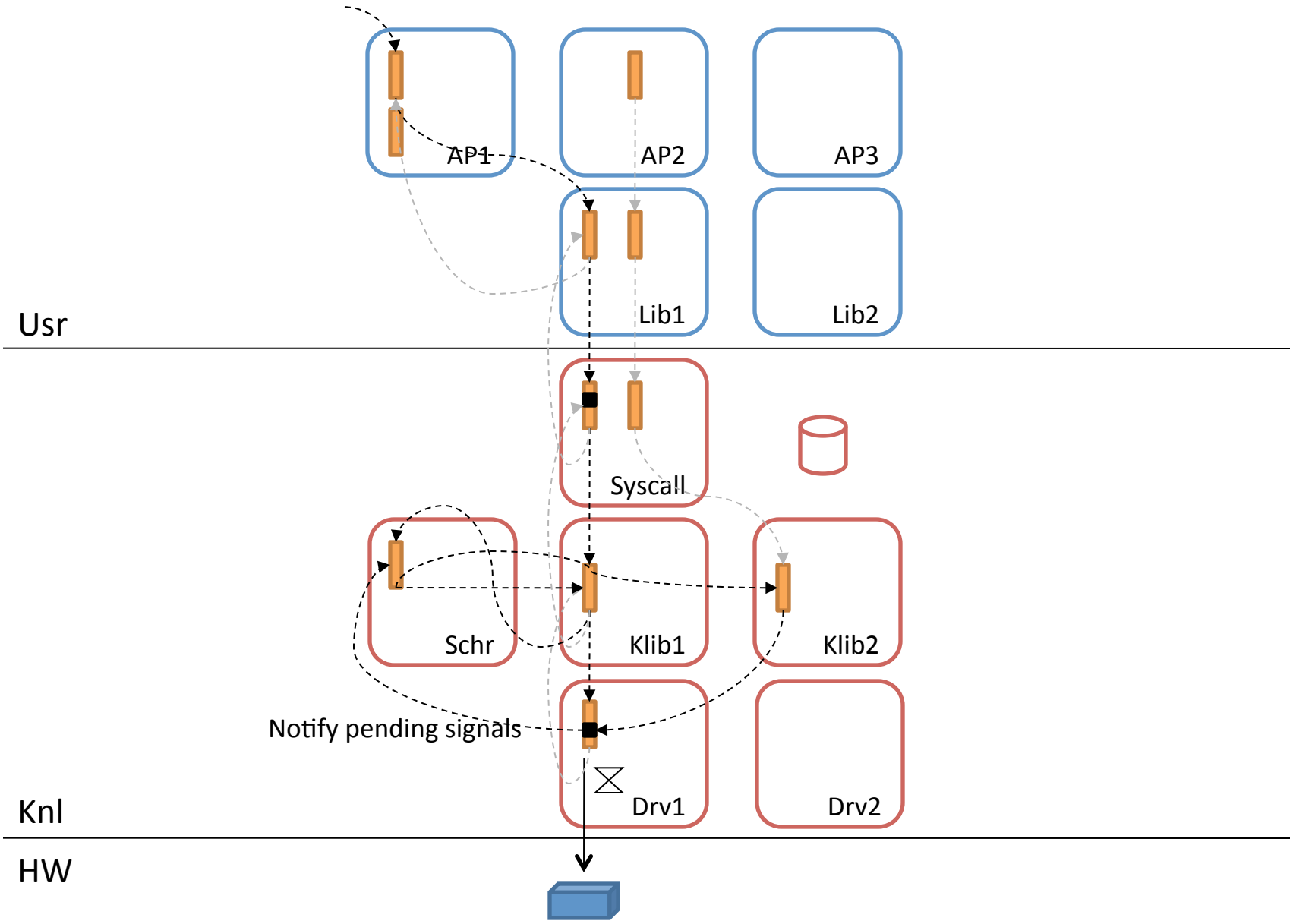

AP1
AP2
AP3
Lib1
Lib2
Usr
Syscall
Schr
Klib1
Klib2
Drv1
Drv2
Knl
HW


AP1
AP2
AP3
Lib1
Lib2
Usr
Syscall
Schr
Klib1
Klib2
Drv1
Drv2
Knl
HW

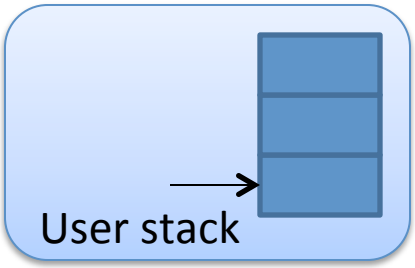

AP1
AP2
AP3
Lib1
Lib2
Usr
Syscall
Schr
Klib1
Klib2
Drv1
Drv2
Knl
HW


AP1
AP2
AP3
Lib1
Lib2
Usr
Syscall
Release resources
Notify pending signals
Schr
Klib1
Klib2
Drv1
Drv2
Knl
HW


AP1
AP2
AP3
Lib1
Lib2
Usr
Syscall
Schr
Klib1
Klib2
Drv1
Drv2
Knl
HW


AP1
AP2
AP3
Lib1
Lib2
Usr
Syscall
Schr
Klib1
Klib2
Notify pending signals
Drv1
Drv2
Knl
HW

Process Context



CPU runs user codes

System calls

int $0x80/Exception vector 128/system_call()

eax: system no.

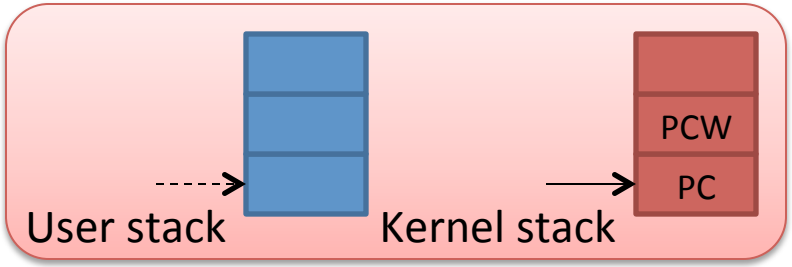
User space

Kernel space

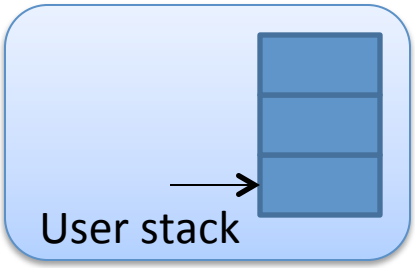
System call complete

CPU runs kernel on behalf of user process

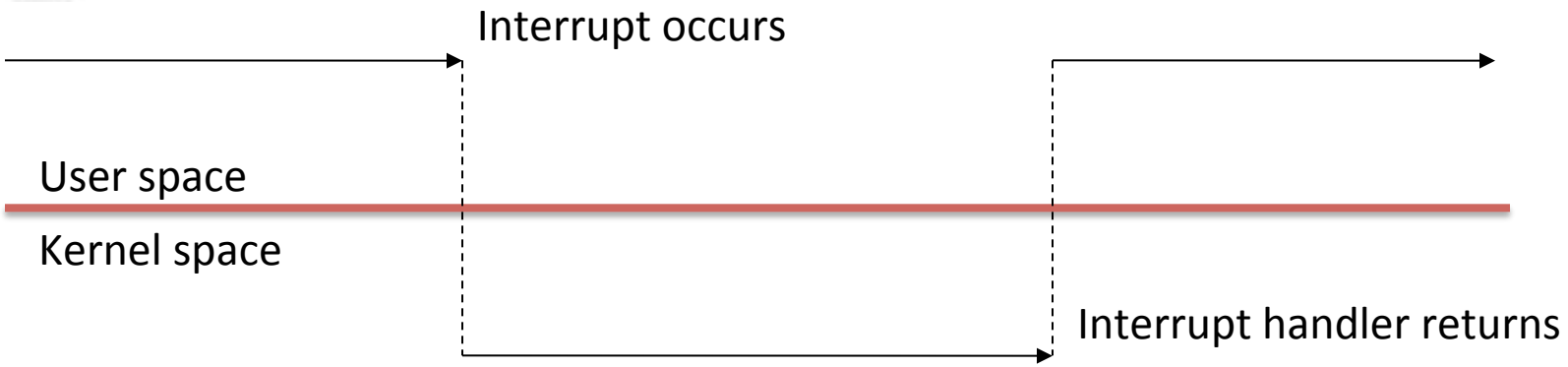
Kernel is in process context



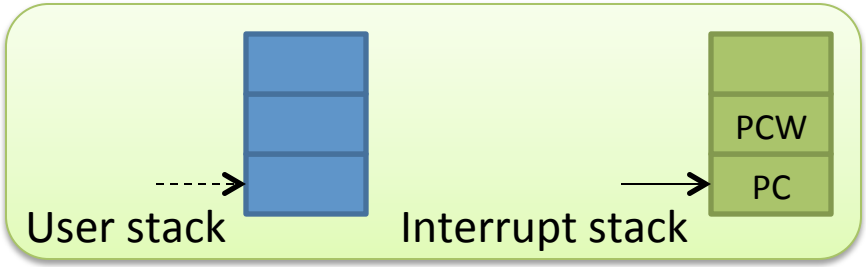
Interrupt Context



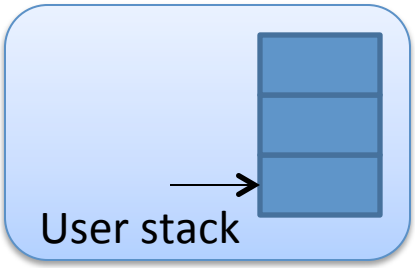
CPU runs user codes



CPU runs interrupt handler (or called ISR) in the kernel space
Kernel is in interrupt context



Interrupt Context



CPU runs user codes

System calls

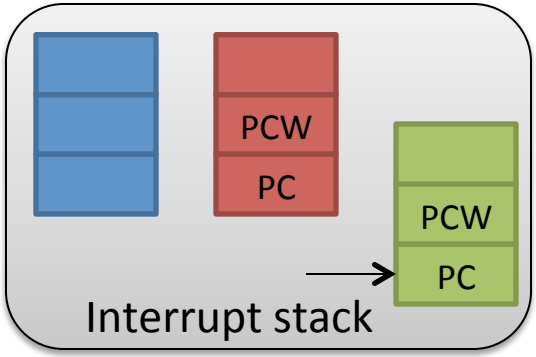
User space

Kernel space

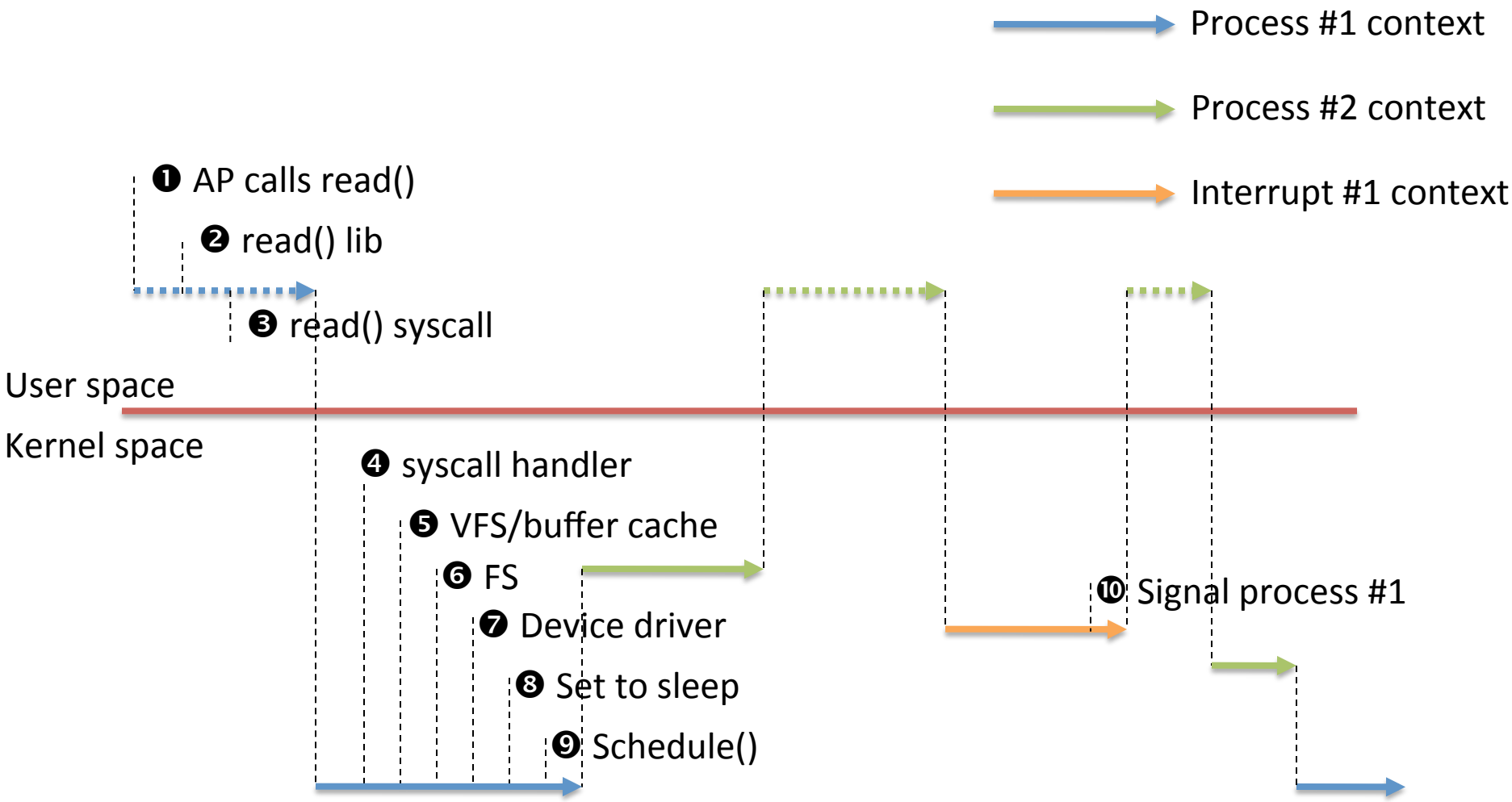
Interrupt handler returns

CPU runs interrupt handler (or called ISR) in the kernel space

Kernel is in interrupt context

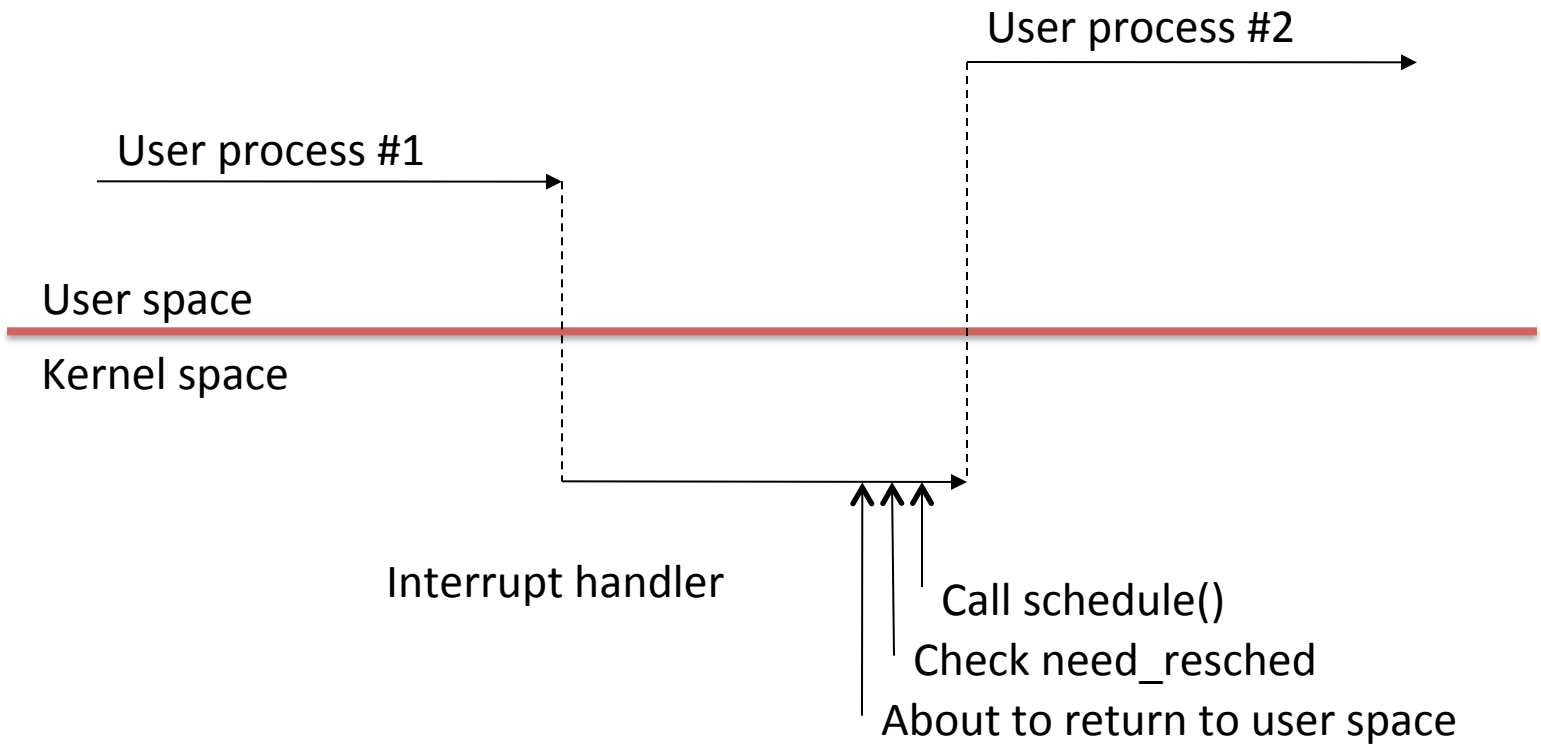


Process context + interrupt context



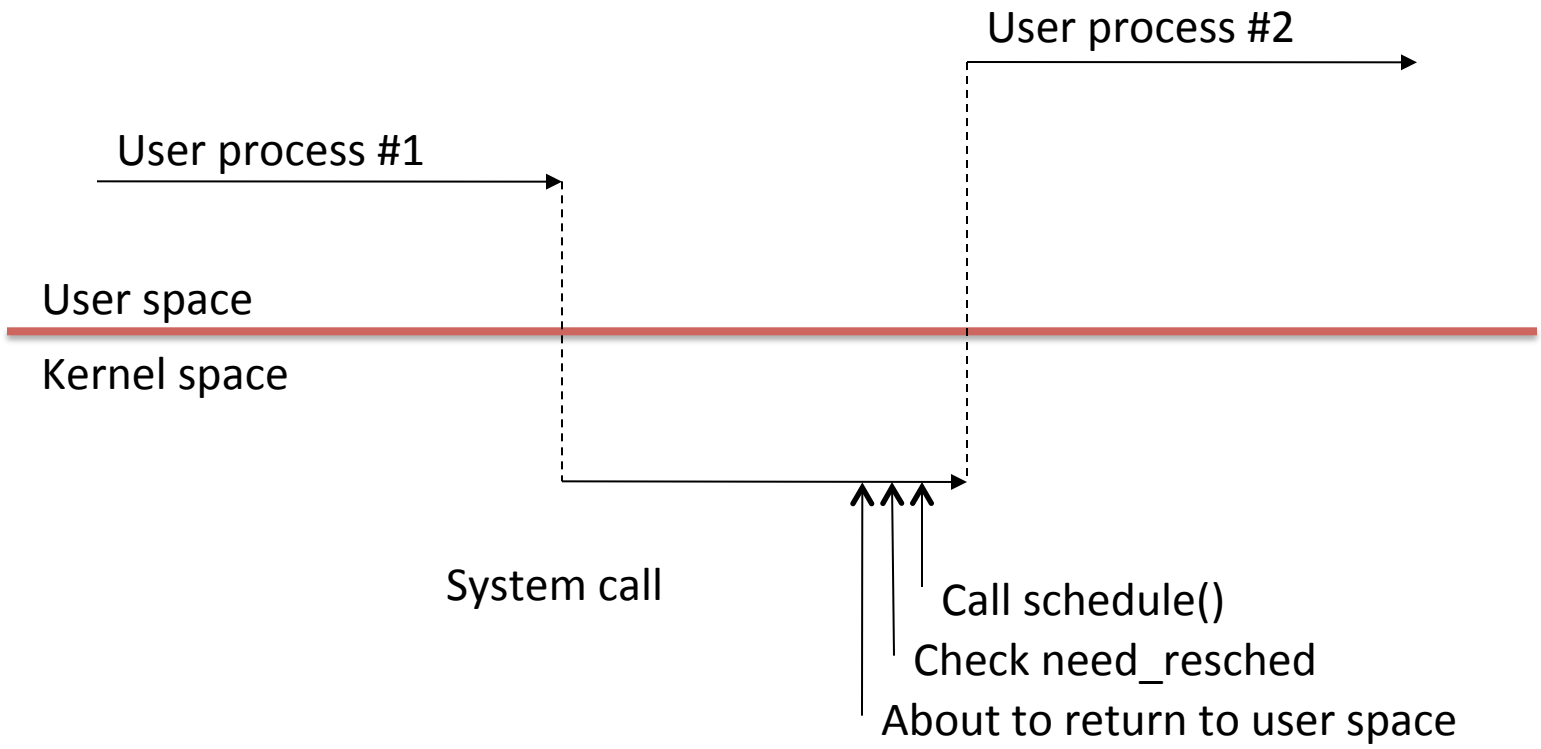
User preemption

- User preemption occurs when the kernel is in a safe state and about to return to user-space



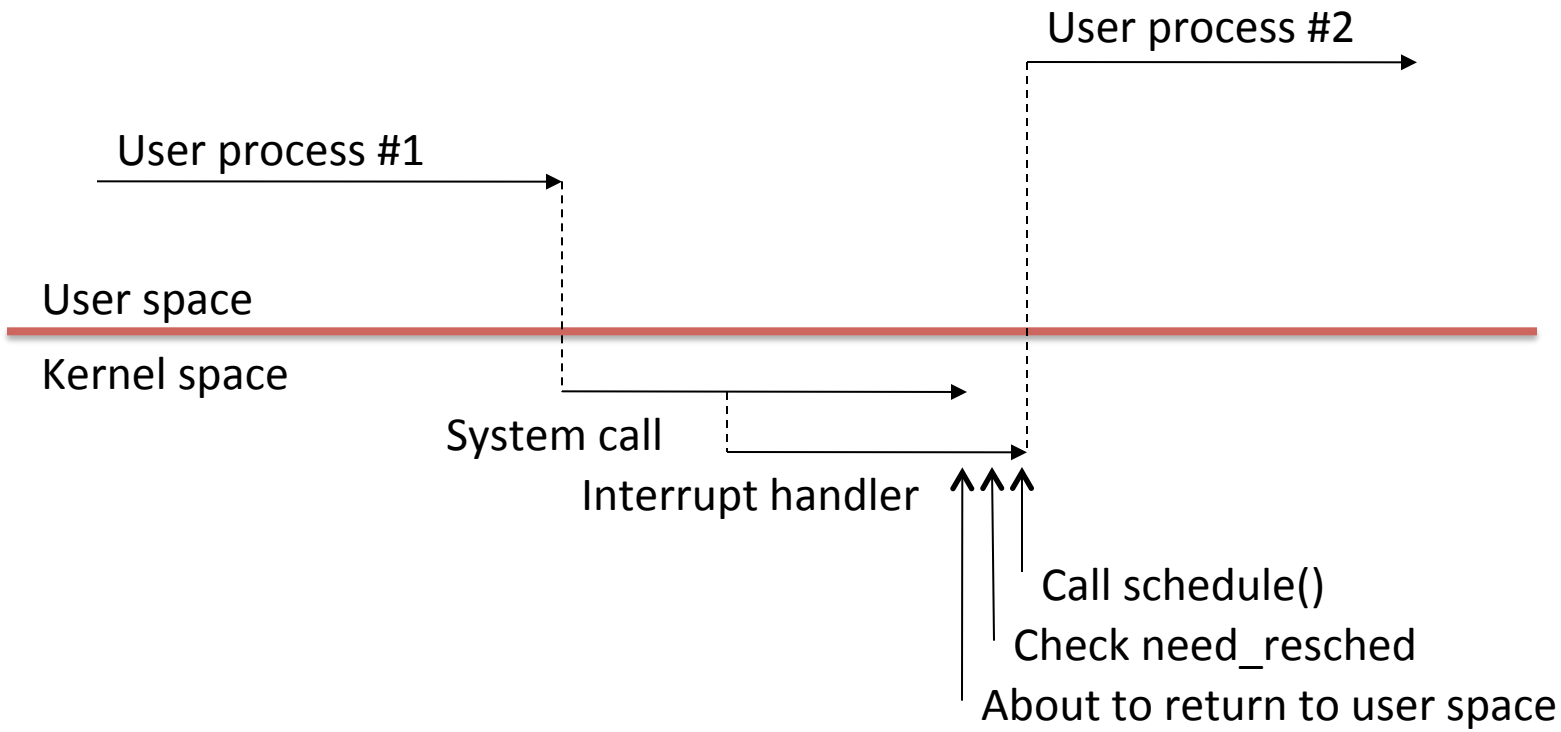
User preemption

- User preemption occurs when the kernel is in a safe state and about to return to user-space



Kernel preemption

- Linux kernel is possible to preempt a task at any point, so long as the kernel does not hold a lock



Preemptive Kernel

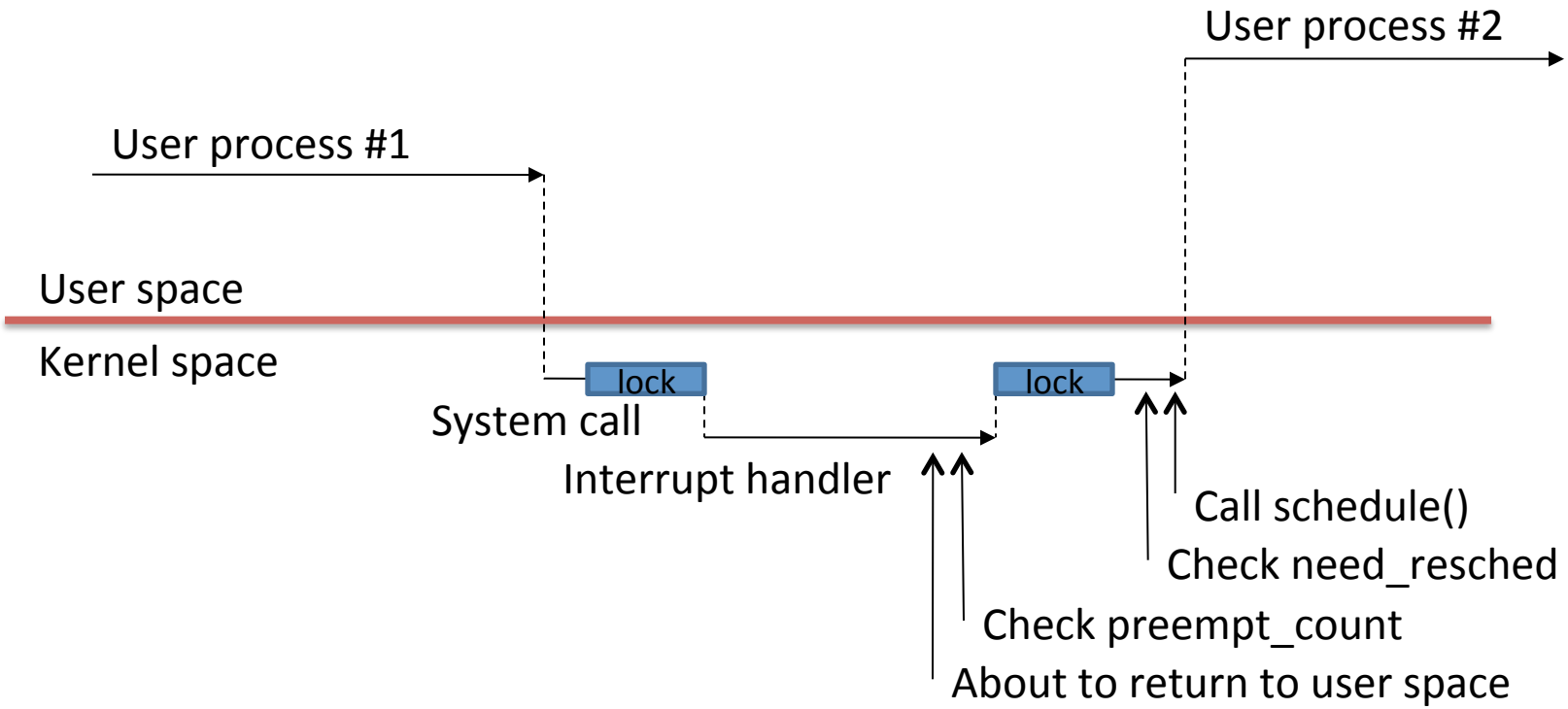
- Non-preemptive kernel supports user preemption
- Preemptive kernel supports kernel/user preemption

- Kernel can be interrupted ≠ kernel is preemptive
  - Non-preemptive kernel, interrupt returns to interrupted process
  - Preemptive kernel, interrupt returns to any schedulable process

Preemptive Kernel

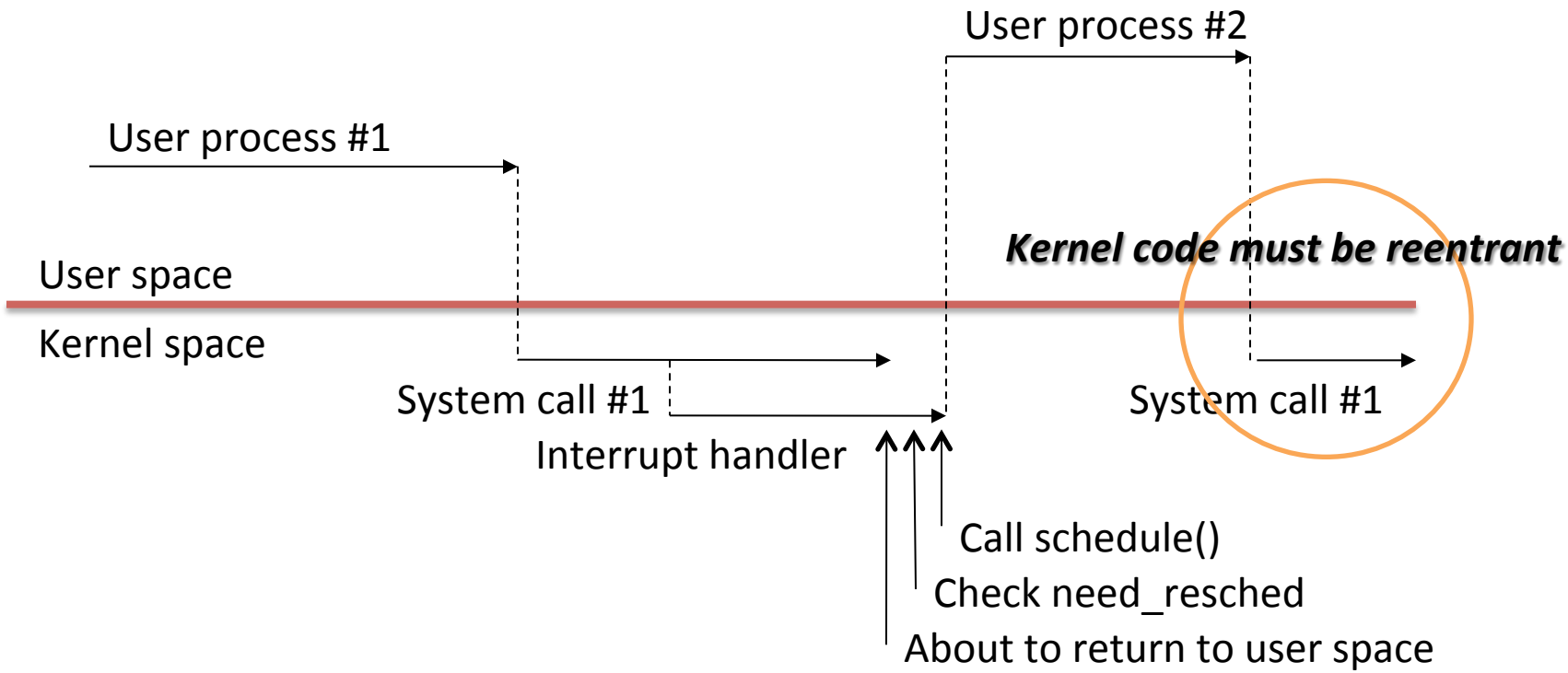
- 2.4 is a non-preemptive kernel
- 2.6 is a preemptive kernel
- 2.6 could disable CONFIG_PREEMPT

Preemptive Kernel

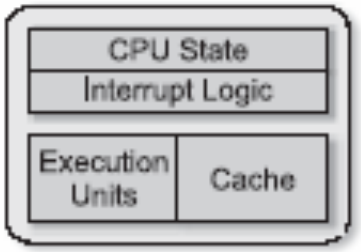


Preemptive Kernel

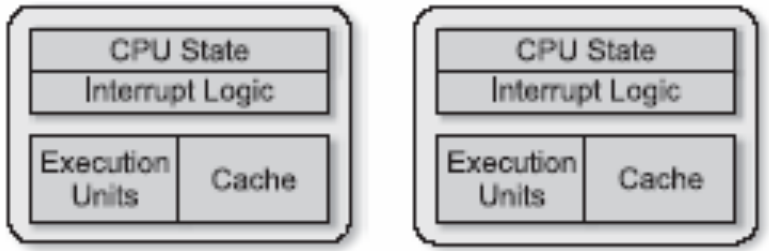
- How difficult to implement a preemptive kernel?



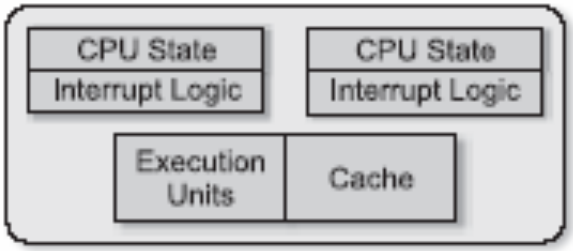
Single Core vs. Multi-core



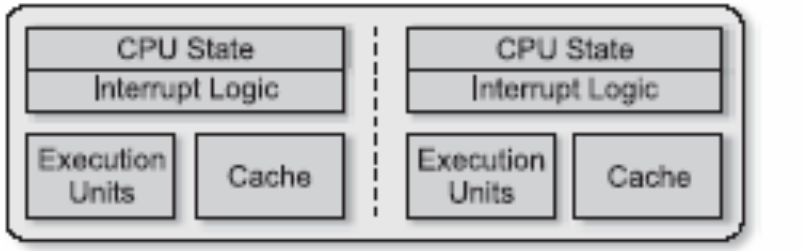
Single processor/single core



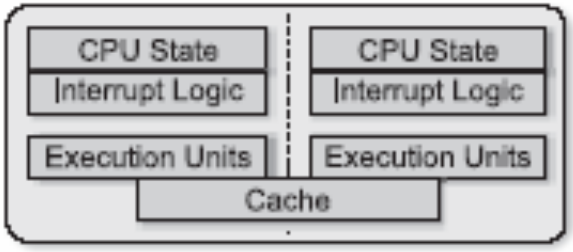
Multiple processor/single core



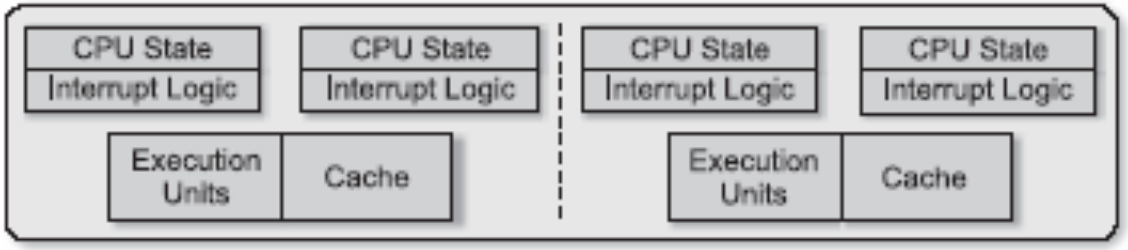
Single processor/single core/hyper threading



Single processor/multi core/separated cache

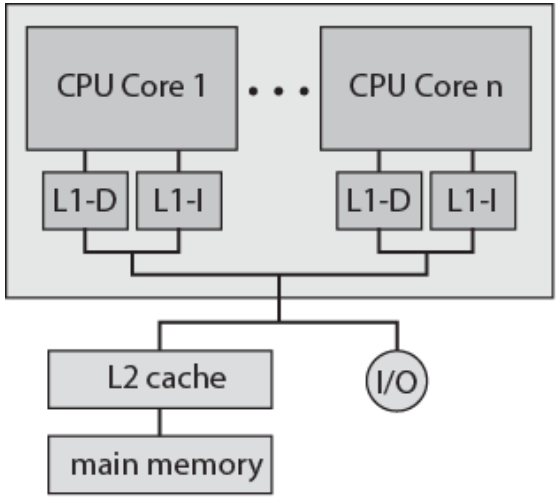


Single processor/multi core/shared cache

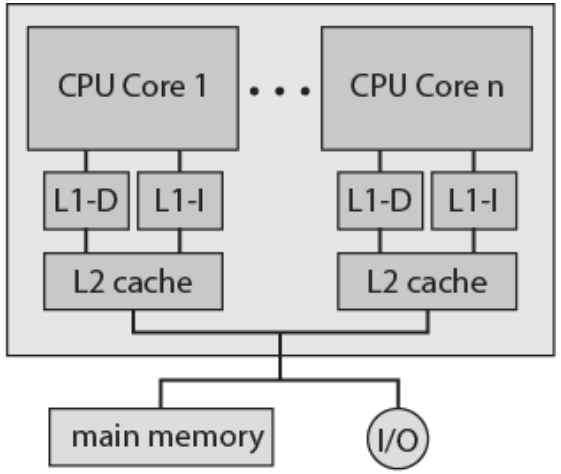


Single processor/multi core/hyper threading

ARM11 MPCore



(a) Dedicated L1 cache

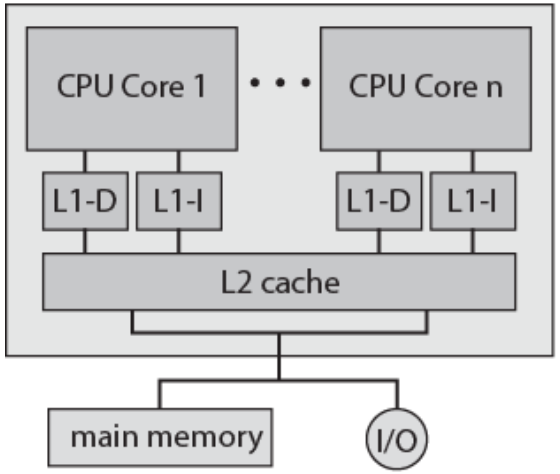


(b) Dedicated L2 cache

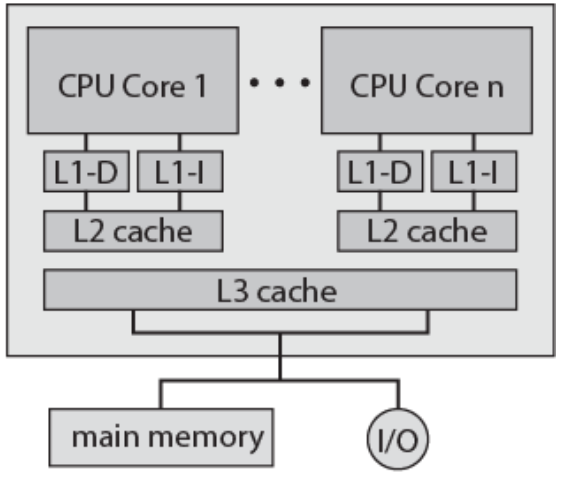
AMD Opteron

No shared

Intel Core Duo



(c) Shared L2 cache

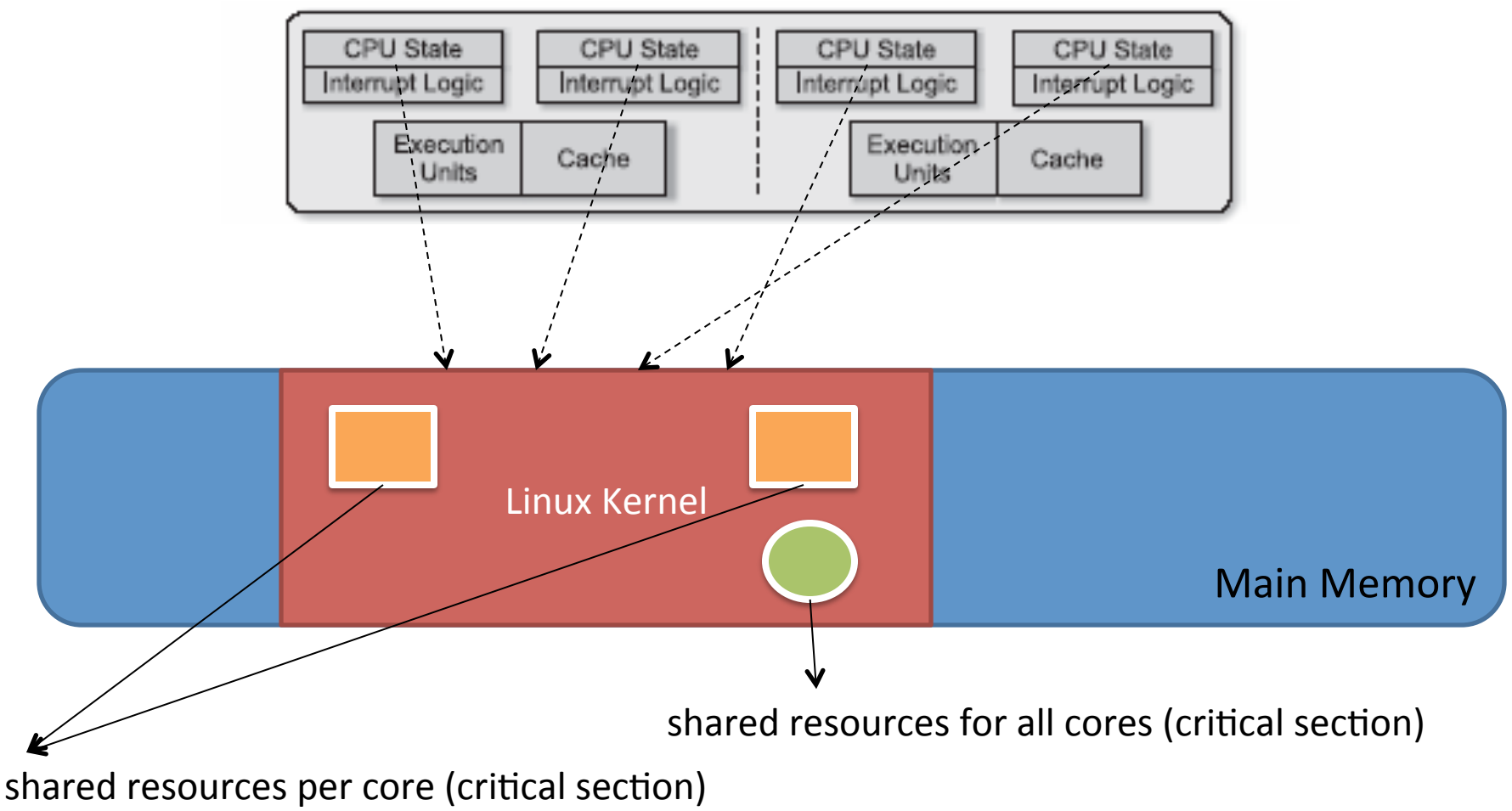


(d) Shared L3 cache

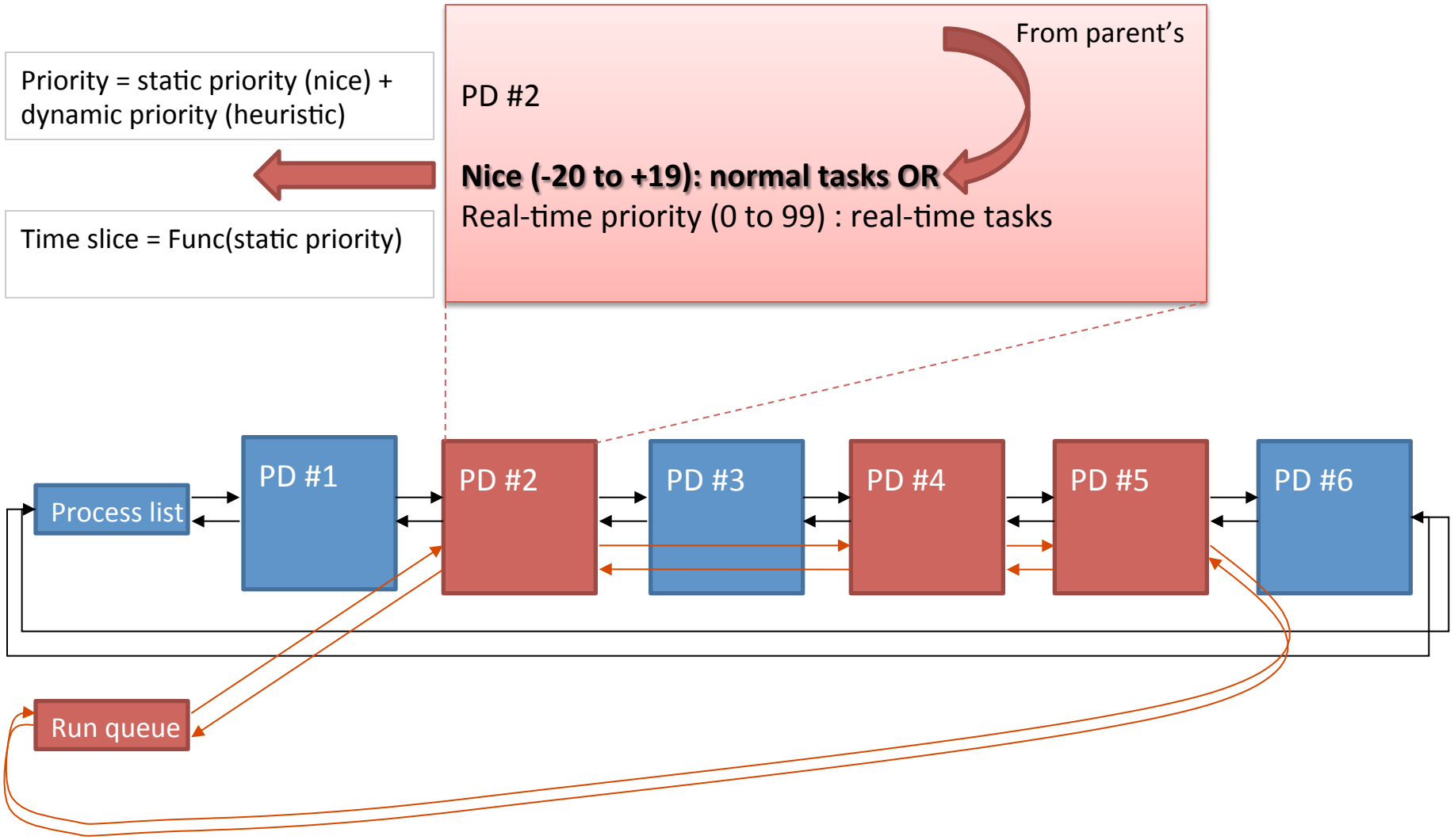
Intel Core i7

Shared

Single Core vs. Multi-core



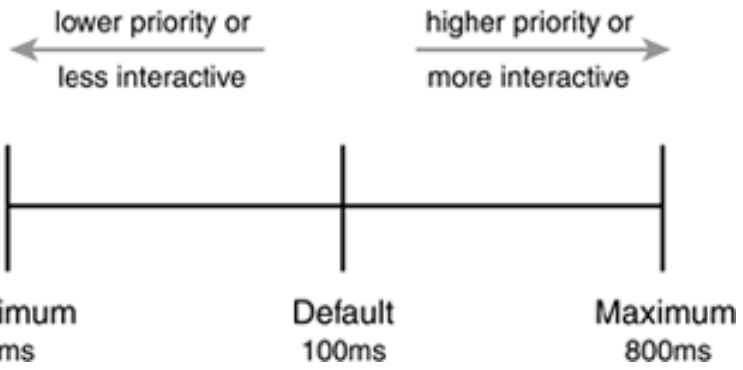
How Linux Scheduler Works



Timeslice

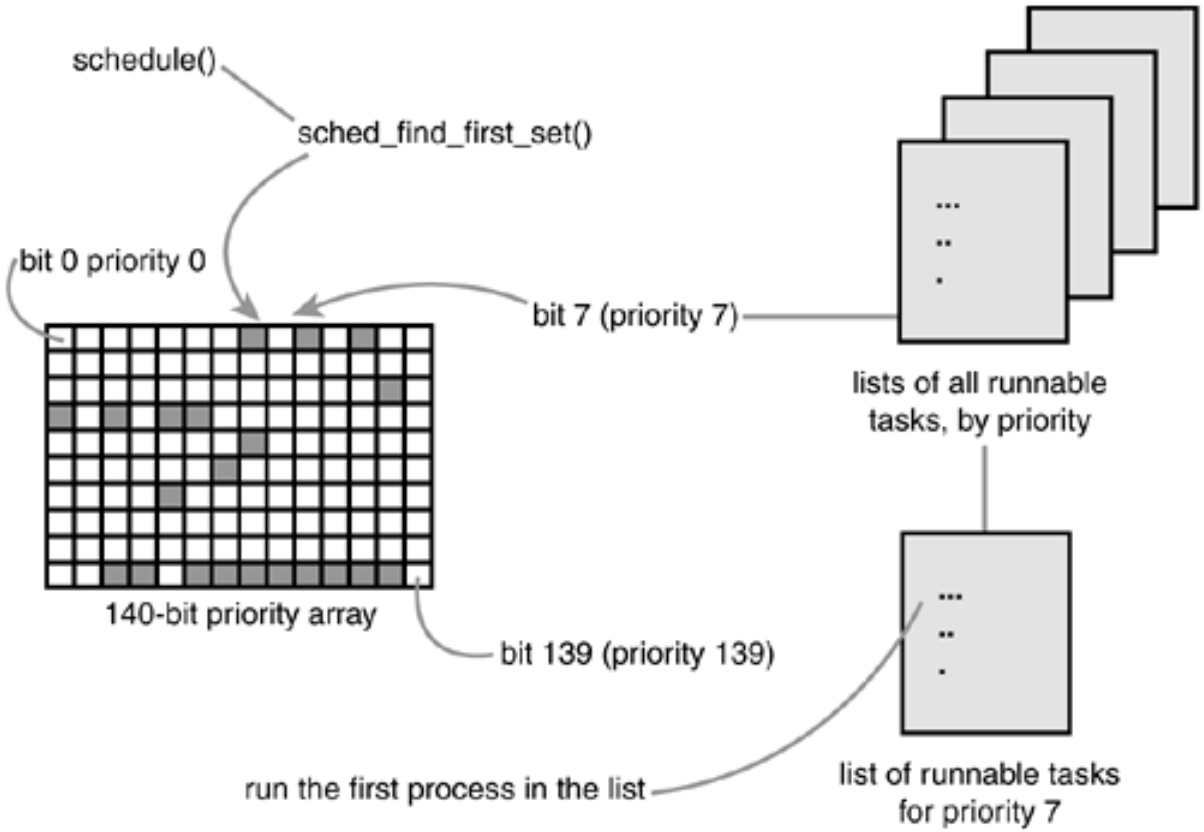
- Timeslice function

Type of Task	Nice Value	Timeslice Duration
Initially created	parent's	half of parent's
Minimum Priority	+19	5ms (MIN_TIMESLICE)
Default Priority	0	100ms (DEF_TIMESLICE)
Maximum Priority	20	800ms (MAX_TIMESLICE)

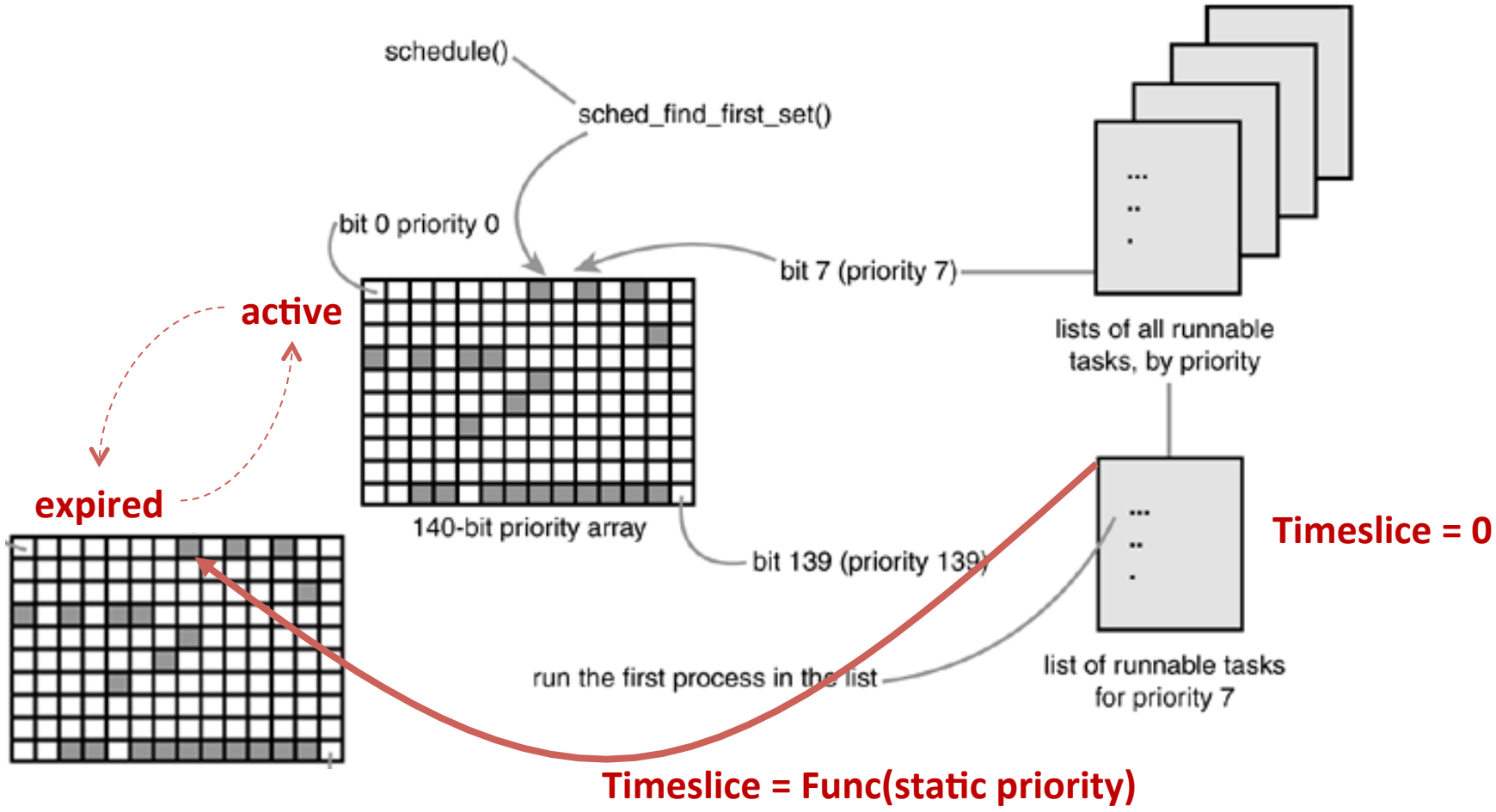


$$\text{base time quantum (in milliseconds)} = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases} \quad (1)$$

Linux O(1) cheduler



Recalculating timeslice



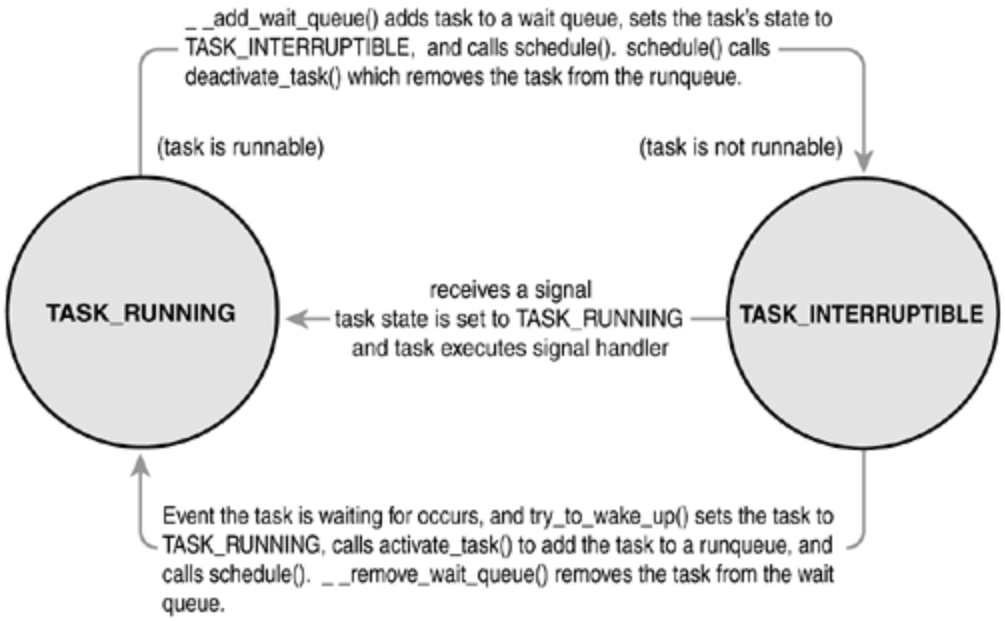
Calculating Priority

- static_prio = nice
- Prio = nice – bonus + 5

  dynamic priority = max (100, min (static priority - bonus + 5, 139))

- Heuristic
  - sleep_avg: (0 to MAX_SLEEP_AVG(10ms))
  - sleep_avg+=sleep (becomes runnable)
  - Sleep_avg-=run (every time tick when task runs)

Sleeping and waking up

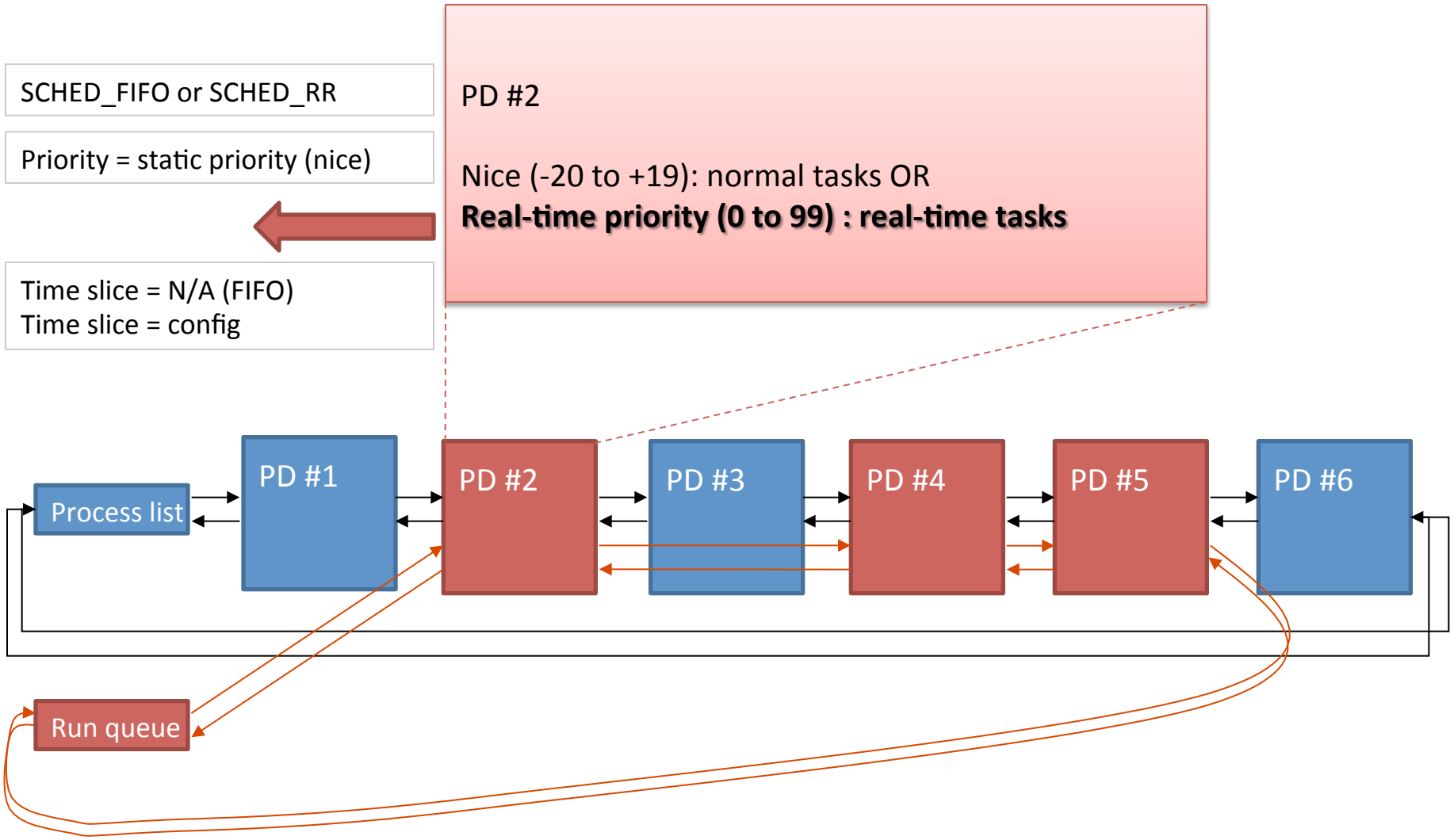



```
add_wait_queue(q, &wait);
while (!condition) {     /* condition is the event that we are waiting for */
        set_current_state(TASK_INTERRUPTIBLE); /* or TASK_UNINTERRUPTIBLE */
        if (signal_pending(current))
                /* handle signal */
        schedule();
}
set_current_state(TASK_RUNNING);
remove_wait_queue(q, &wait);
```

System calls related to scheduling

System call	Description
nice( )	Change the static priority of a conventional process
getpriority( )	Get the maximum static priority of a group of conventional processes
setpriority( )	Set the static priority of a group of conventional processes
sched_getscheduler( )	Get the scheduling policy of a process
sched_setscheduler( )	Set the scheduling policy and the real-time priority of a process
sched_getparam( )	Get the real-time priority of a process
sched_setparam( )	Set the real-time priority of a process
sched_yield( )	Relinquish the processor voluntarily without blocking
sched_get_ priority_min( )	Get the minimum real-time priority value for a policy
sched_get_ priority_max( )	Get the maximum real-time priority value for a policy
sched_rr_get_interval( )	Get the time quantum value for the Round Robin policy
sched_setaffinity( )	Set the CPU affinity mask of a process
sched_getaffinity( )	Get the CPU affinity mask of a process

How Linux Scheduler Works



Process schedule and context switching in Linux

- Context switch
  - Hardware context switch
    - Task State Segment Descriptor (Old Linux)
  - Step by step context switch
    - Better control and optimize
- Context switch
  - switch_mm()
    - Switch virtual memory mapping
  - switch_to()
    - Switch processor state
- Process switching occurs only in kernel mode
- The contents of all registers used by a process in User Mode have already been saved

Kernel Synchronization

Why we need kernel synchronization?

- Protect critical section
- Code paths that access and manipulate shared data are called critical regions
- Data
  - Shared by processes
  - Shared by processors

Example

- i: shared data
- i++

Thread 1	Thread 2
get i (7)	-
increment i (7 -> 8)	-
write back i (8)	-
-	get i (8)
-	increment i (8 -> 9)
-	write back i (9)

Thread 1	Thread 2
get i (7)	get i (7)
increment i (7 -> 8)	-
-	increment i (7 -> 8)
write back i (8)	-
-	write back i (8)

Solution

- Atomic operations

Thread 1	**Thread 2**
`increment i (7 -> 8)`	-
-	`increment i (8 -> 9)`

Or:

Thread 1	**Thread 2**
-	`increment i (7 -> 8)`
`increment i (8 -> 9)`	-

How to protect critical section?

- Atomic Integer Operations

ATOMIC_INIT(int i)	At declaration, initialize an atomic_t to i
int atomic_read(atomic_t *v)	Atomically read the integer value of v
void atomic_set(atomic_t *v, int i)	Atomically set v equal to i
void atomic_add(int i, atomic_t *v)	Atomically add i to v
void atomic_sub(int i, atomic_t *v)	Atomically subtract i from v
void atomic_inc(atomic_t *v)	Atomically add one to v
void atomic_dec(atomic_t *v)	Atomically subtract one from v
int atomic_sub_and_test(int i, atomic_t *v)	Atomically subtract i from v and return true if the result is zero; otherwise false
int atomic_add_negative(int i, atomic_t *v)	Atomically add i to v and return true if the result is negative; otherwise false
int atomic_dec_and_test(atomic_t *v)	Atomically decrement v by one and return true if zero; false otherwise
int atomic_inc_and_test(atomic_t *v)	Atomically increment v by one and return true if the result is zero; false otherwise

Atomic operation

- Atomic Bitwise Operations

void set_bit(int nr, void *addr)	Atomically set the nr-th bit starting from addr
void clear_bit(int nr, void *addr)	Atomically clear the nr-th bit starting from addr
void change_bit(int nr, void *addr)	Atomically flip the value of the nr-th bit starting from addr
int test_and_set_bit(int nr, void *addr)	Atomically set the nr-th bit starting from addr and return the previous value
int test_and_clear_bit(int nr, void *addr)	Atomically clear the nr-th bit starting from addr and return the previous value
int test_and_change_bit(int nr, void *addr)	Atomically flip the nr-th bit starting from addr and return the previous value
int test_bit(int nr, void *addr)	Atomically return the value of the nr-th bit starting from addr

Spin lock

- spin_lock
  - Normally, critical section is more than just a variable, spin lock protects a sequence of codes manipulating the critical section
  - Perform busy loops spins for waiting the lock to become available
  - Spin_lock disables the kernel preemption

Spin lock

- Principles for using spin lock
  - Lock the data, not the code
  - Don't hold a spin lock for a long time (because someone may wait outside)
  - Don't recursively spin lock in Linux (like you lock your key in your car)
  - Spin lock (instead of using mutex) if you can finish within 2 context switches time
  - In a preemptive kernel, spin lock will perform preempt_disable( )
  - Can be used in interrupt handler (mutex/semaphores cannot) *must disable local interrupt*

Spin lock

- When you use spin_lock(x) in the kernel, make sure you don't spend too much time in the lock, and there is no recursive spin_lock(x)
  - No spin_lock(x) in the spin_lock(x) period
  - No spin_lock(x) in interrupt handler
- When you use spin_lock(x) in interrupt handler, make sure the interrupt has been disabled. Otherwise, use spin_lock_irq(x) or spin_lock_irqsave(x)

Spin lock irq

- Disables interrupts and acquires the lock
- spin_unlock_irq() unlocks the given lock
- Use of spin_lock_irq() therefore is not recommended

Spin lock irqsave

- Disables interrupt if it is not disabled and acquires the lock
- spin_unlock_irqrestore () unlocks the given lock and returns interrupts to their previous state

Spin lock methods

spin_lock()	Acquires given lock
spin_lock_irq()	Disables local interrupts and acquires given lock
spin_lock_irqsave()	Saves current state of local interrupts, disables local interrupts, and acquires given lock
spin_unlock()	Releases given lock
spin_unlock_irq()	Releases given lock and enables local interrupts
spin_unlock_irqrestore()	Releases given lock and restores local interrupts to given previous state
spin_lock_init()	Dynamically initializes given spinlock_t
spin_trylock()	Tries to acquire given lock; if unavailable, returns nonzero
spin_is_locked()	Returns nonzero if the given lock is currently acquired, otherwise it returns zero

Read write spin lock

- Lock usage can be clearly divided into readers and writers (e.g. search and update)
- Linux task list is protected by a reader-writer spin lock

```
read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);
```

```
write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_lock);
```

- it
  - You cannot do this

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

Read write spin lock

Method	Description
read_lock()	Acquires given lock for reading
read_lock_irq()	Disables local interrupts and acquires given lock for reading
read_lock_irqsave()	Saves the current state of local interrupts, disables local interrupts, and acquires the given lock for reading
read_unlock()	Releases given lock for reading
read_unlock_irq()	Releases given lock and enables local interrupts
read_unlock_irqrestore()	Releases given lock and restores local interrupts to the given previous state
write_lock()	Acquires given lock for writing
write_lock_irq()	Disables local interrupts and acquires the given lock for writing
write_lock_irqsave()	Saves current state of local interrupts, disables local interrupts, and acquires the given lock for writing
write_unlock()	Releases given lock
write_unlock_irq()	Releases given lock and enables local interrupts
write_unlock_irqrestore()	Releases given lock and restores local interrupts to given previous state
write_trylock()	Tries to acquire given lock for writing; if unavailable, returns nonzero
rw_lock_init()	Initializes given rwlock_t
rw_is_locked()	Returns nonzero if the given lock is currently acquired, or else it returns zero

Mutex/Semaphores

- sleeping locks
- do not disable kernel preemption
- the mutex/semaphore place the task onto a wait queue and put the task to sleep
- well suited to locks that are held for a long time
- not optimal for locks that are held for very short periods
- can be obtained only in process context, not interrupt context
- can sleep while holding a semaphore
- cannot hold a spin lock while you acquire a semaphore

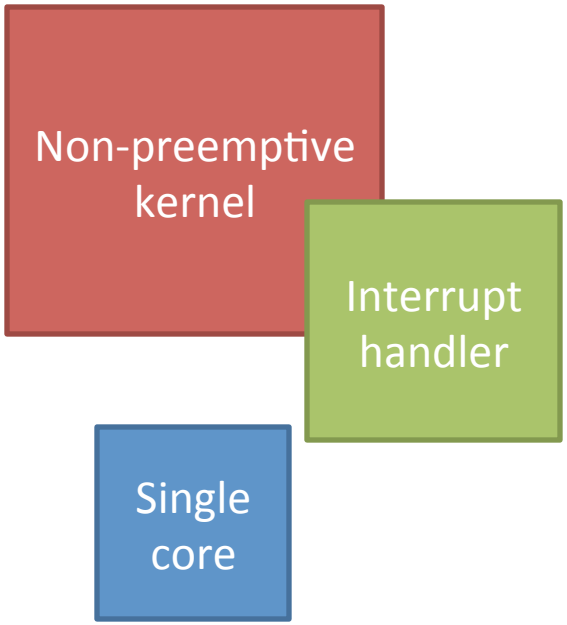
Mutex/Semaphores

- binary semaphore vs. counting semaphore (*not used much in the kernel*)
- down_interruptible()
  - attempts to acquire the given semaphore. If it fails, it sleeps in the TASK_INTERRUPTIBLE
- down()
  - If it fails, places the task in the TASK_UNINTERRUPTIBLE state if it sleeps (*normally not used*)

Mutex/Semaphores

sema_init(struct semaphore *, int)	Initializes the dynamically created semaphore to the given count
init_MUTEX(struct semaphore *)	Initializes the dynamically created semaphore with a count of one
init_MUTEX_LOCKED(struct semaphore *)	Initializes the dynamically created semaphore with a count of zero (so it is initially locked)
down_interruptible(struct semaphore *)	Tries to acquire the given semaphore and enter interruptible sleep if it is contended
down(struct semaphore *)	Tries to acquire the given semaphore and enter uninterruptible sleep if it is contended
down_trylock(struct semaphore *)	Tries to acquire the given semaphore and immediately return nonzero if it is contended
up(struct semaphore *)	Releases the given semaphore and wakes a waiting task, if any

Single Core + Non-preemptive kernel



local_irq_disable disables the kernel preemption

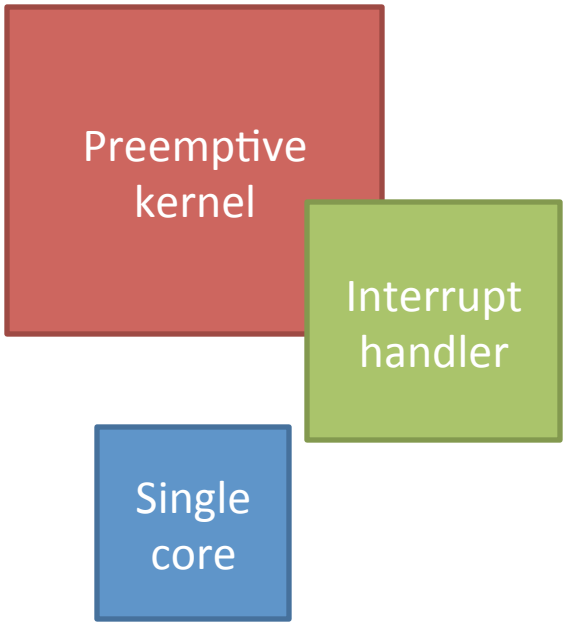
Non-preemptive kernel

Sync methods	Notes
local_irq_disable	warning, enable irq while leaving CS
local_irq_save	***OK, short period lock***
spin_lock	warning, if spin lock in interrupt handler
spin_lock_irq	warning, enable irq while leaving CS
spin_lock_irqsave	OK, short period lock
Semaphore/mutex	***OK, long period lock***

Interrupt handler

Sync methods	Notes
local_irq_disable	warning, enable irq while leaving CS
local_irq_save	warning, user process may hold CS
spin_lock	***OK***
spin_lock_irq	warning, enable irq while leaving CS
spin_lock_irqsave	OK
Semaphore/mutex	No

Single Core + Preemptive kernel



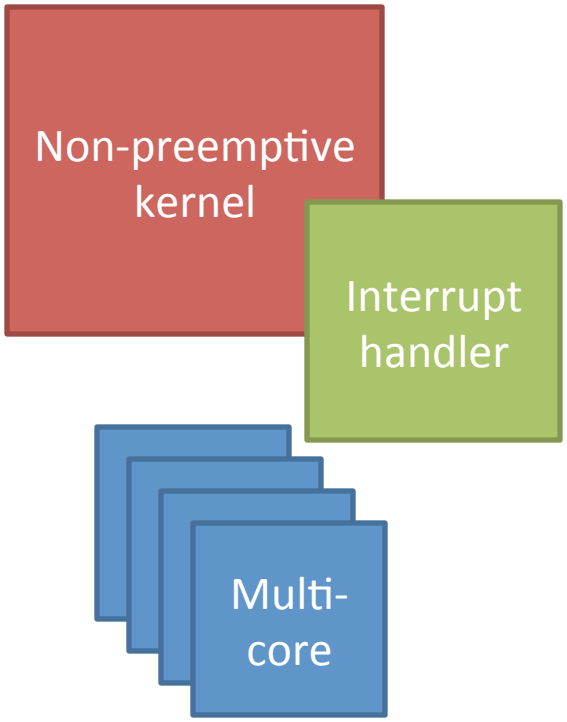
Preemptive kernel

Sync methods	Notes
local_irq_disable	warning, may schedule another thread
local_irq_save	warning, may schedule another thread
spin_lock	warning, if spin lock in interrupt handler
spin_lock_irq	warning , enable irq while leaving CS
spin_lock_irqsave	***OK, short period lock***
Semaphore/mutex	***OK, long period lock***

Interrupt handler

Sync methods	Notes
local_irq_disable	warning, enable irq while leaving CS
local_irq_save	warning, user process may hold CS
spin_lock	***OK***
spin_lock_irq	warning, enable irq while leaving CS
spin_lock_irqsave	OK
Semaphore/mutex	No

Multi-Core + Non-preemptive kernel



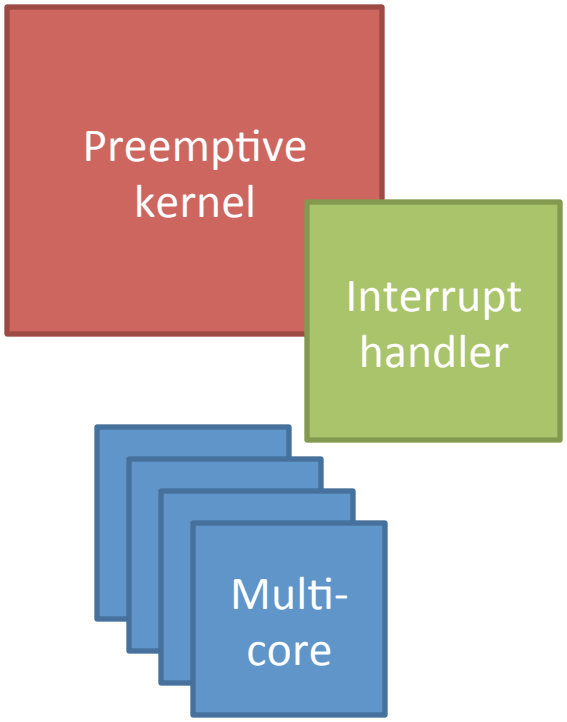
Non-preemptive kernel

Sync methods	Notes
local_irq_disable	warning, another CPU may enter
local_irq_save	warning, another CPU may enter
spin_lock	warning, if spin lock in interrupt handler
spin_lock_irq	warning, enable irq while leaving CS
spin_lock_irqsave	***OK, short period lock***
Semaphore/mutex	***OK, long period lock***

Interrupt handler

Sync methods	Notes
local_irq_disable	warning, another CPU may enter
local_irq_save	warning, another CPU may enter
spin_lock	***OK***
spin_lock_irq	warning , enable irq while leaving CS
spin_lock_irqsave	OK
Semaphore/mutex	No

Multi-Core + Preemptive kernel



Preemptive kernel

Sync methods	Notes
local_irq_disable	warning, another CPU may enter
local_irq_save	warning, another CPU may enter
spin_lock	warning, if spin lock in interrupt handler
spin_lock_irq	warning , enable irq while leaving CS
spin_lock_irqsave	***OK, short period lock***
Semaphore/mutex	***OK, long period lock***

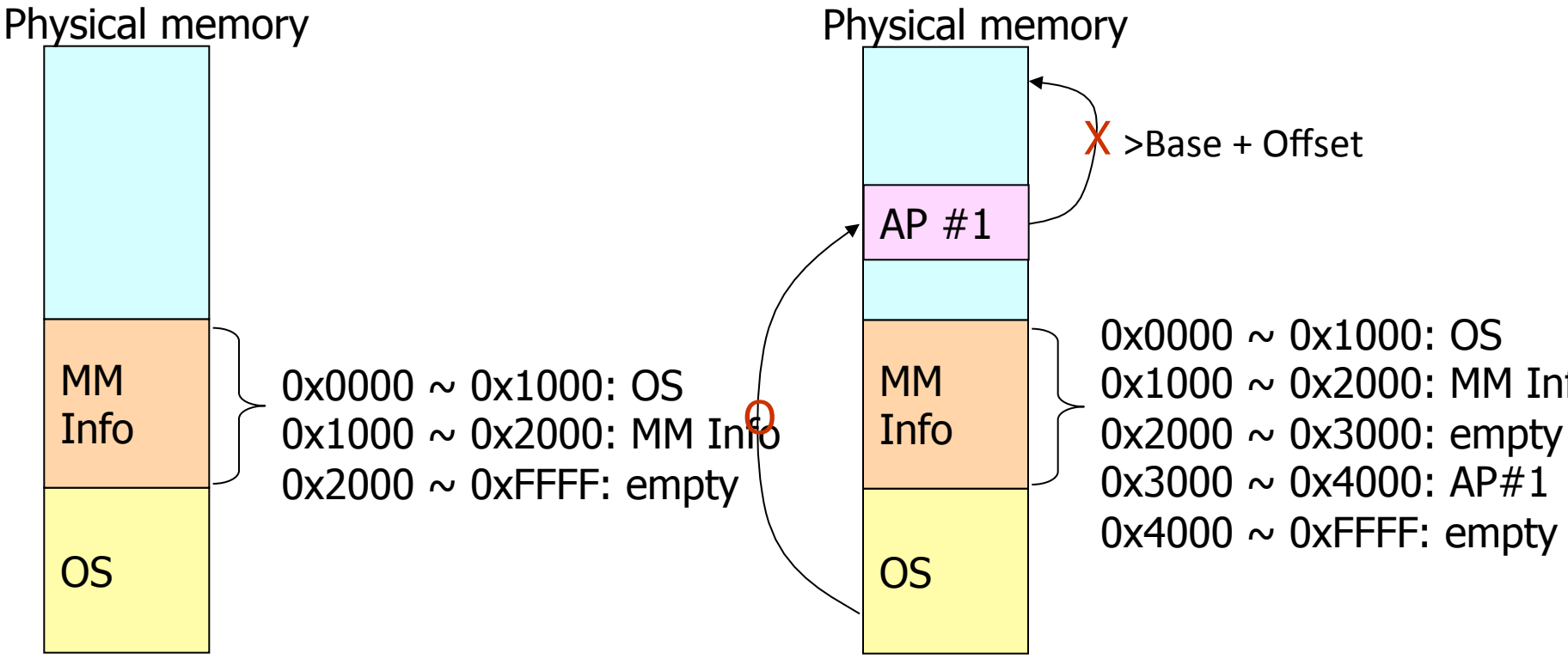
Interrupt handler

Sync methods	Notes
local_irq_disable	warning, another CPU may enter
local_irq_save	warning, another CPU may enter
spin_lock	***OK***
spin_lock_irq	warning, enable irq while leaving CS
spin_lock_irqsave	OK
Semaphore/mutex	No

Memory Management

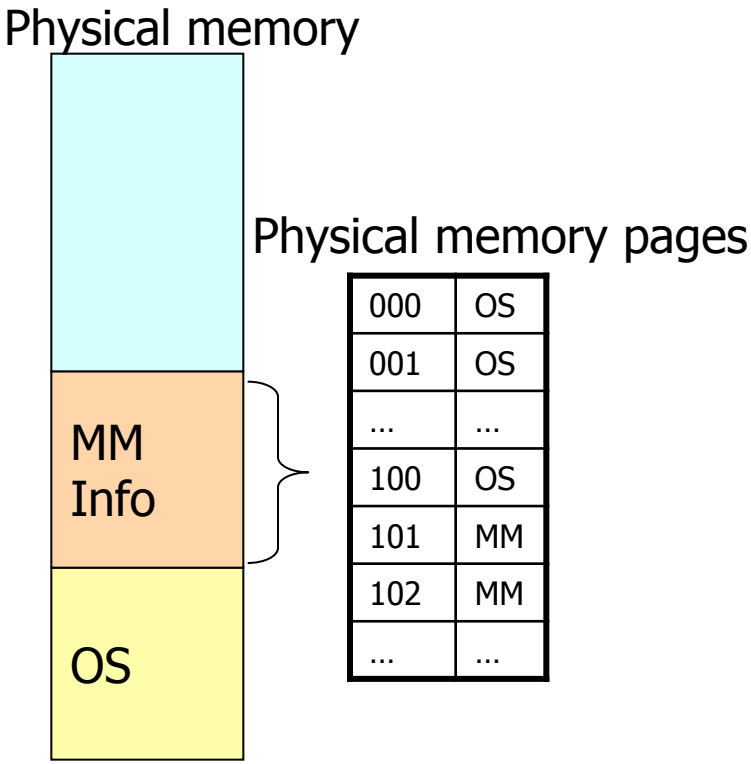
MM Basics

- Management physical memory



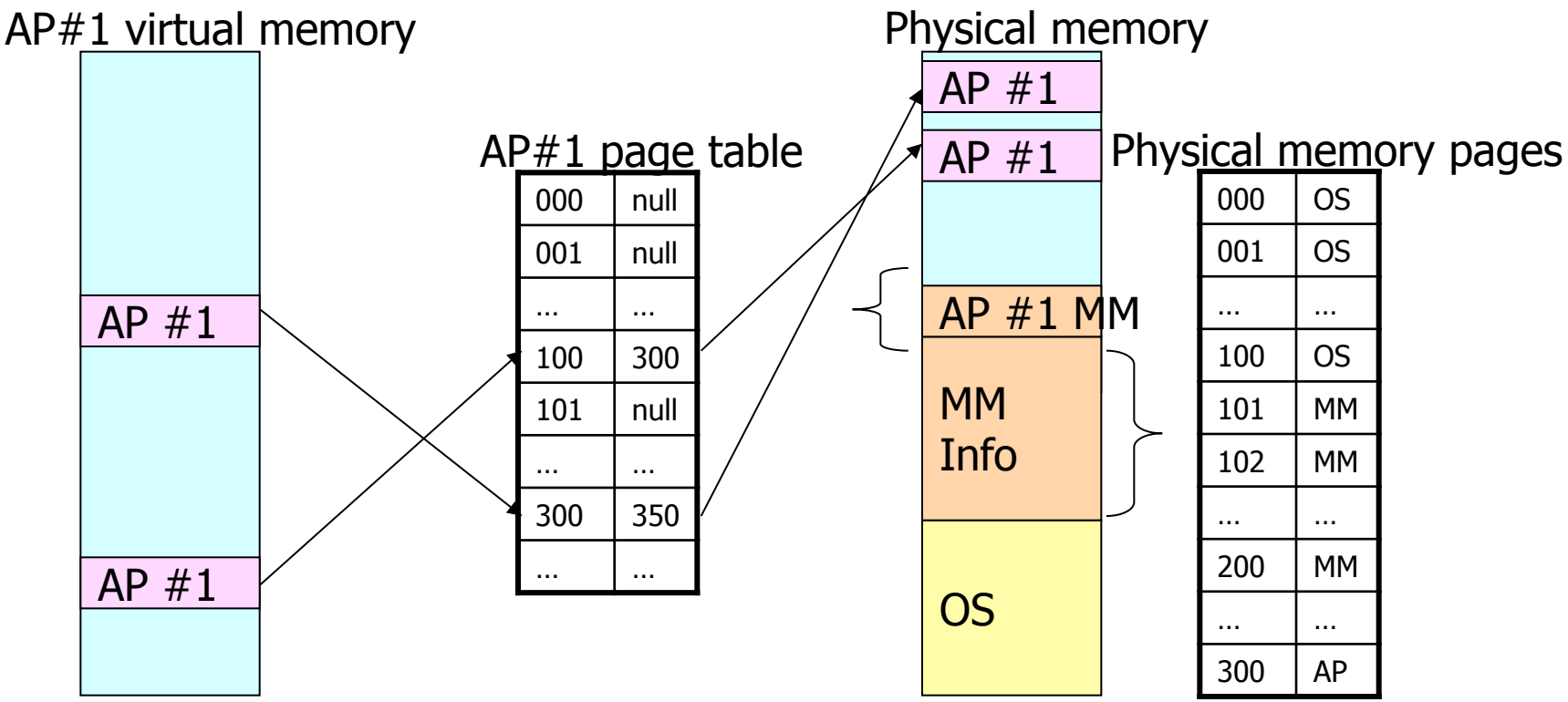
MM Basics (Cont.)

- Management virtual memory



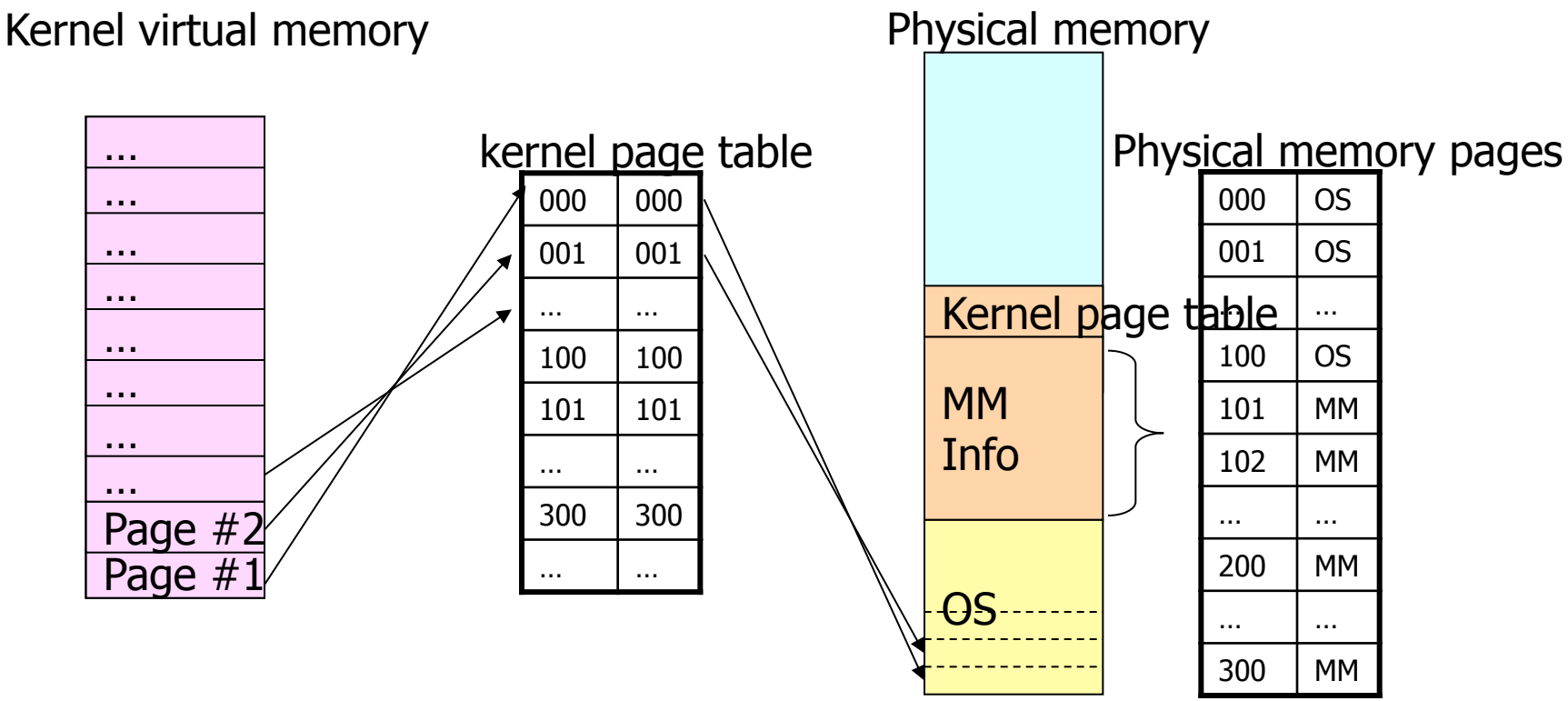
MM Basics (Cont.)

- Management physical memory (page)



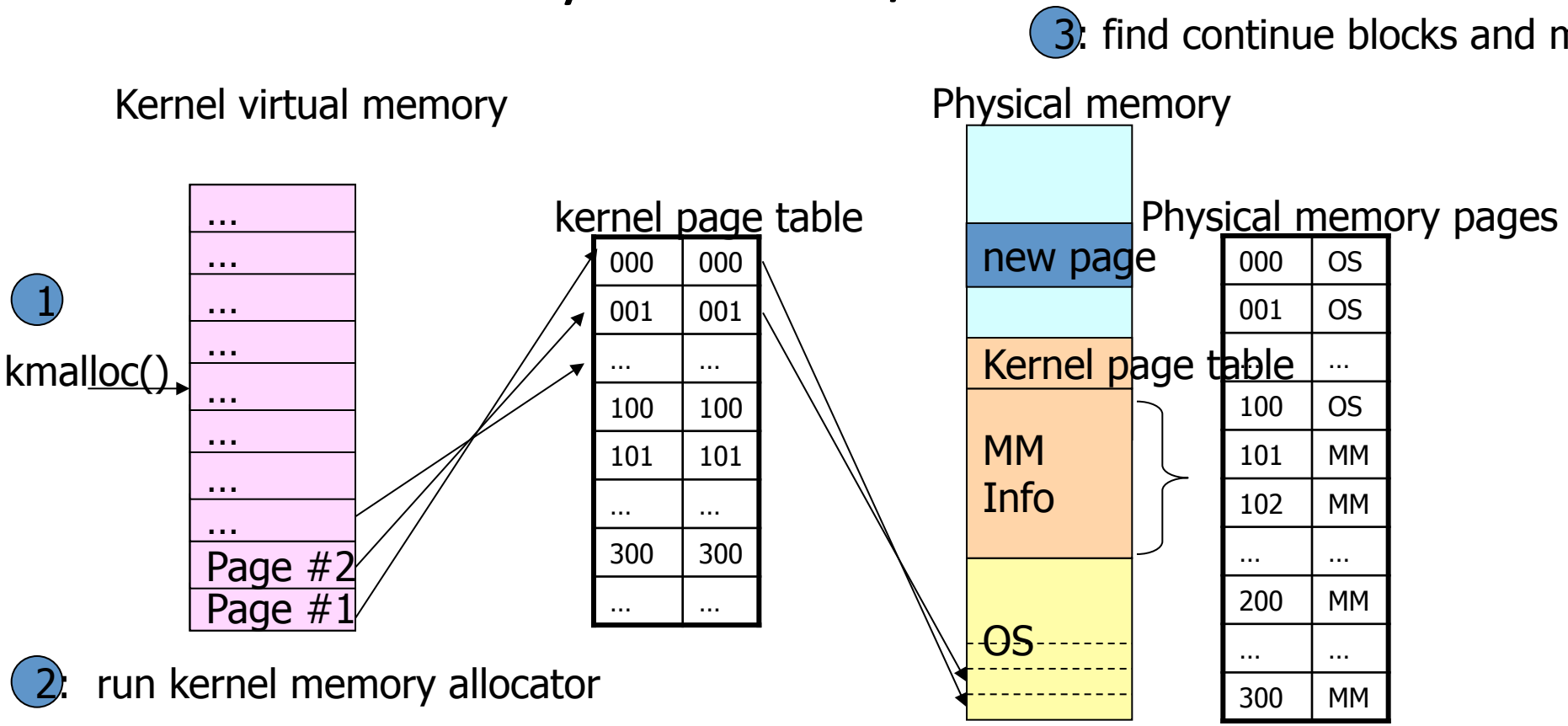
MM Basics (Cont.)

- Kernel memory allocation/release



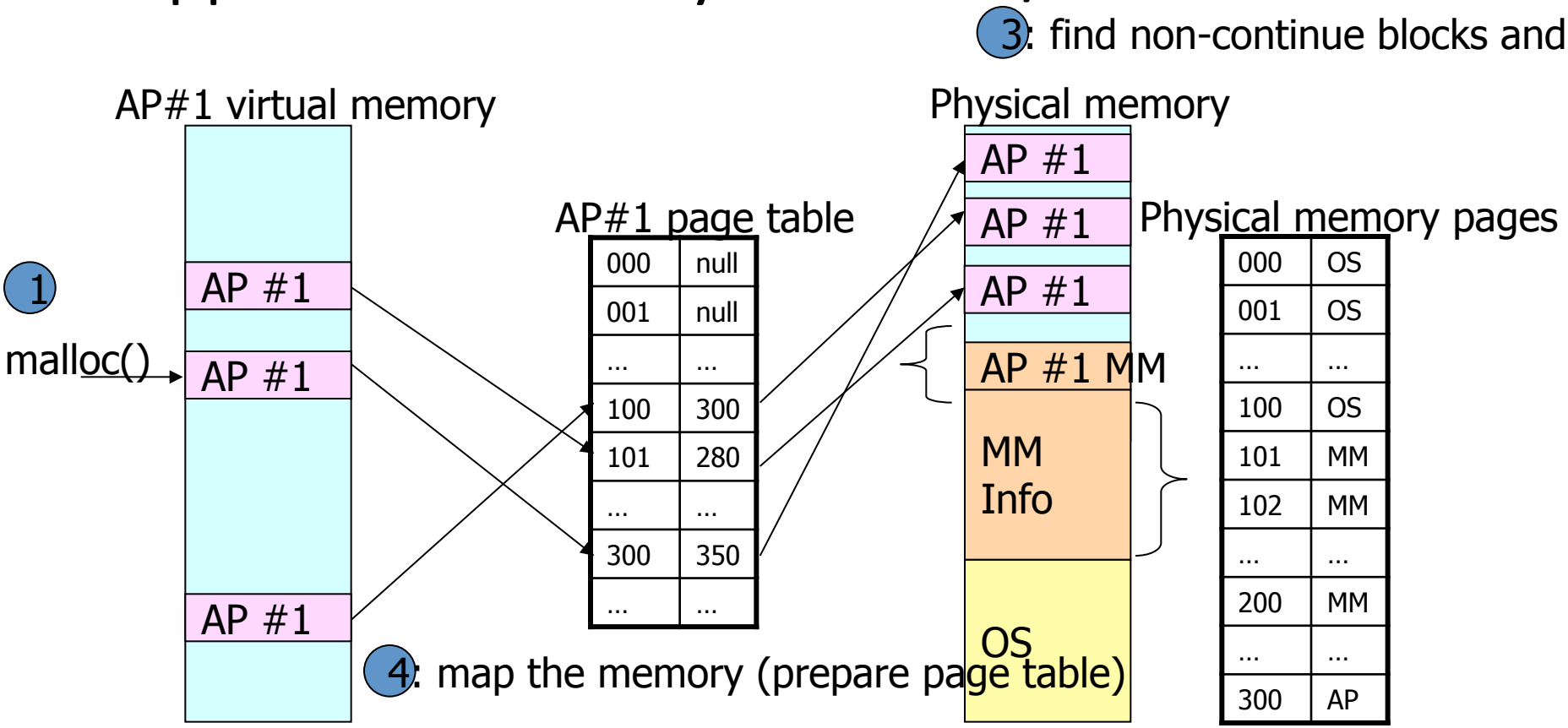
MM Basics (Cont.)

- Kernel memory allocation/release



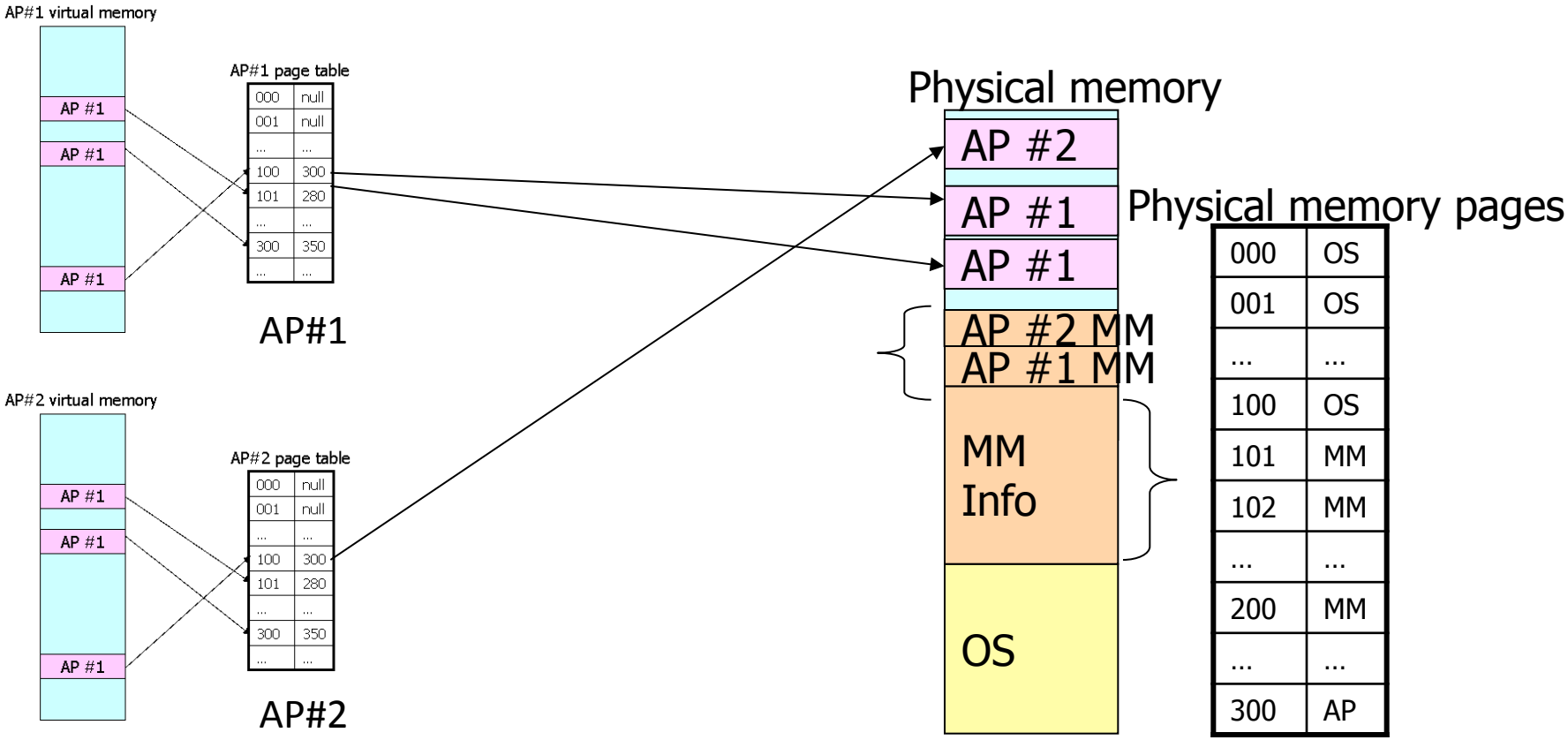
MM Basics (Cont.)

- application memory allocation/release

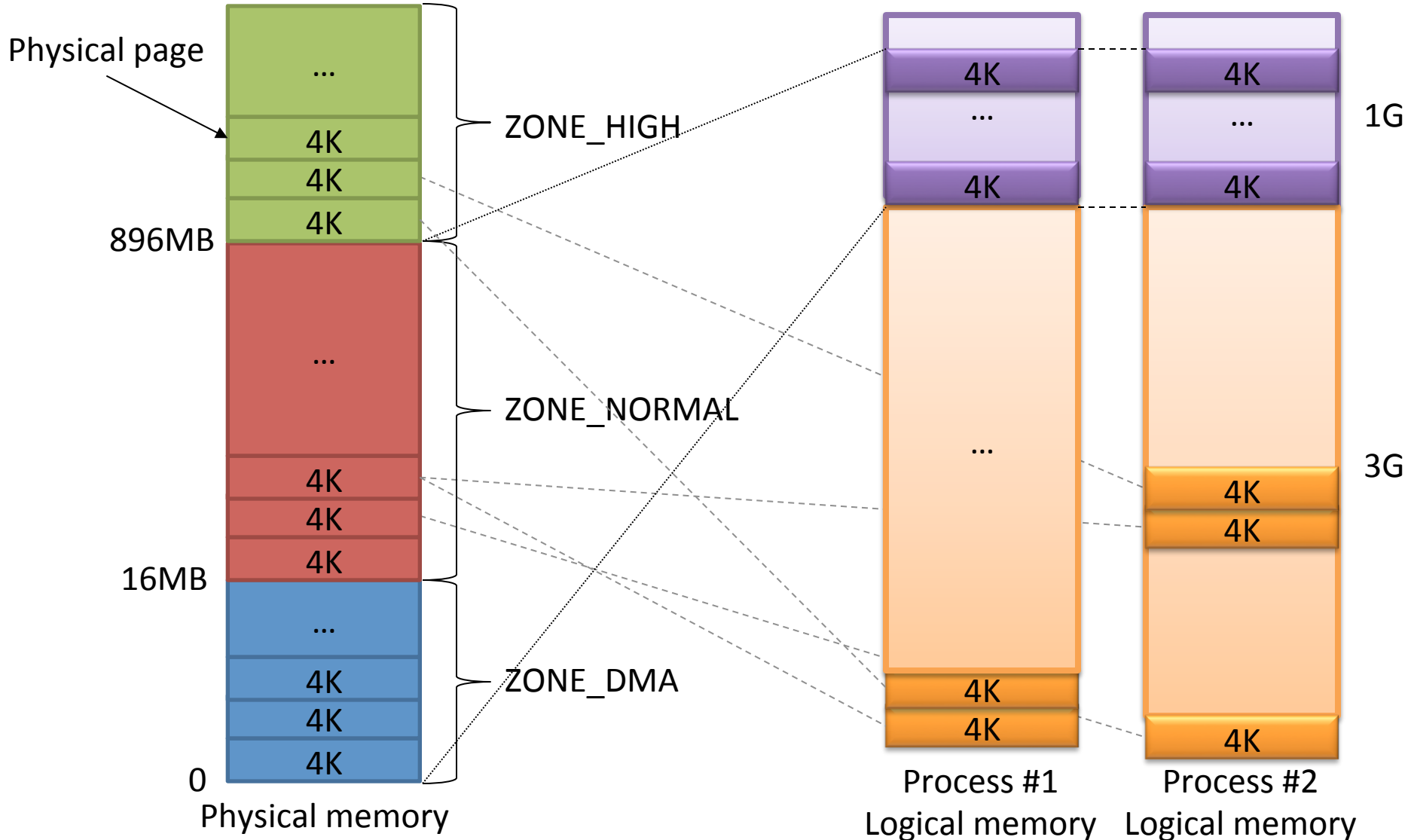


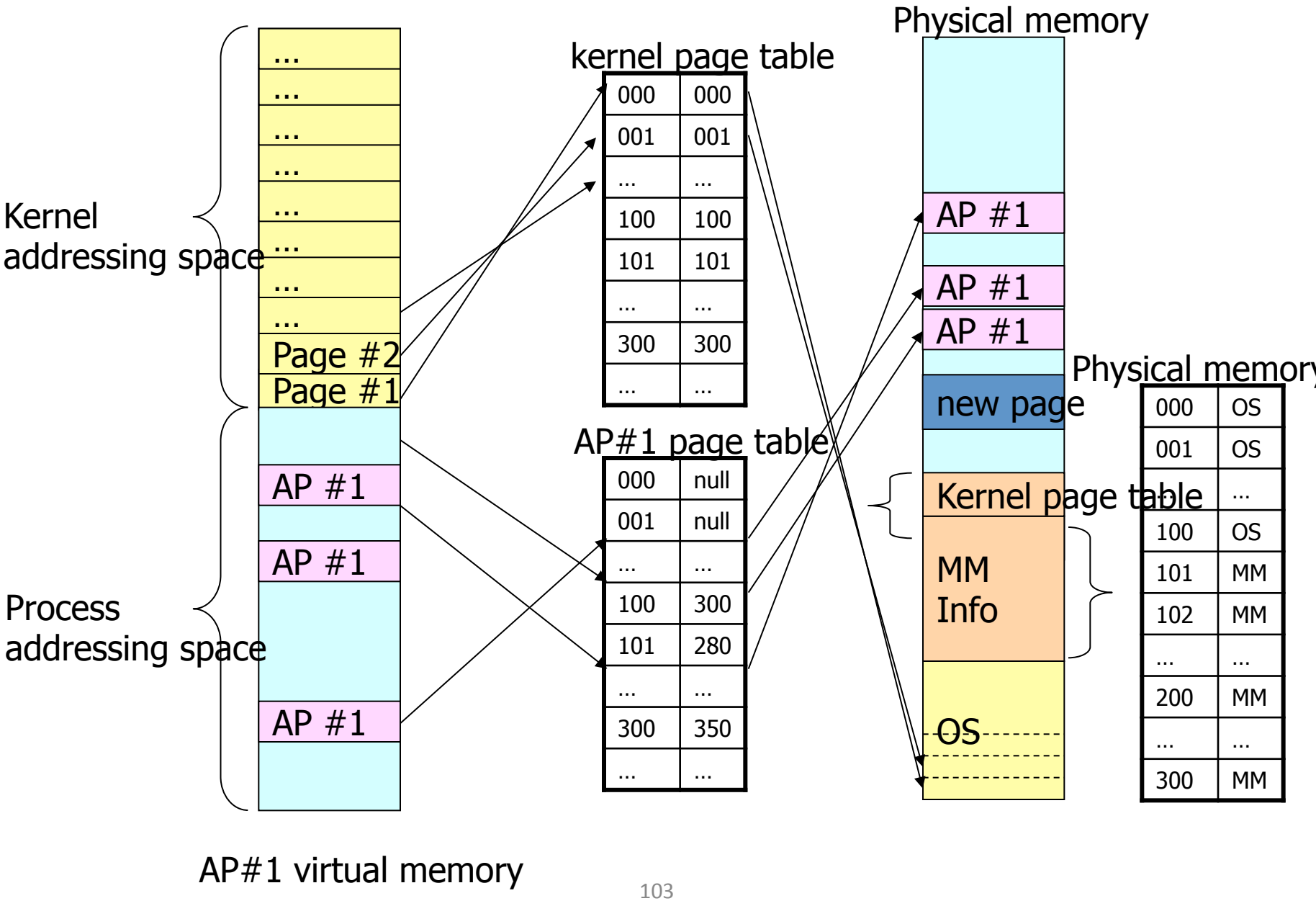
MM Basics (Cont.)

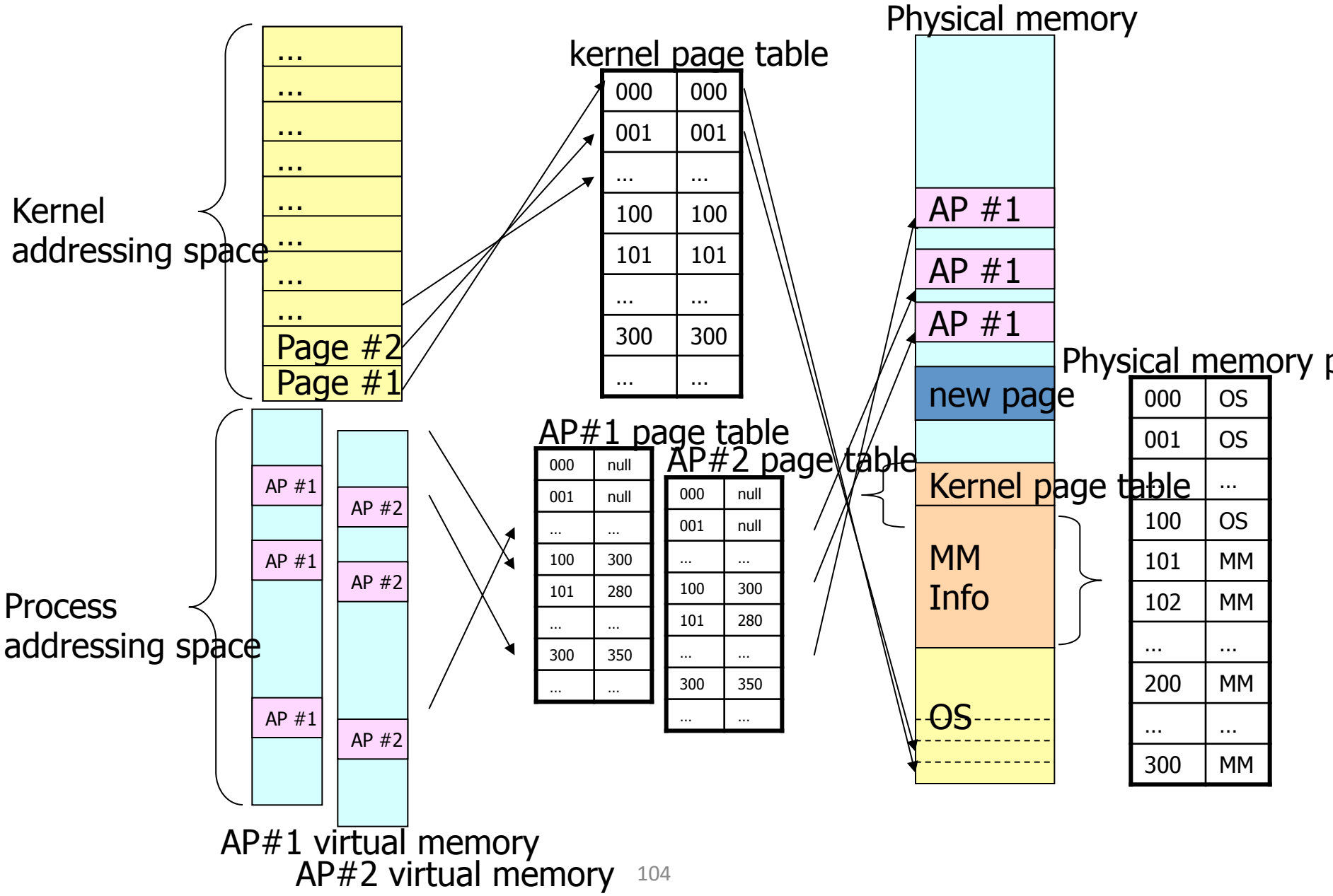
- application memory allocation/release



Overall architecture

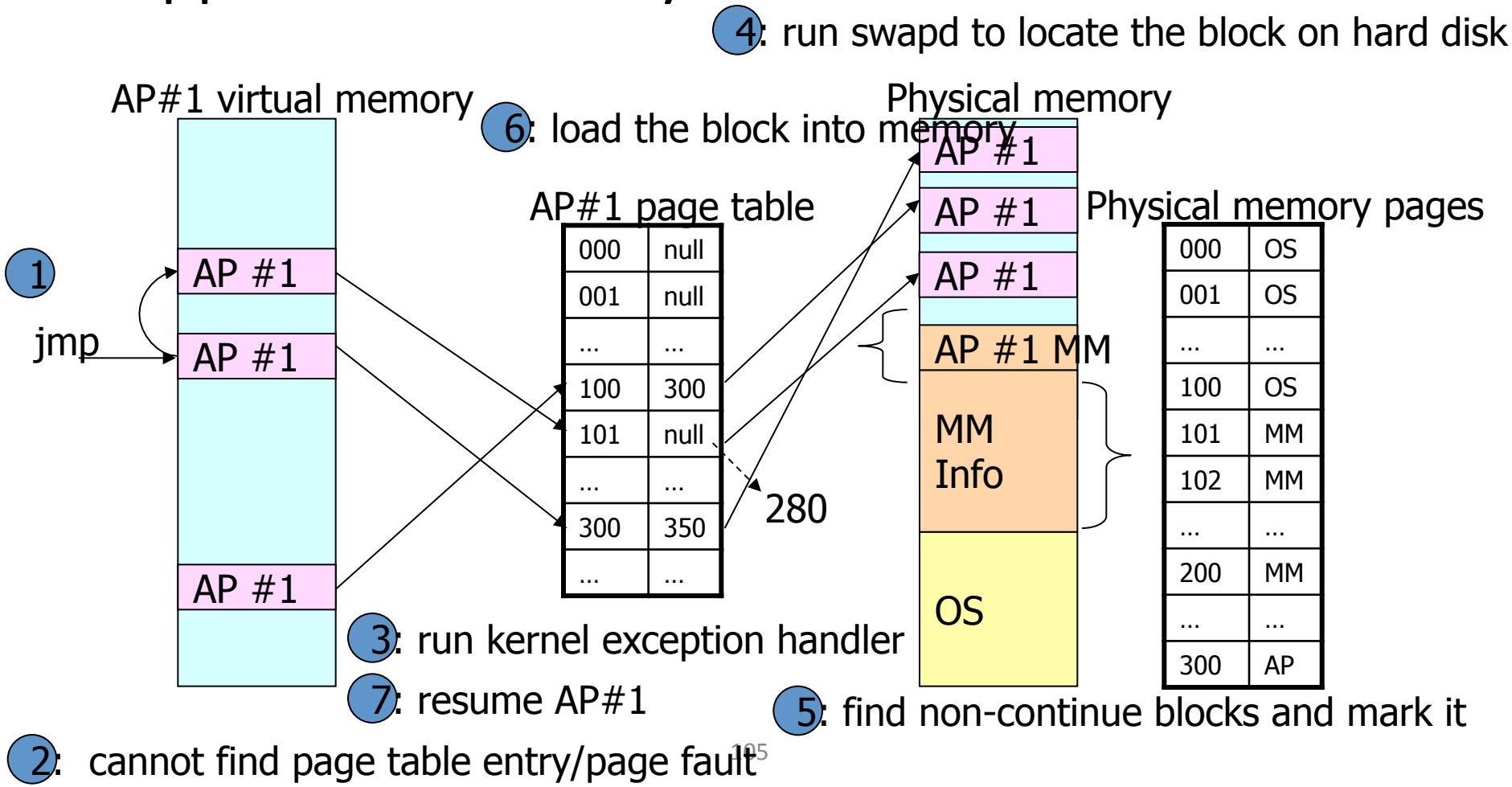



Physical memory
kernel page table
000 000
001 001
... ...
100 100
101 101
... ...
300 300
... ...
AP#1 page table
000 null
001 null
... ...
100 300
101 280
... ...
300 350
... ...
Kernel addressing space
Process addressing space
...
...
...
...
...
...
...
Page #2
Page #1
AP #1
AP #1
AP #1
AP #1
AP #1
new page
Kernel page table
MM Info
OS
Physical memory p
000 OS
001 OS
... ...
100 OS
101 MM
102 MM
... ...
200 MM
... ...
300 MM
AP#1 virtual memory

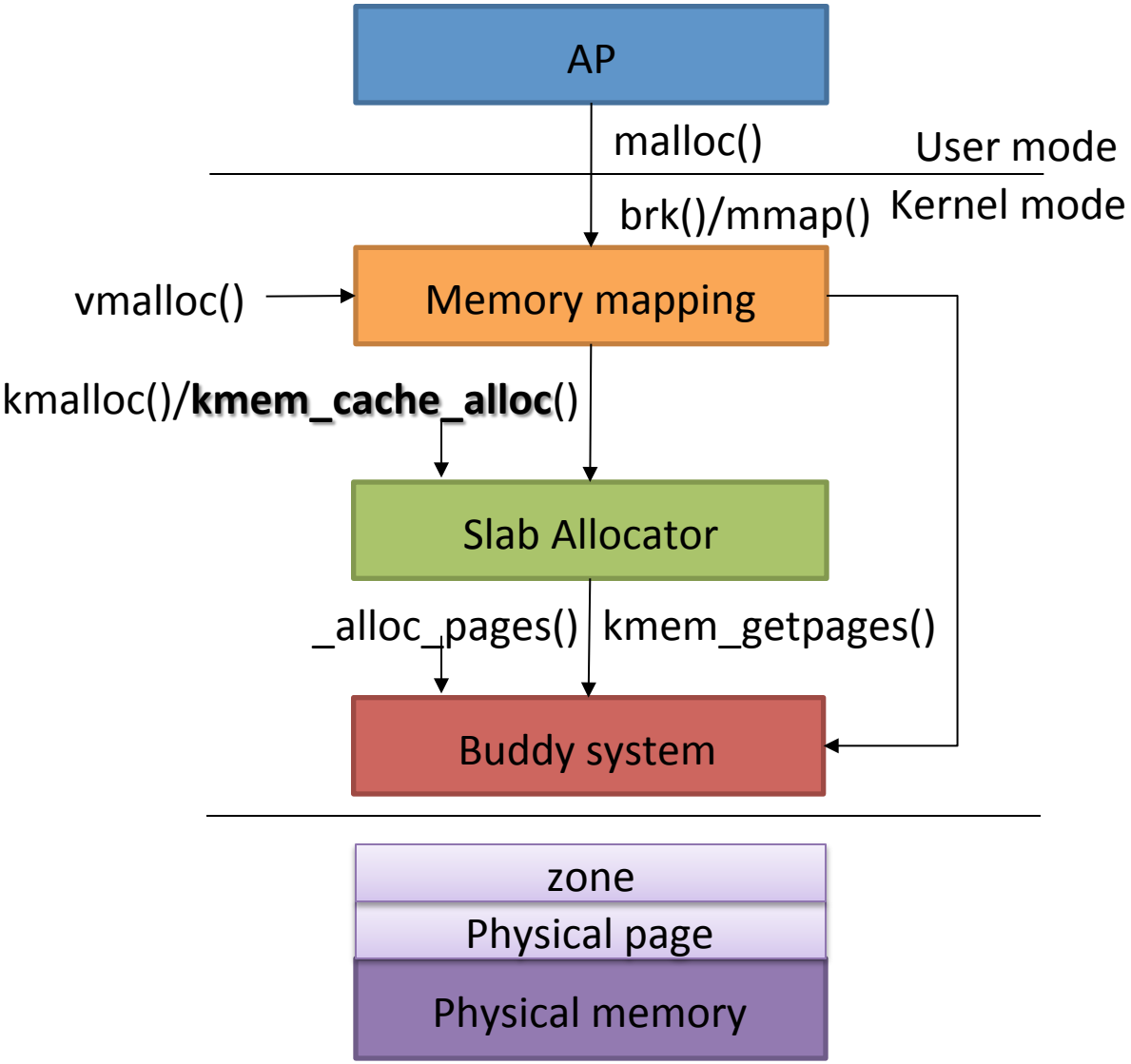

Physical memory
kernel page table
000 000
001 001
... ...
100 100
101 101
... ...
300 300
... ...
Kernel addressing space
...
...
...
...
...
...
...
Page #2
Page #1
AP #1
AP #1
AP #1
new page
Kernel page table
MM Info
OS
Physical memory p
000 OS
001 OS
... ...
100 OS
101 MM
102 MM
... ...
200 MM
... ...
300 MM
AP#1 page table
000 null
001 null
... ...
100 300
101 280
... ...
300 350
... ...
AP#2 page table
000 null
001 null
... ...
100 300
101 280
... ...
300 350
... ...
Process addressing space
AP #1
AP #1
AP #1
AP #2
AP #2
AP #2
AP#1 virtual memory
AP#2 virtual memory

MM Basics (Cont.)

- application memory fault

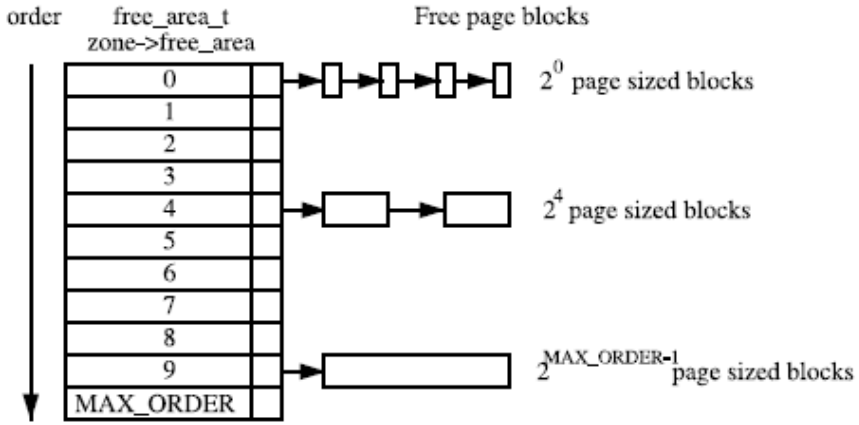


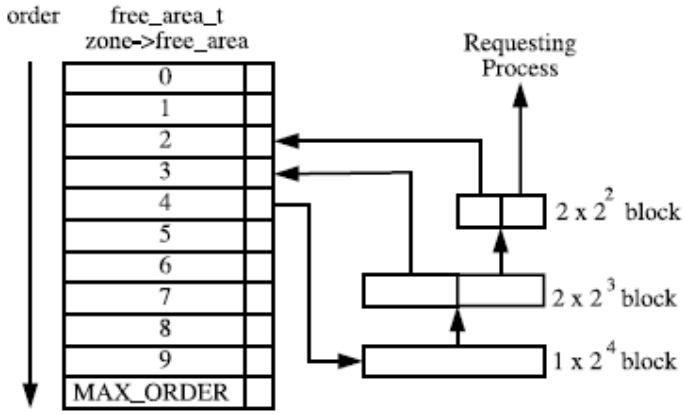
Overall architecture



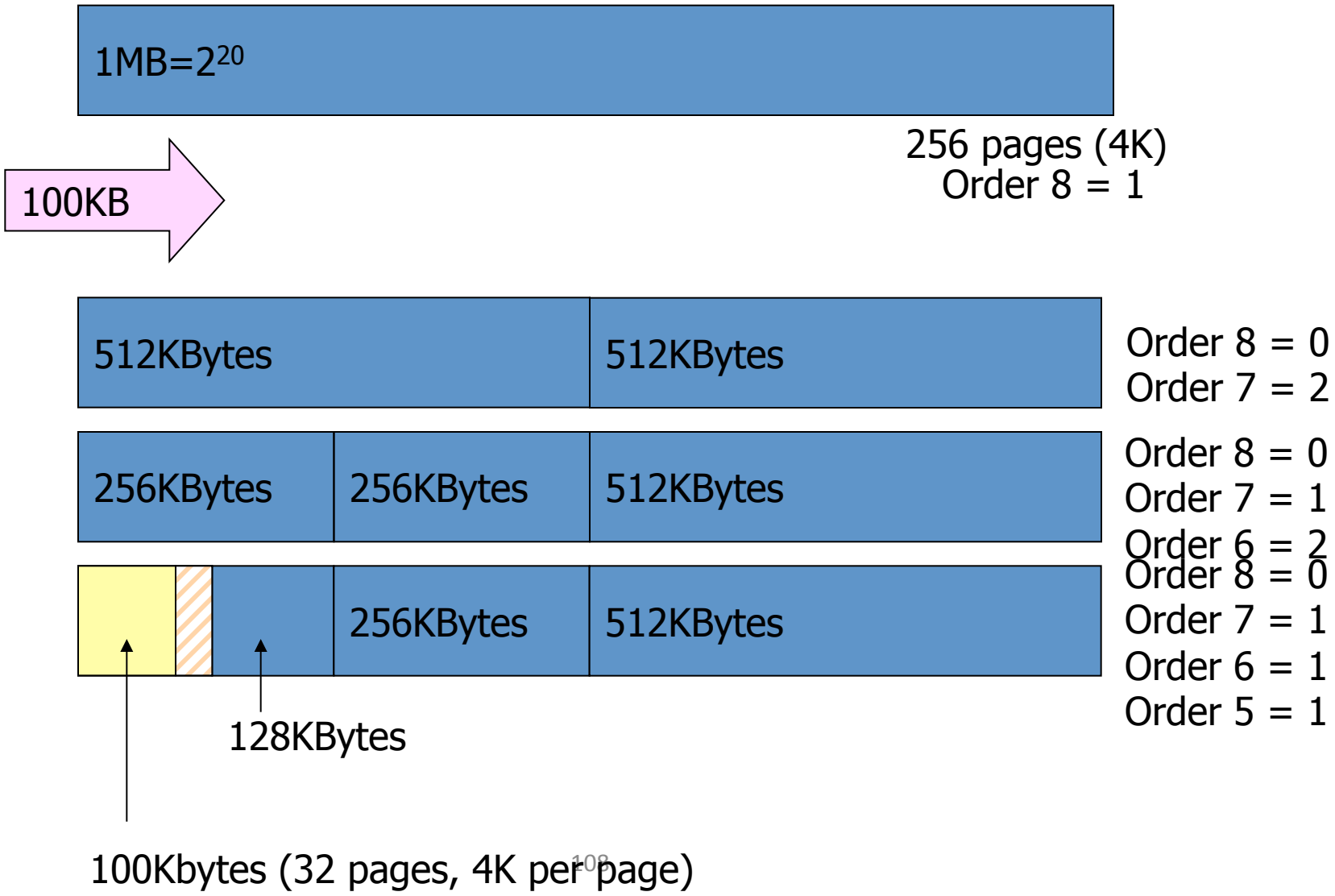
Linux MM

- Buddy allocation
  - Why ?
  - alloc_page

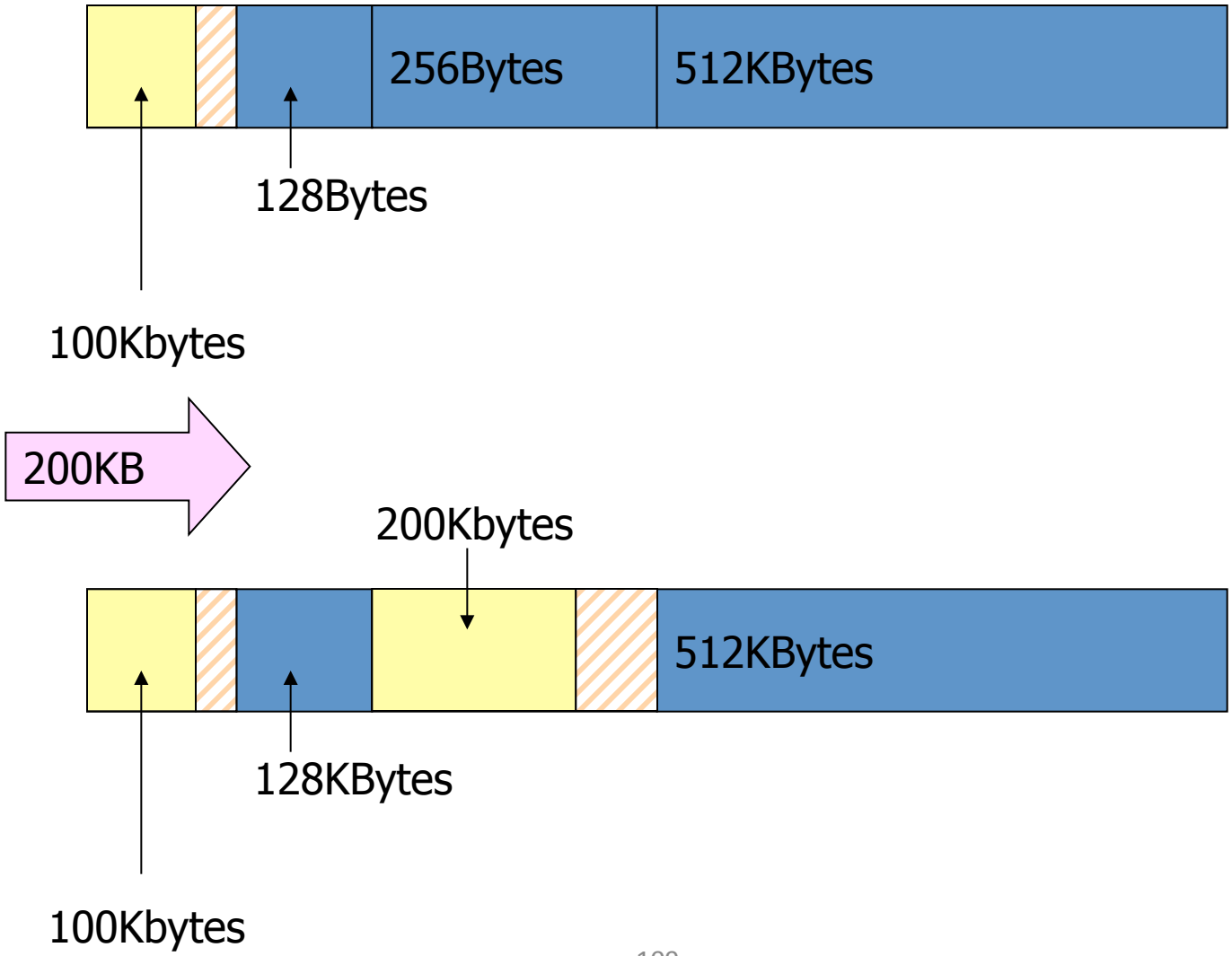




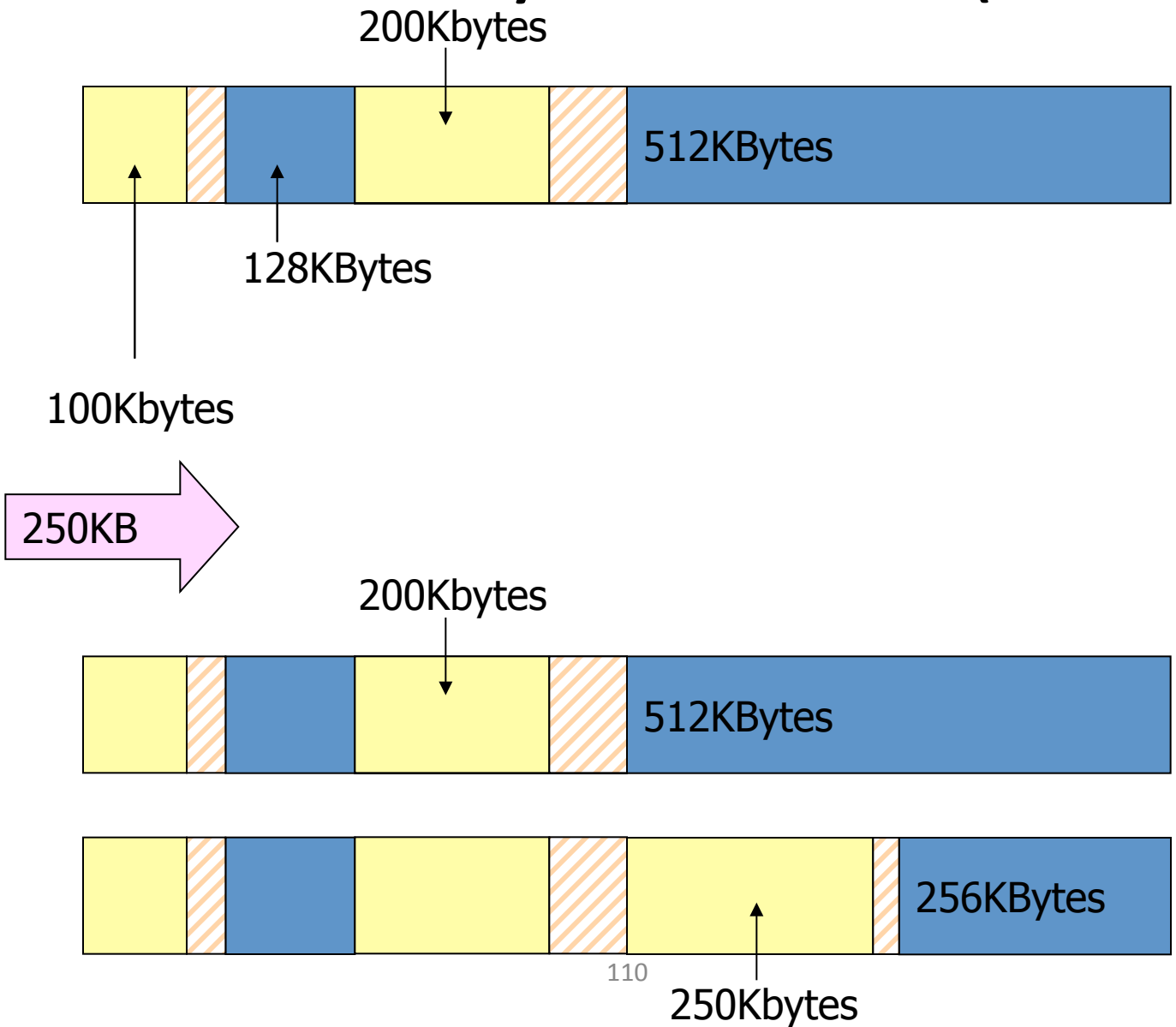
Zoned Buddy Allocator



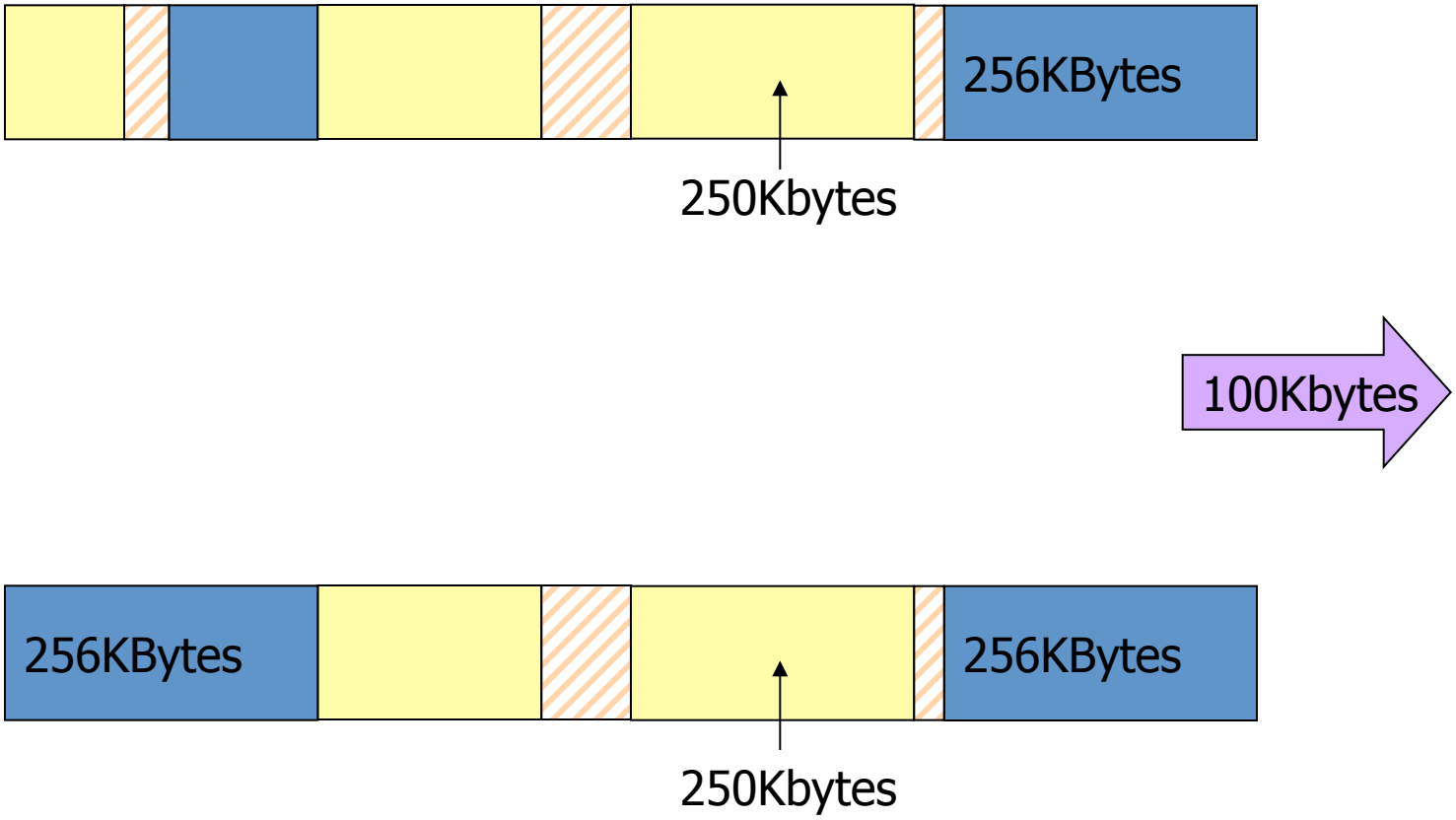
Zoned Buddy Allocator (Cont.)



Zoned Buddy Allocator (Cont.)

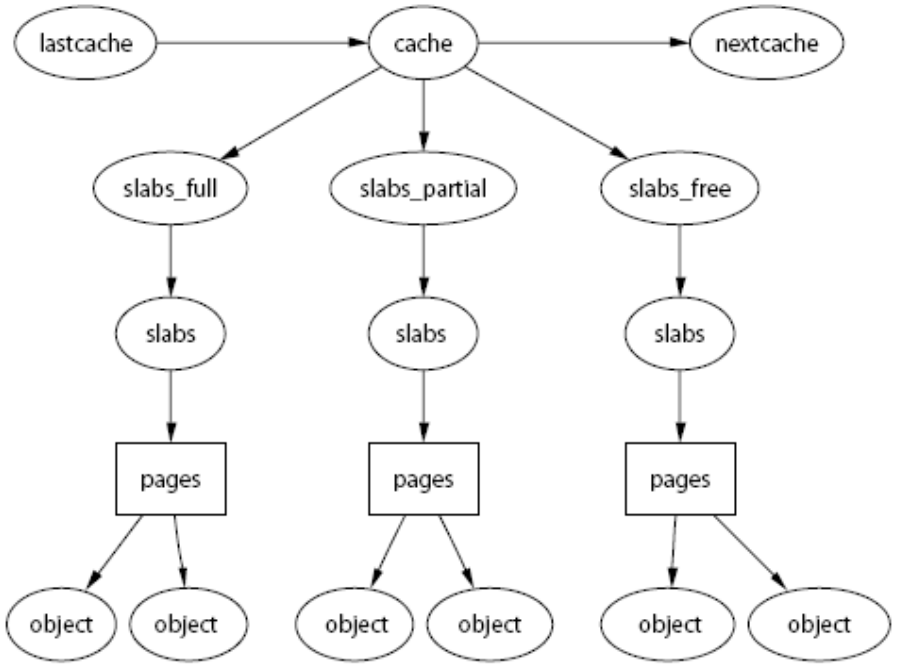


Zoned Buddy Allocator (Cont.)



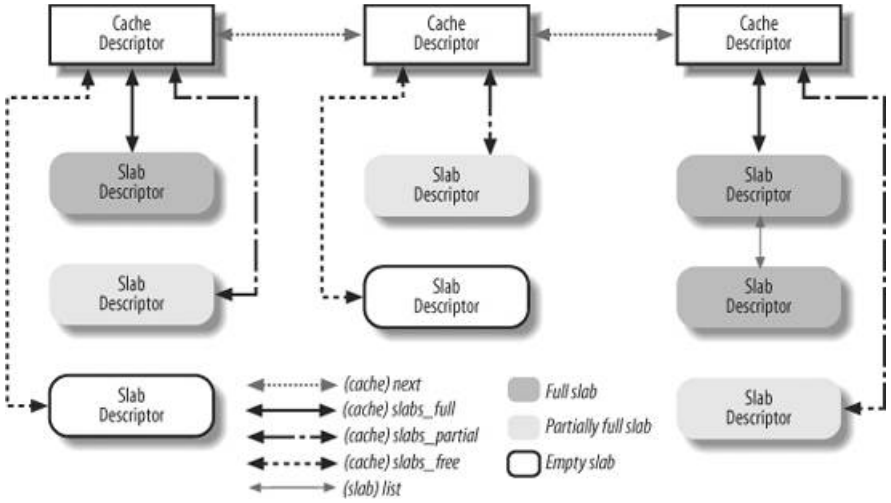
Linux MM

- Slab allocation
  - Why ?



Linux MM

- Slab allocation



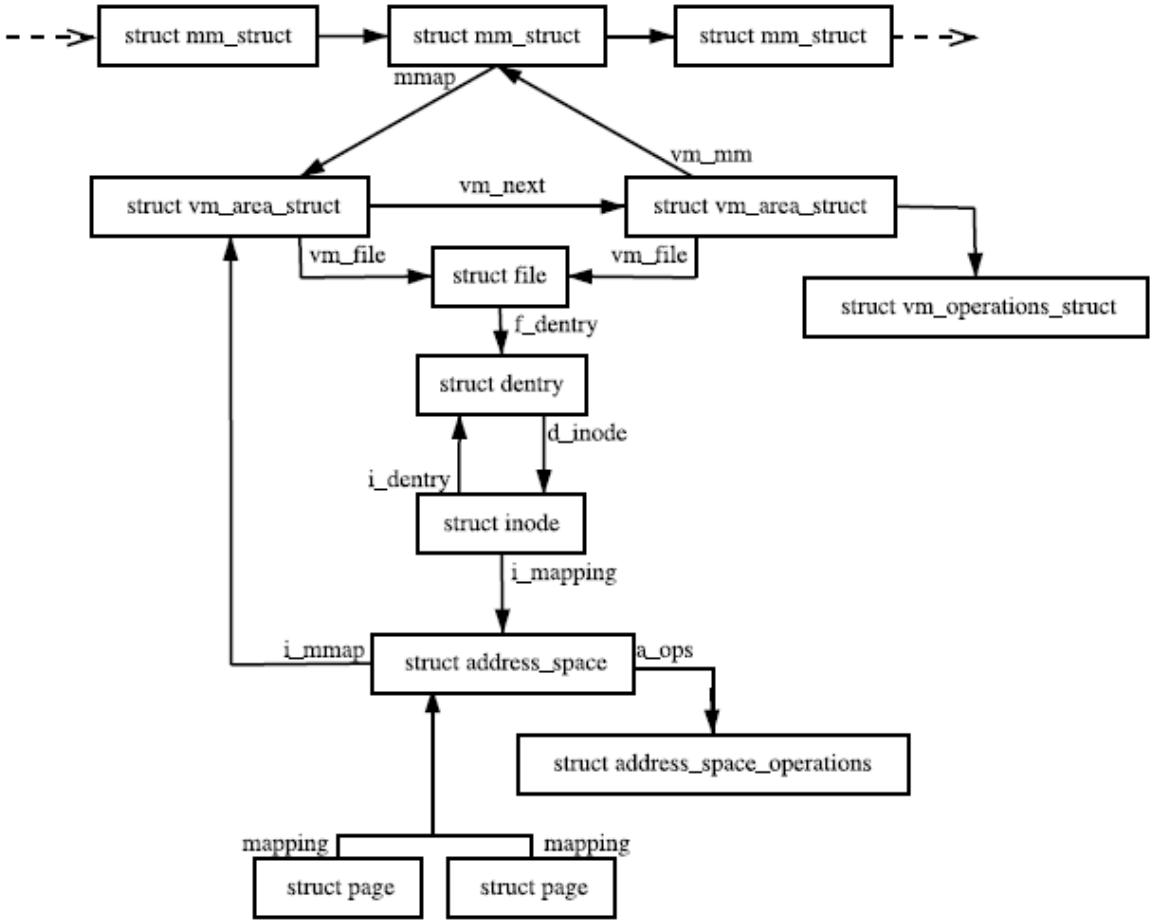
Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 3rd Edition

Linux MM

- kmalloc/vmalloc
  - kmalloc()
    - similar to that of user-space's familiar malloc() routine
    - byte-sized chunks
    - memory allocated is physically contiguous
    - Through slab allocator
  - vmalloc()
    - virtually contiguous and not necessarily physically contiguous
    - user-space allocation function works
    - allocating potentially noncontiguous chunks of physical memory and "fixing up" the page tables to map the memory into a contiguous chunk of the logical address space

Linux MM

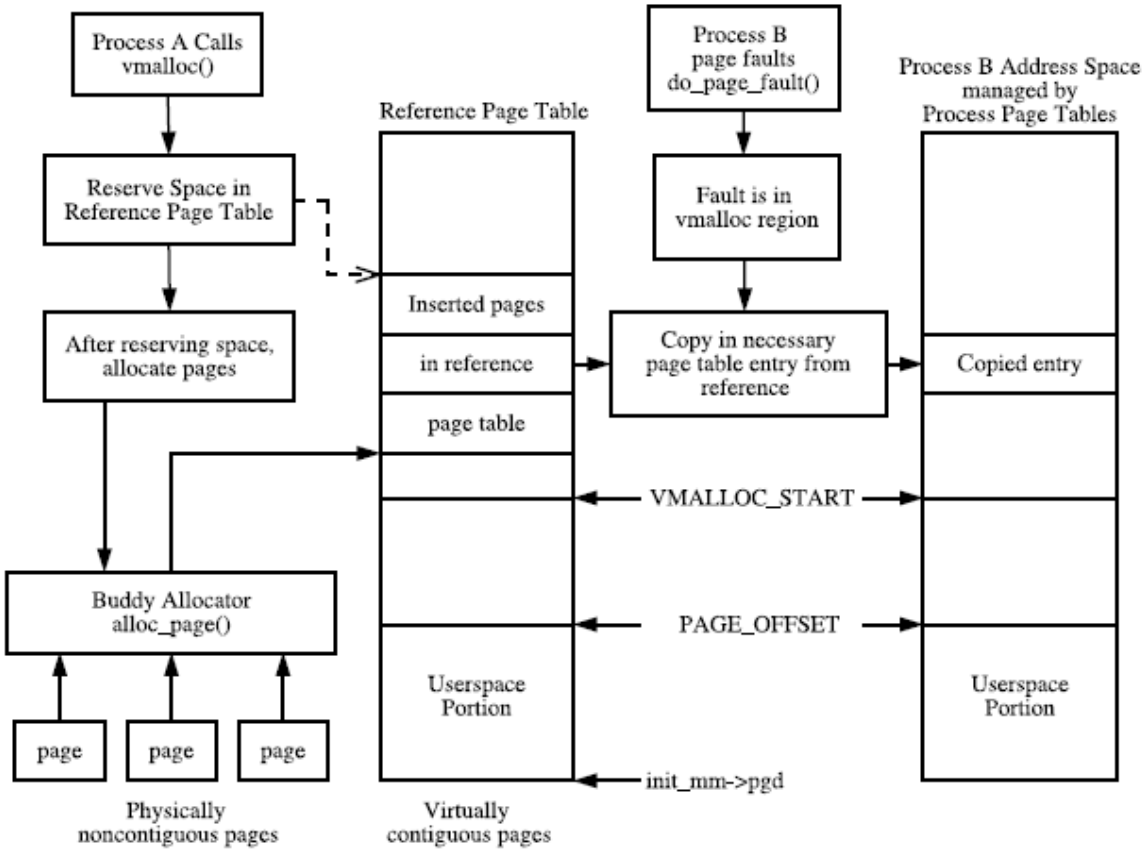
- Address space descriptor and Memory region



Mel Gorman, "**Understanding the Linux Virtual Memory Manager**"

Linux MM

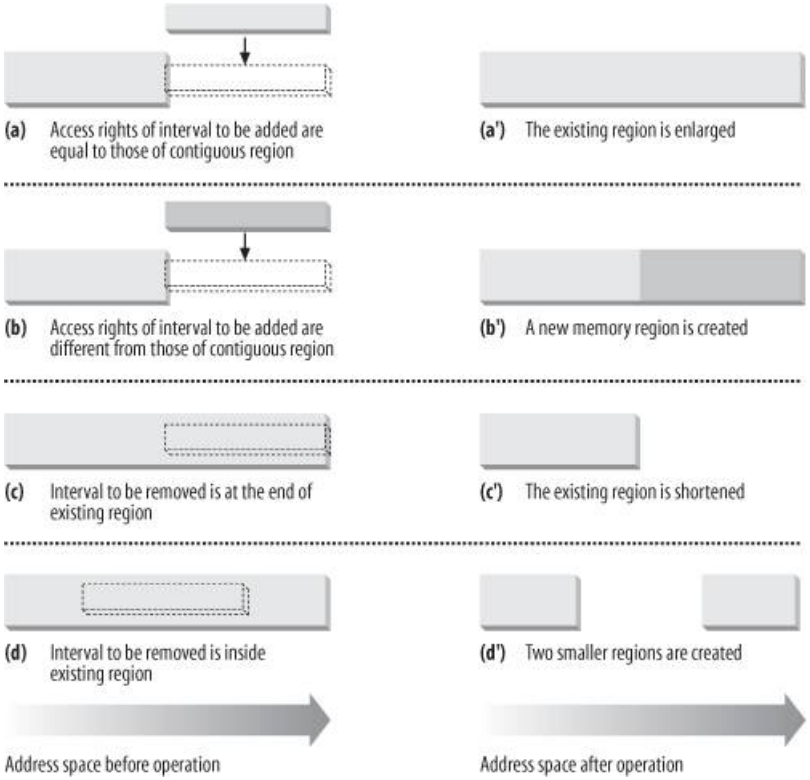
- User memory allocation



Mel Gorman, "**Understanding the Linux Virtual Memory Manager**"

Linux MM

- Process address space



Interrupt/Exception Handling

Outline

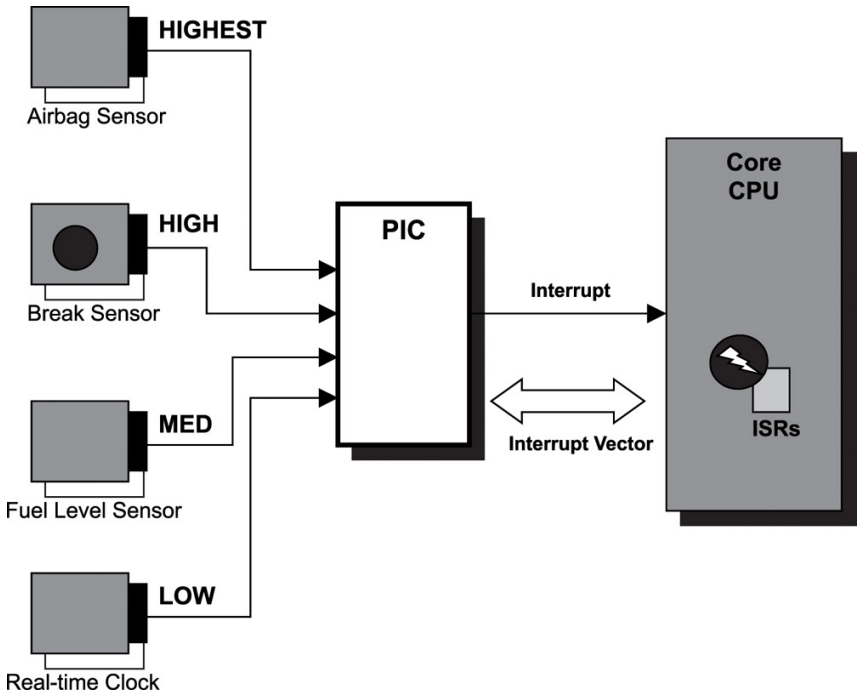
- Interrupt and exception definitions
- Interrupt and exception procedures

Interrupt and exception definitions

- External vs. Internal
- Synchronous vs. Asynchronous
- Exception
  - Any event that disrupts the normal execution of the processor and forces the processor into execution of special instructions in a privileged state
- Interrupt
  - An asynchronous exception triggered by an event that an external hardware device generates

Exceptions and interrupts (Cont.)

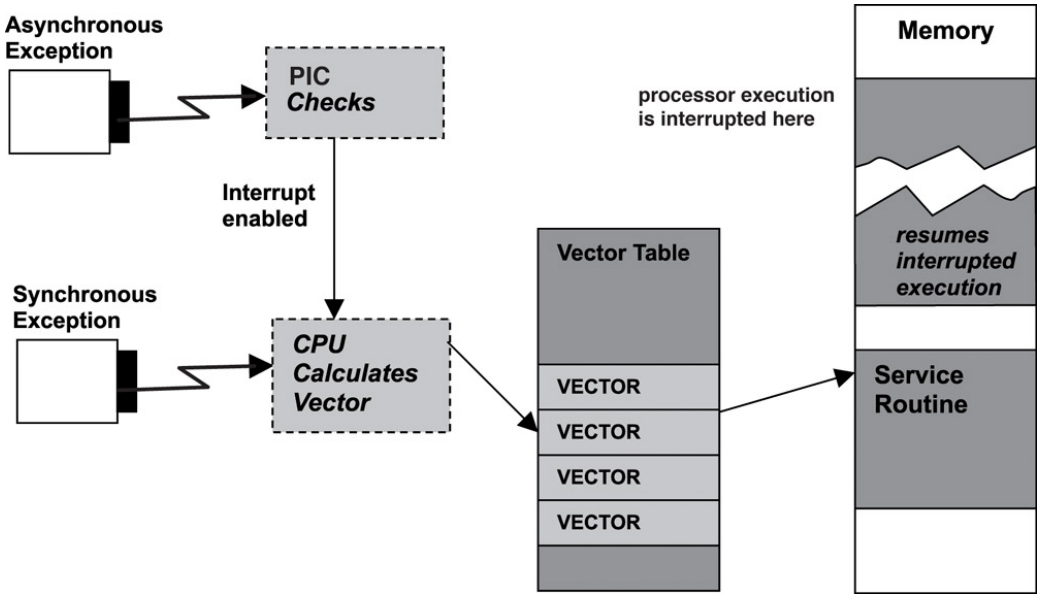
- How does exceptions and interrupts work ?
  - Programmable interrupt controller



Source: Qing Li "real-time concepts for embedded s

Exceptions and interrupts (Cont.)

- Processing general exceptions (Cont.)
  - Loading and invoking exception handlers

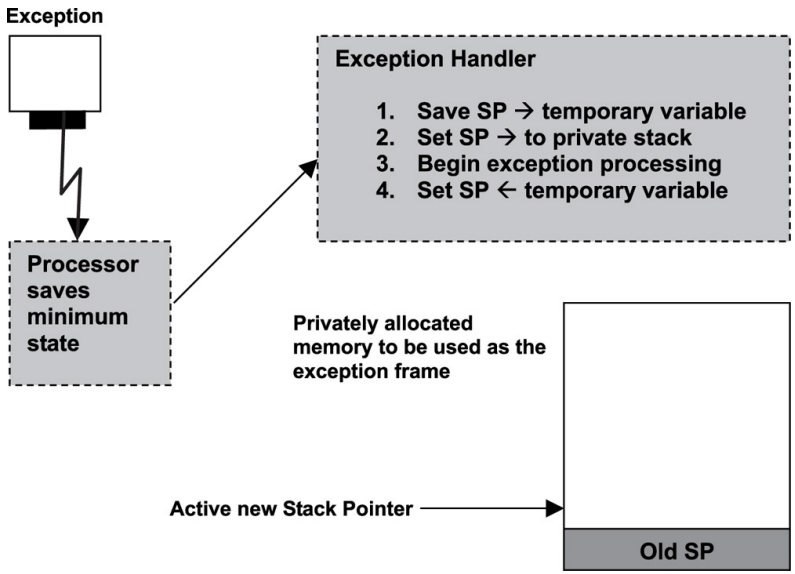


  - Nested exceptions and stack overflow

Source: Qing Li "real-time concepts for embedded systems"

Exceptions and interrupts (Cont.)

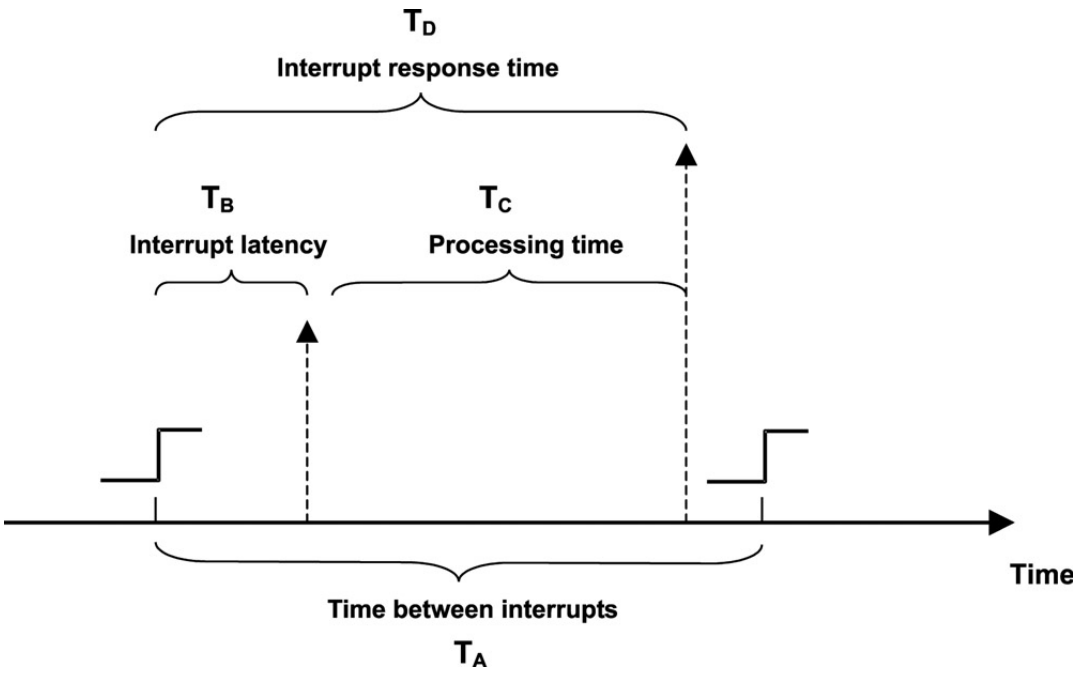
- Exception handlers
  - Exception frame
    - The exception frame is also called the interrupt stack in the context of asynchronous exceptions.



Source: Qing Li "real-time concepts for embedded systems"

Exceptions and interrupts (Cont.)

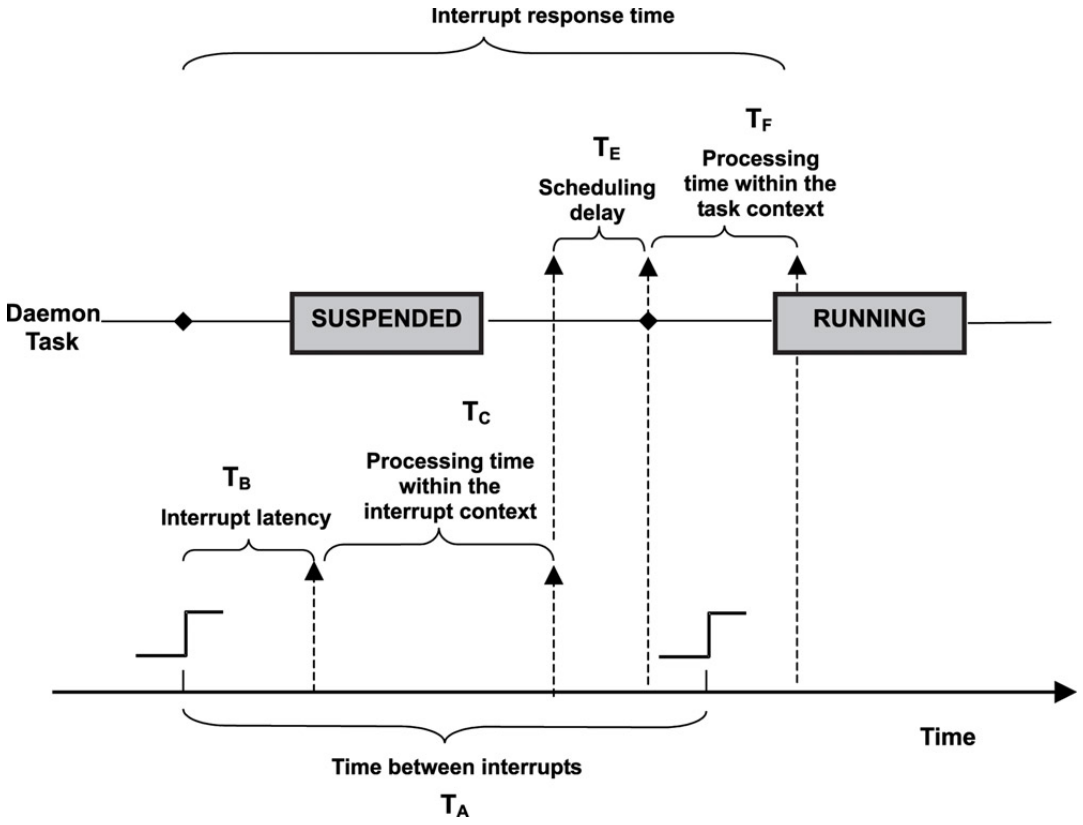
- Exception timing



Source: Qing Li "real-time concepts for embedded systems"

Exceptions and interrupts (Cont.)

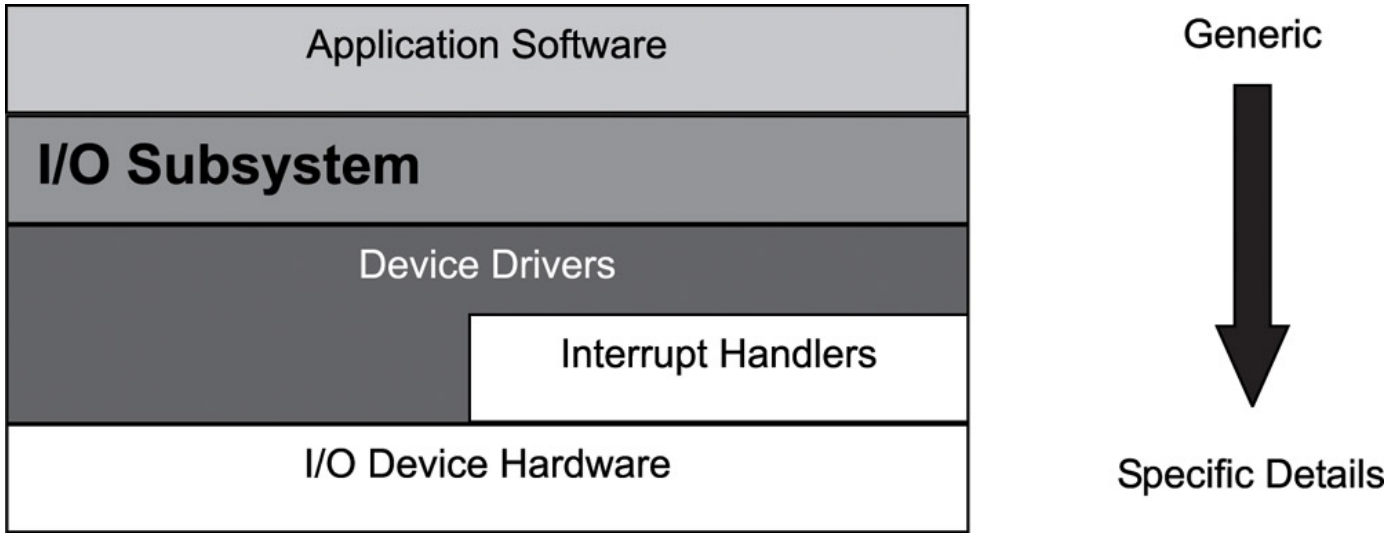
- Exception timing (Cont.)



Source: Qing Li "real-time concepts for embedded systems"

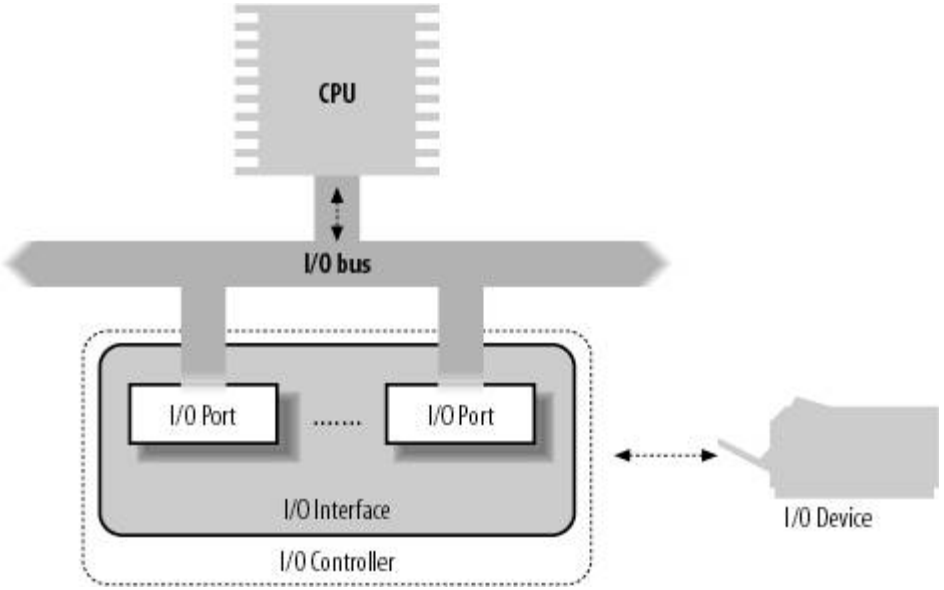
I/O subsystem

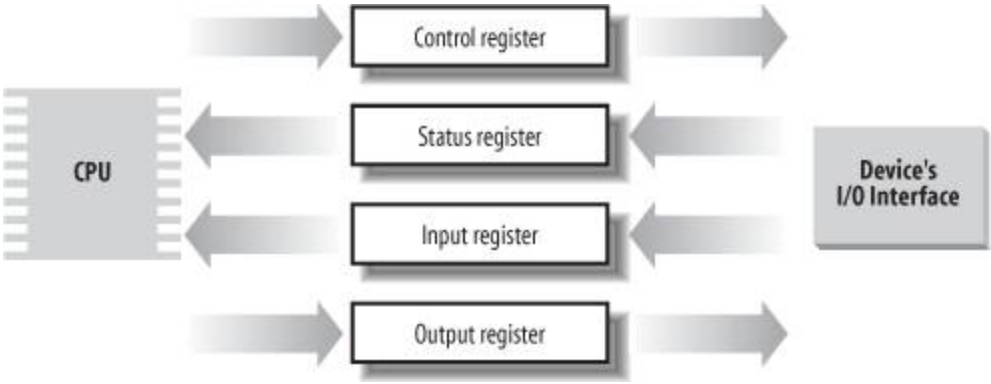
- I/O subsystem : a layered model



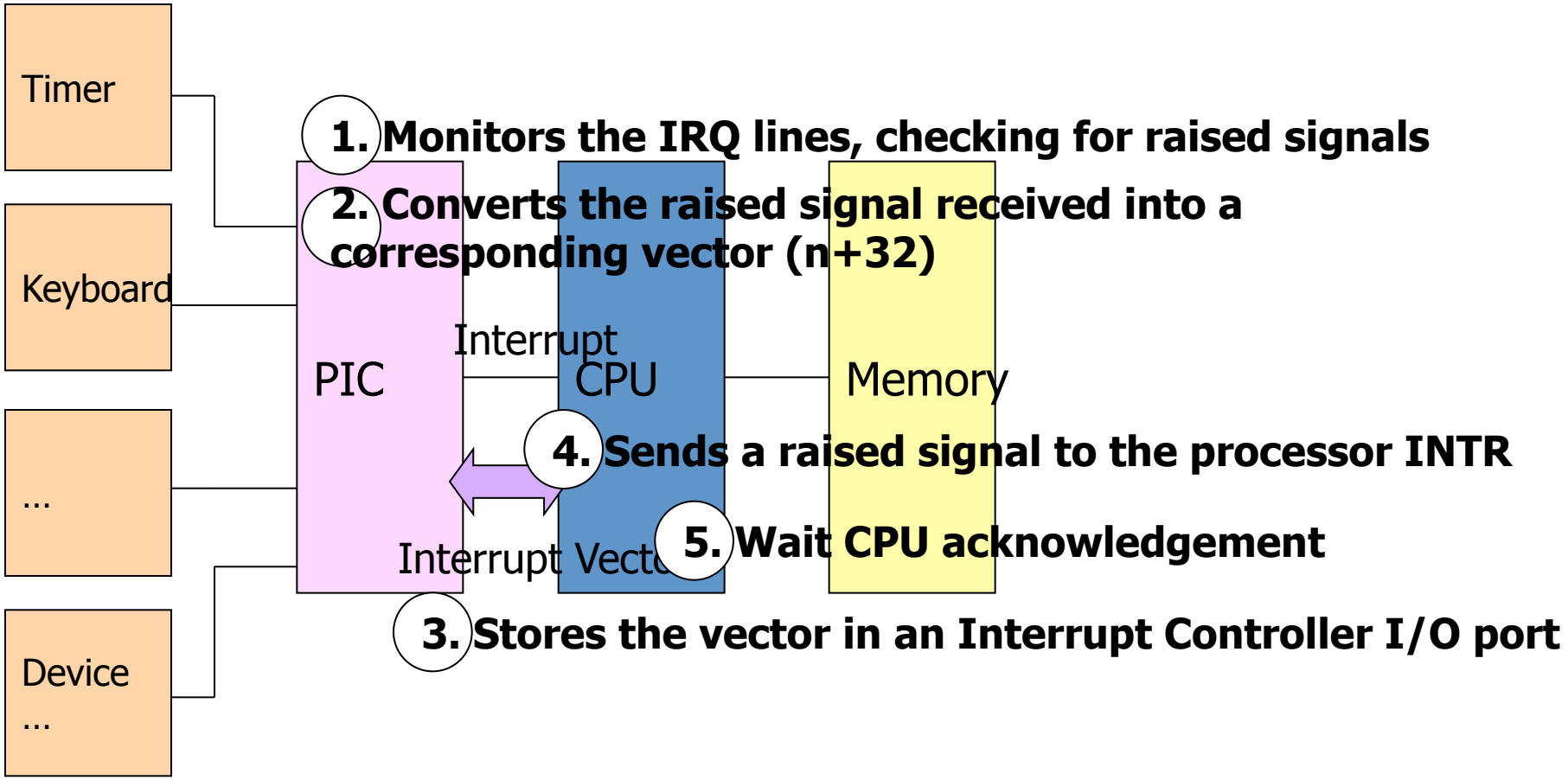
Source: Qing Li "real-time concepts for embedded systems"

I/O subsystem

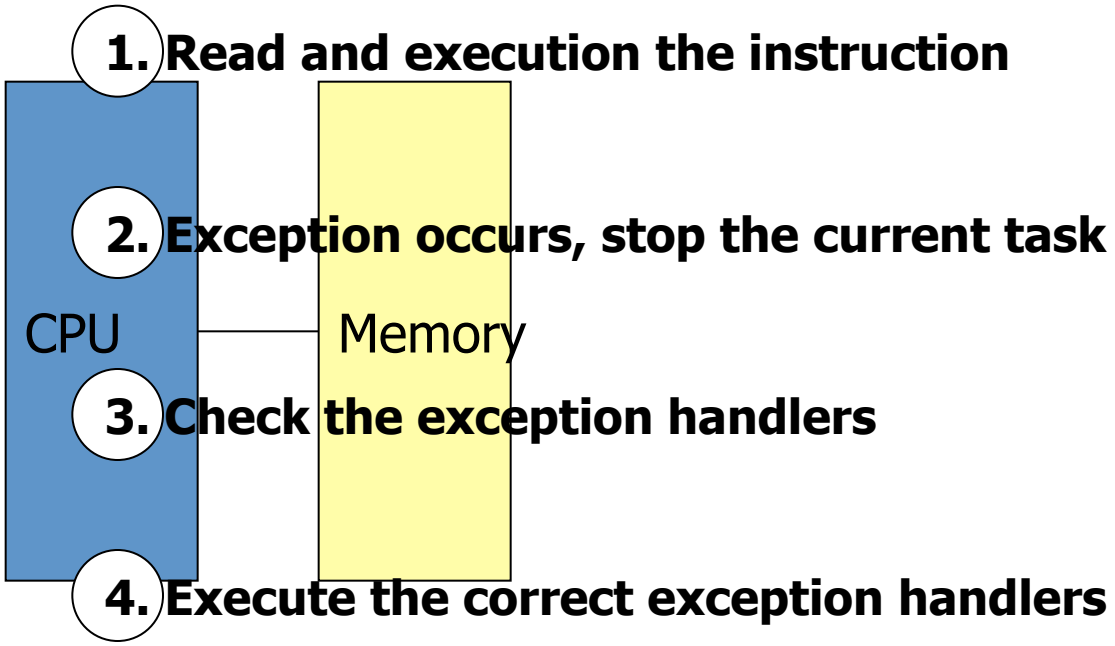




Interrupt


Timer
Keyboard
...
Device
...
PIC
Interrupt
CPU
Memory
Interrupt Vecto
1. Monitors the IRQ lines, checking for raised signals
2. Converts the raised signal received into a corresponding vector (n+32)
3. Stores the vector in an Interrupt Controller I/O port
4. Sends a raised signal to the processor INTR
5. Wait CPU acknowledgement

Exception


1. Read and execution the instruction
2. Exception occurs, stop the current task
CPU
Memory
3. Check the exception handlers
4. Execute the correct exception handlers

Interrupt and exception definitions

- Interrupts
  - Maskable interrupts (device interrupt)
  - Nonmaskable interrupts (hardware failure)
- Exceptions
  - Processor-detected exceptions (divided by 0)
  - Faults (page fault)
  - Traps (breakpoint)
  - Aborts (serious error)
  - Programmed exceptions (software interrupt)

Linux interrupt handler

```
/* request_irq: allocate a given interrupt line */
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long irqflags,
                const char *devname,
                void *dev_id)
```

- SA_INTERRUPT
  - This enables a fast handler to complete quickly, without possible interruption from other interrupts.
- SA_SAMPLE_RANDOM
  - This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool. The kernel entropy pool provides truly random numbers derived from various random events.
- SA_SHIRQ
  - This flag specifies that the interrupt line can be shared among multiple interrupt handlers.

Linux interrupt handler

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs)
```

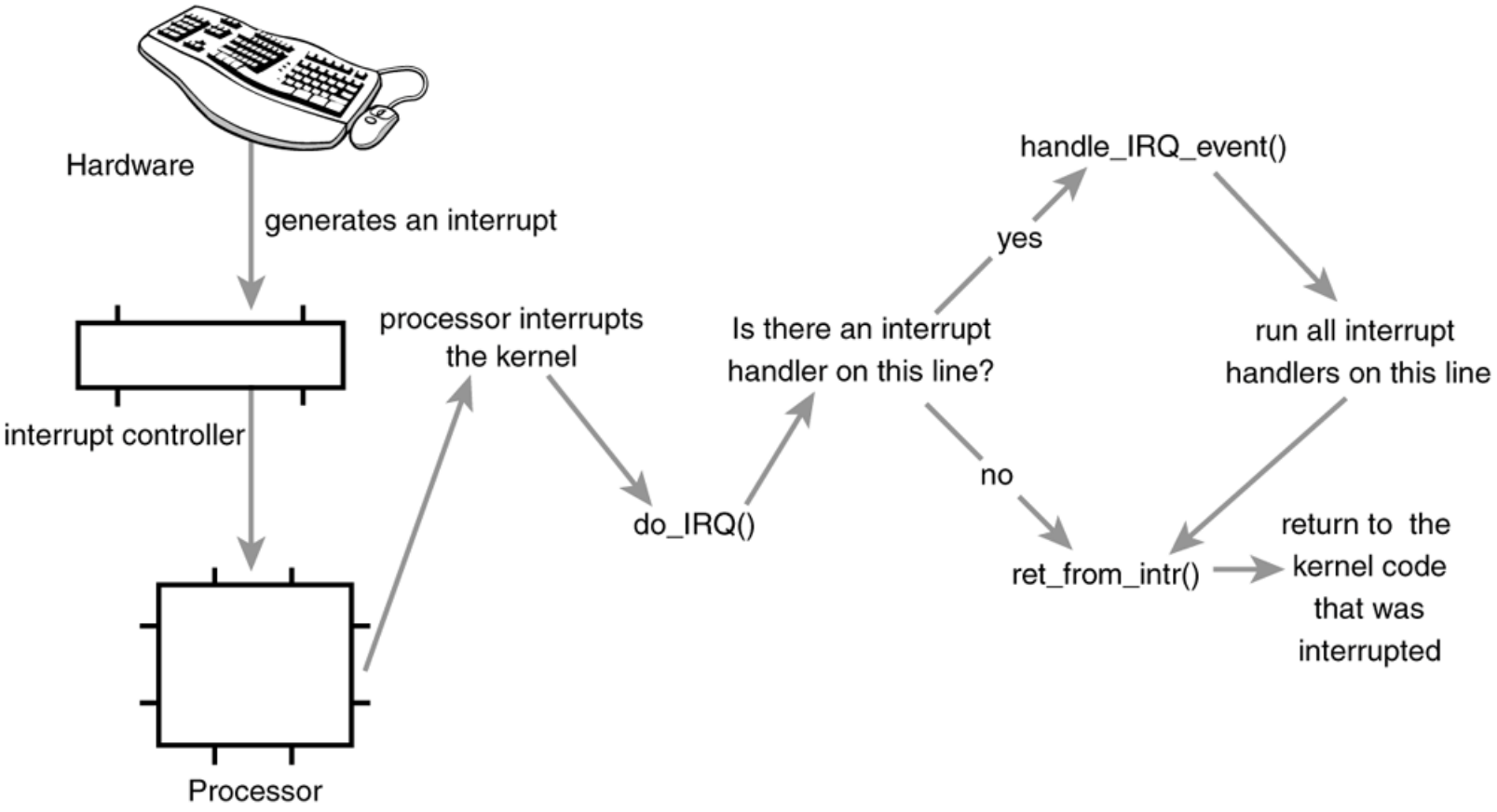
```
    /* register rtc_interrupt on RTC_IRQ */
    if (request_irq(RTC_IRQ, rtc_interrupt, SA_INTERRUPT, "rtc", NULL) {
            printk(KERN_ERR "rtc: cannot register IRQ %d\n", RTC_IRQ);
            return -EIO;
    }
```

```
static irqreturn_t rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        /*
         * Can be an alarm interrupt, update complete interrupt,
         * or a periodic interrupt. We store the status in the
         * low byte and the number of interrupts received since
         * the last read in the remainder of rtc_irq_data.
         */

        spin_lock (&rtc_lock);

        rtc_irq_data += 0x100;
        rtc_irq_data &= ~0xff;
        rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);

        if (rtc_status & RTC_TIMER_ON)
            mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

        spin_unlock (&rtc_lock);

        /*
         * Now do the rest of the actions
         */
        spin_lock(&rtc_task_lock);
        if (rtc_callback)
                rtc_callback->func(rtc_callback->private_data);
        spin_unlock(&rtc_task_lock);
        wake_up_interruptible(&rtc_wait);

        kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);

        return IRQ_HANDLED;
}
```

Interrupt and exception definitions

#	Exception	Exception handler
0	Divide error	divide_error( )
1	Debug	debug( )
2	NMI	nmi( )
3	Breakpoint	int3( )
4	Overflow	overflow( )
5	Bounds check	bounds( )
6	Invalid opcode	invalid_op( )
7	Device not available	device_not_available( )
8	Double fault	doublefault_fn( )
9	Coprocessor segment overrun	coprocessor_segment_overrun( )
10	Invalid TSS	invalid_TSS( )
11	Segment not present	segment_not_present( )
12	Stack segment fault	stack_segment( )
13	General protection	general_protection( )
14	Page Fault	page_fault( )
…	…	…

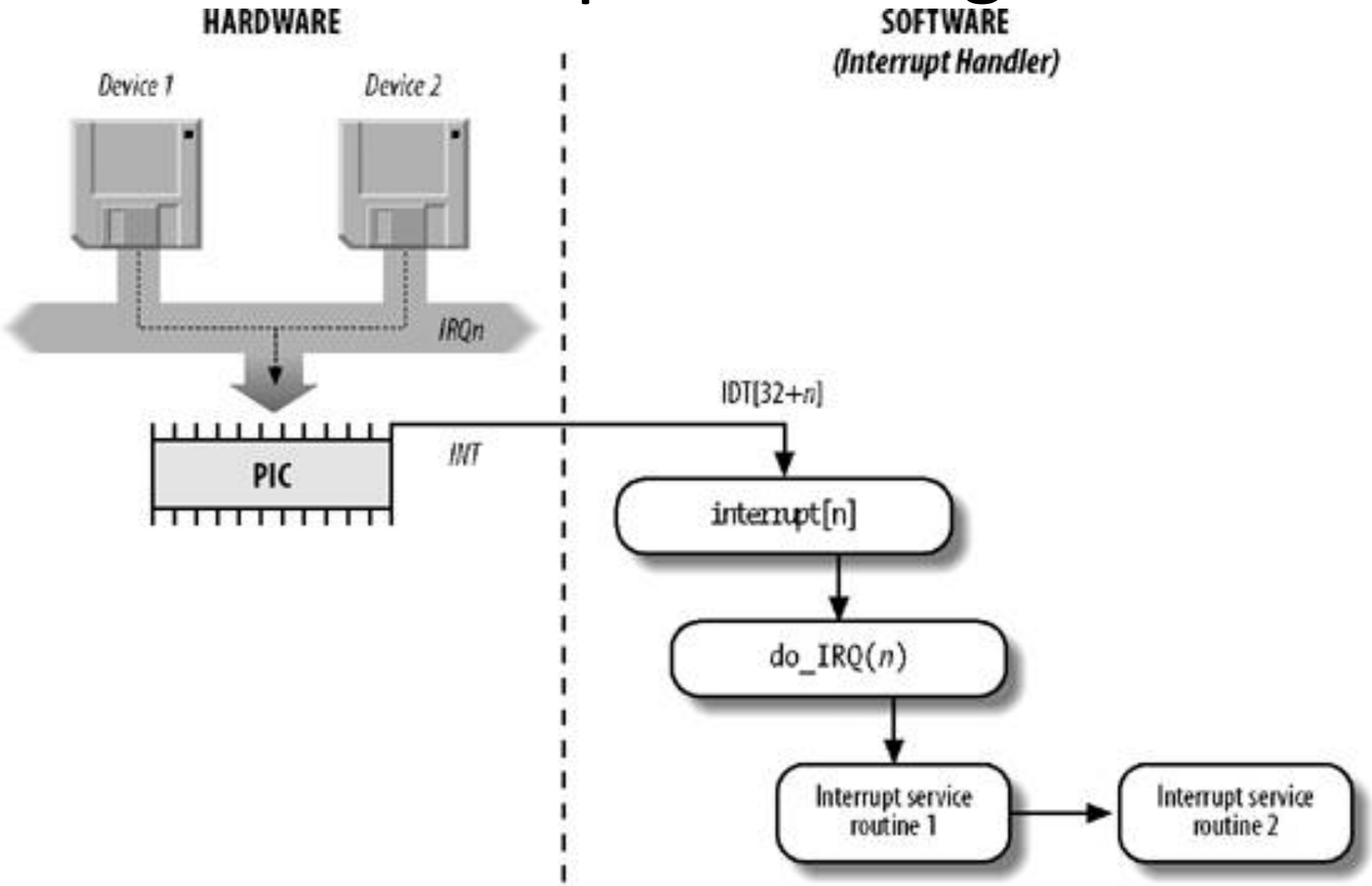
Interrupt Handling



Robert Love, "Linux Kernel Development," Second Edition

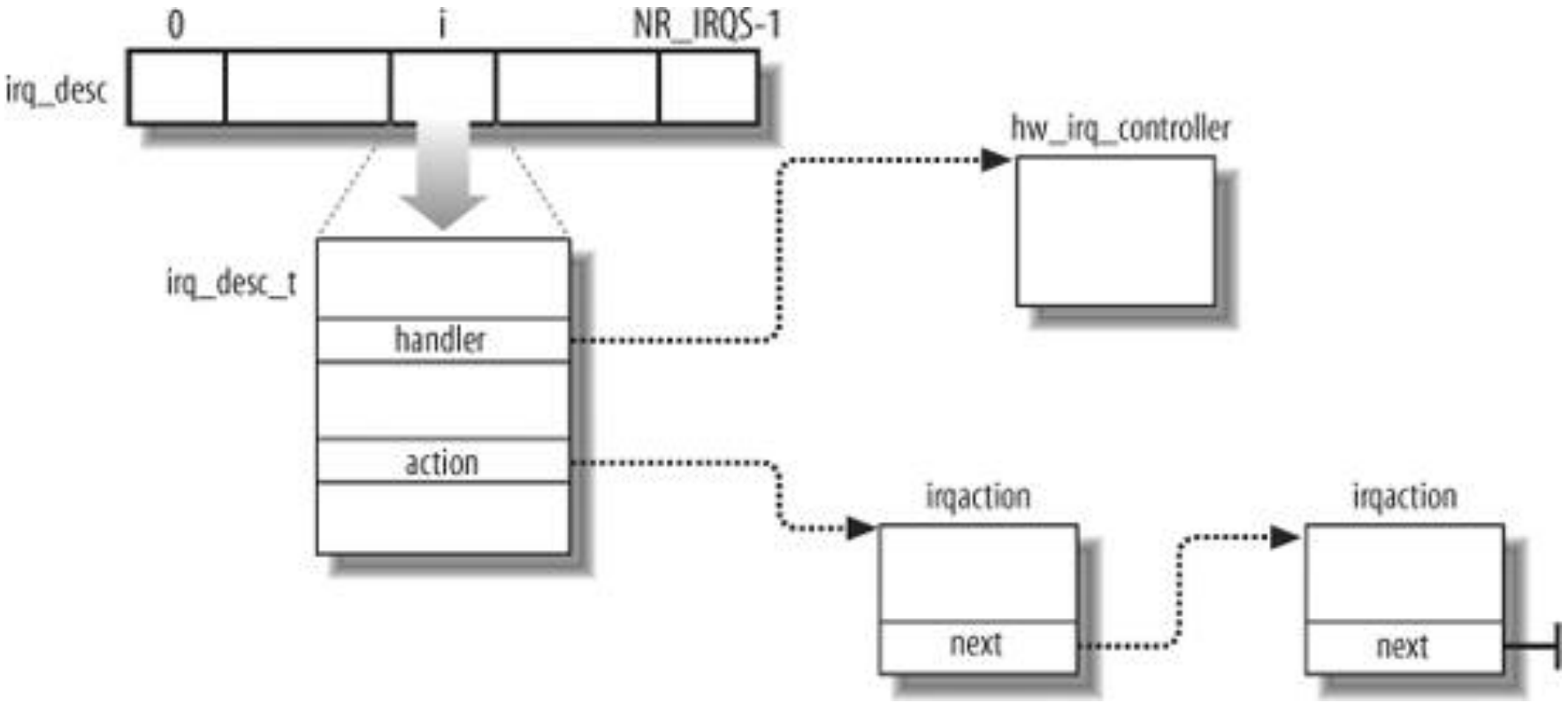
```
asmlinkage int handle_IRQ_event(unsigned int irq, struct pt_regs *regs,
                                struct irqaction *action)
{
        int status = 1;
        int retval = 0;

        if (!(action->flags & SA_INTERRUPT))
                local_irq_enable();

        do {
                status |= action->flags;
                retval |= action->handler(irq, action->dev_id, regs);
                action = action->next;
        } while (action);

        if (status & SA_SAMPLE_RANDOM)
                add_interrupt_randomness(irq);

        local_irq_disable();

        return retval;
}
```

Interrupt Handling



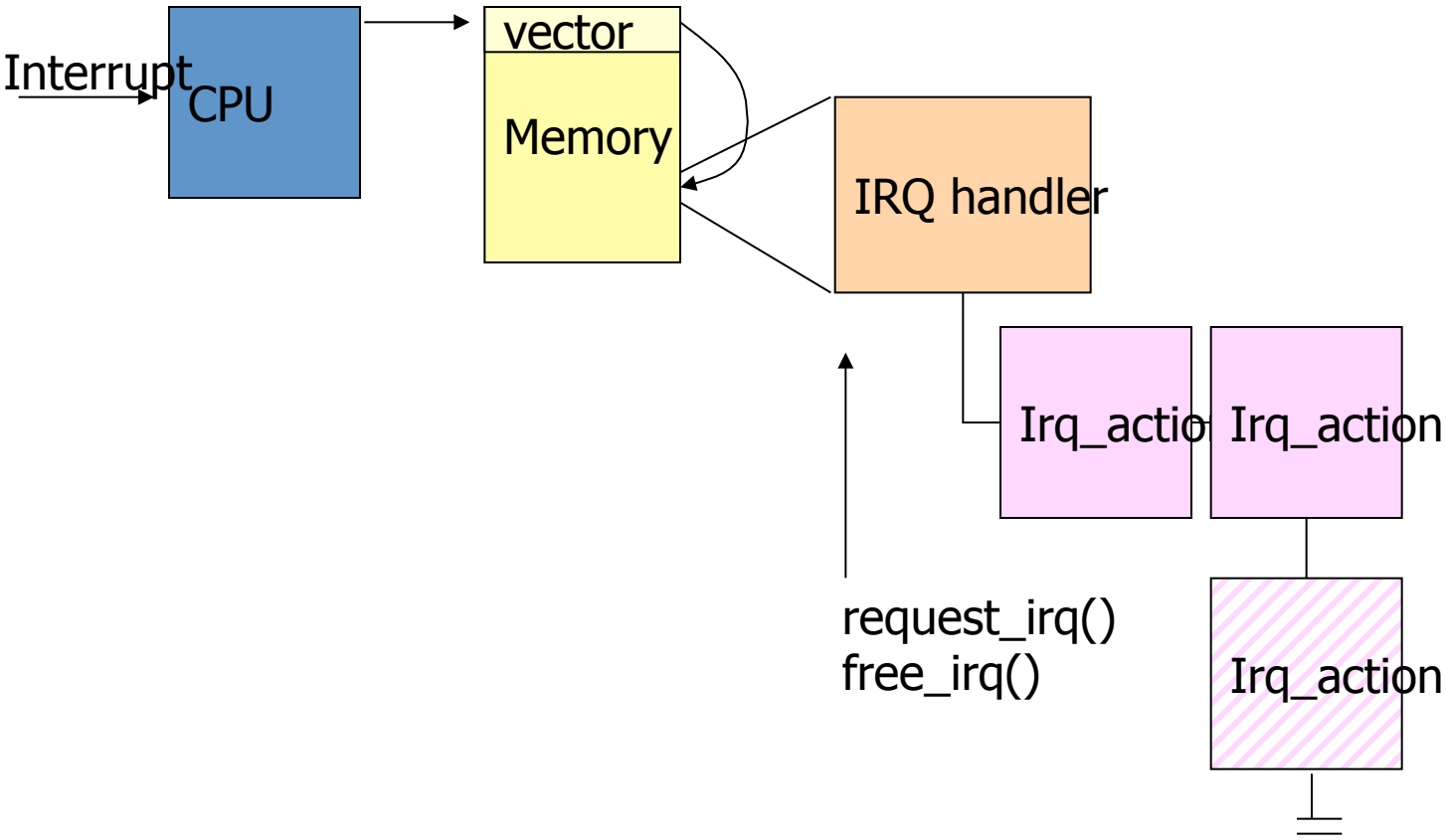
Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 3rd Edition

IRQ descriptors

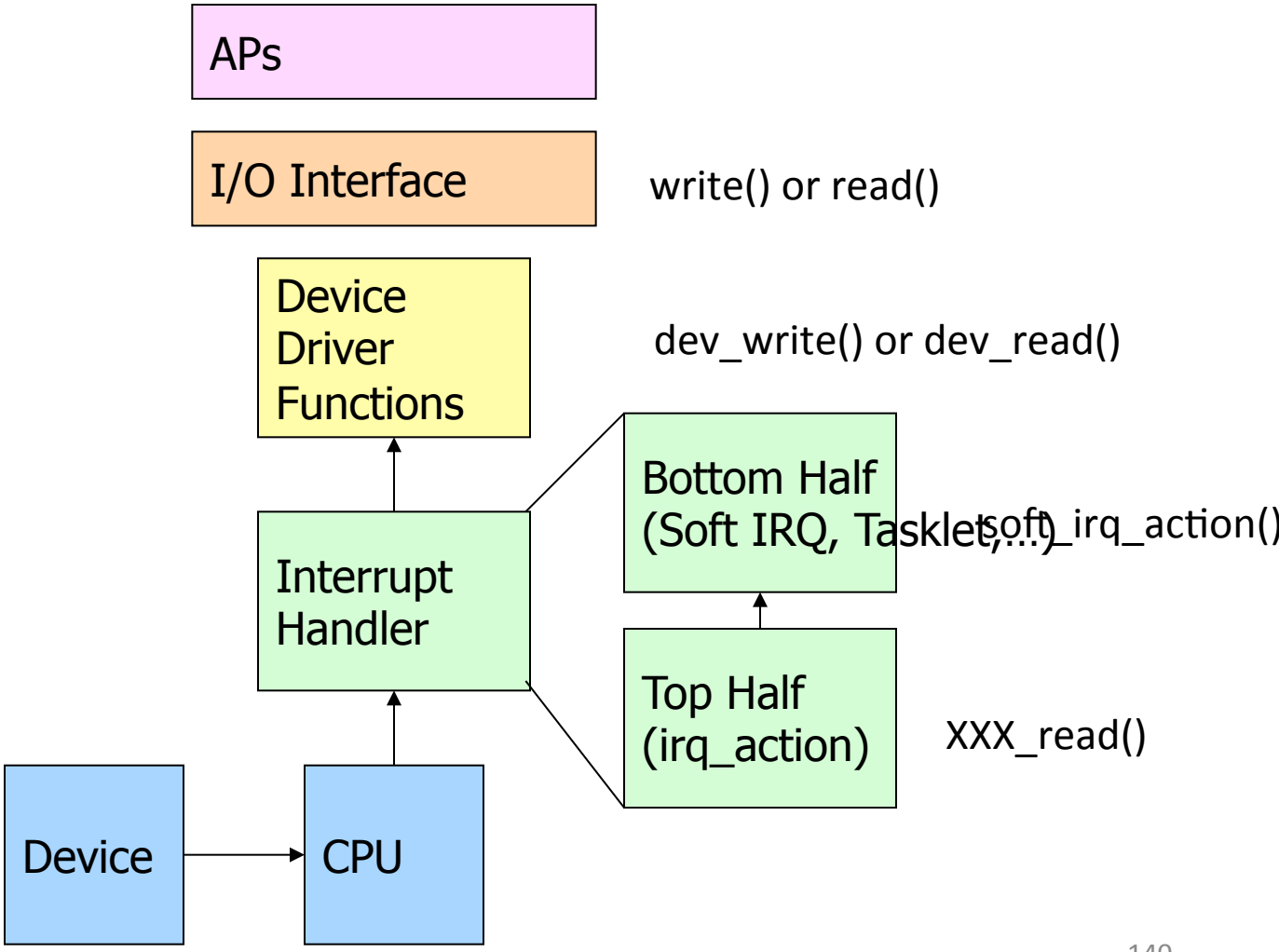


Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 3rd Edition

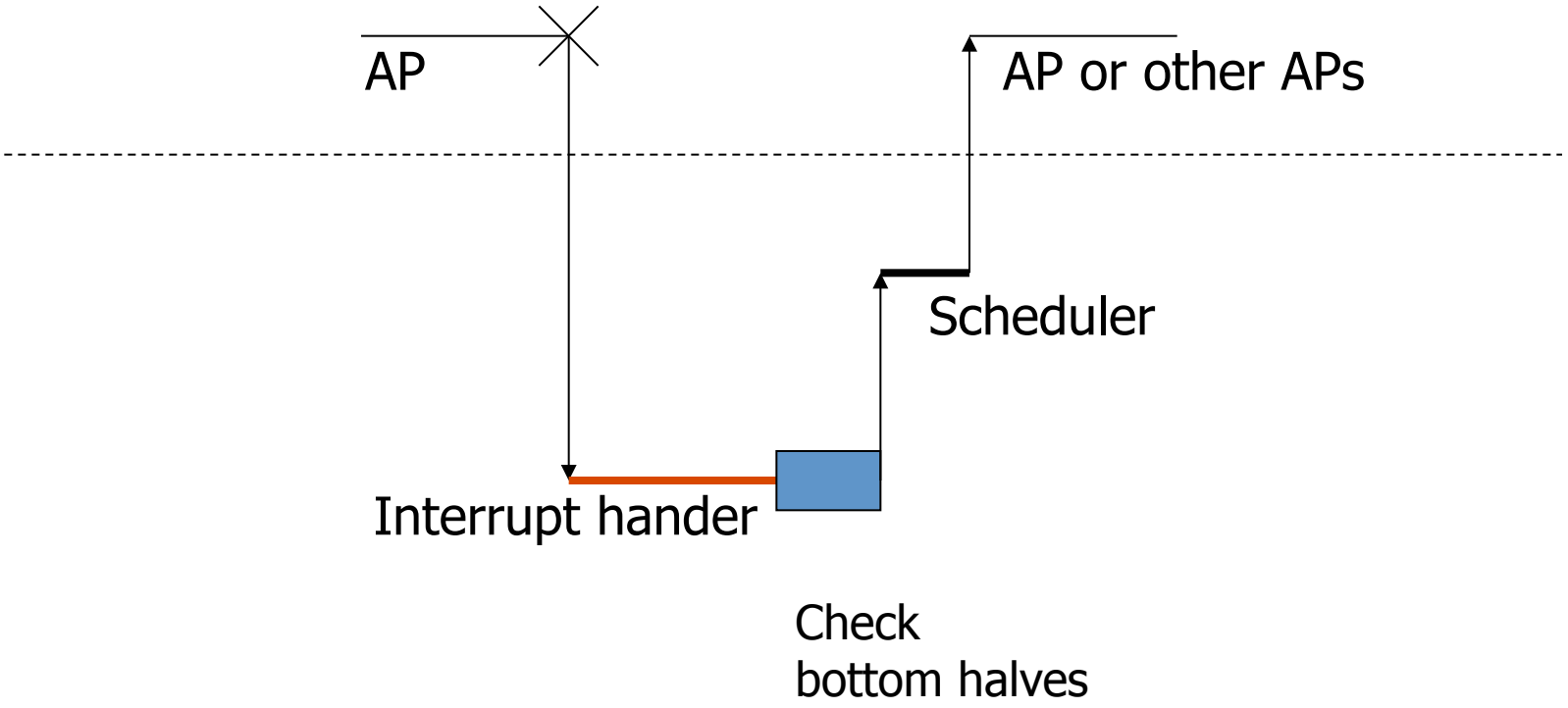
Registering Interrupt Handler



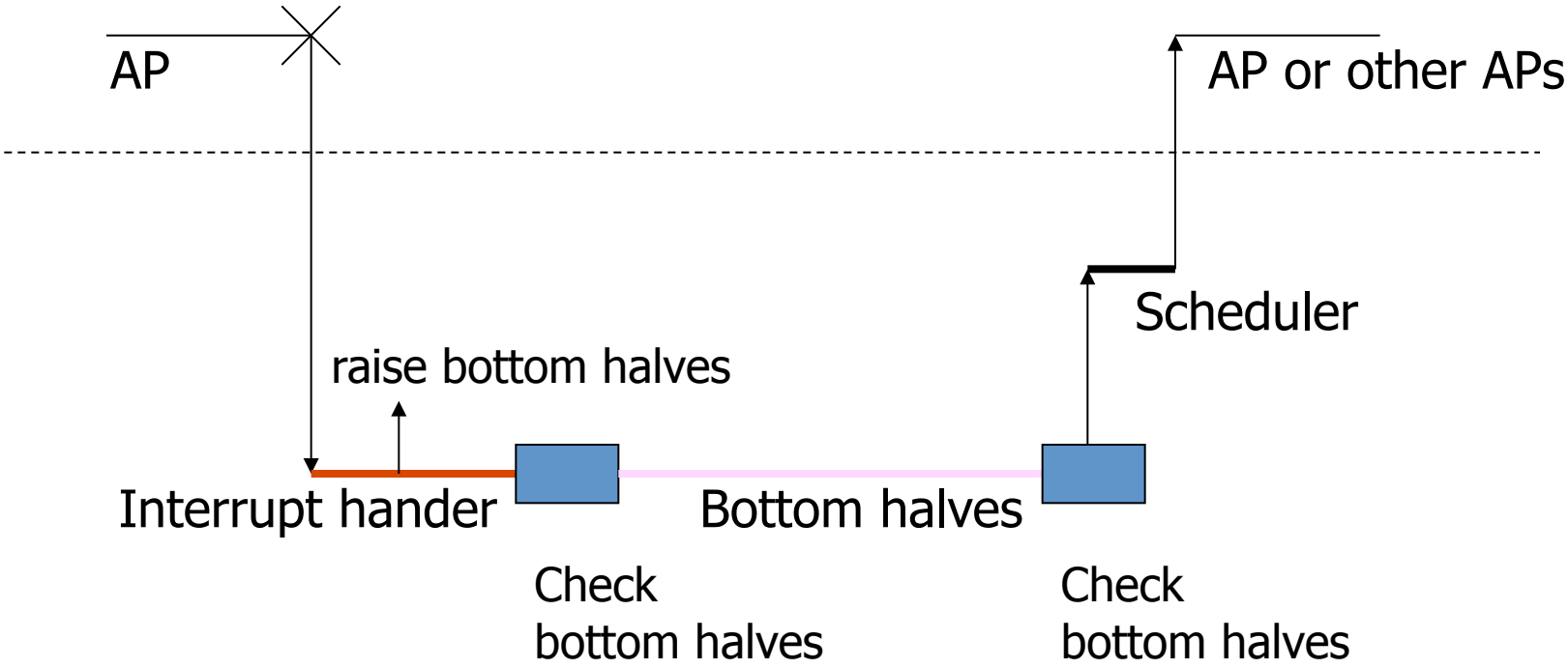
Interrupt, Interrupt Handler, and Device Driver



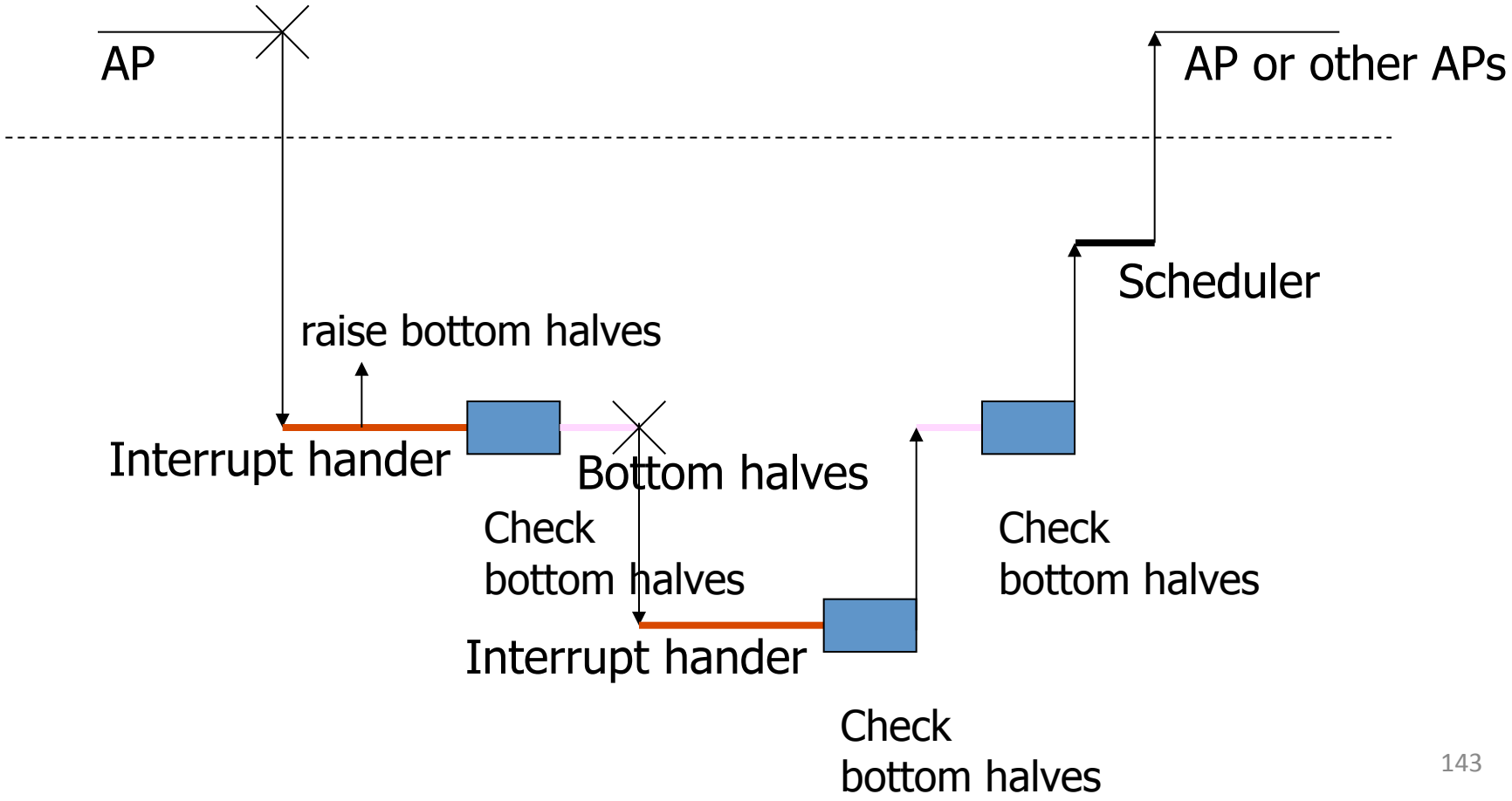
Top Halves vs. Bottom Halves



Top Halves vs. Bottom Halves



Top Halves vs. Bottom Halves



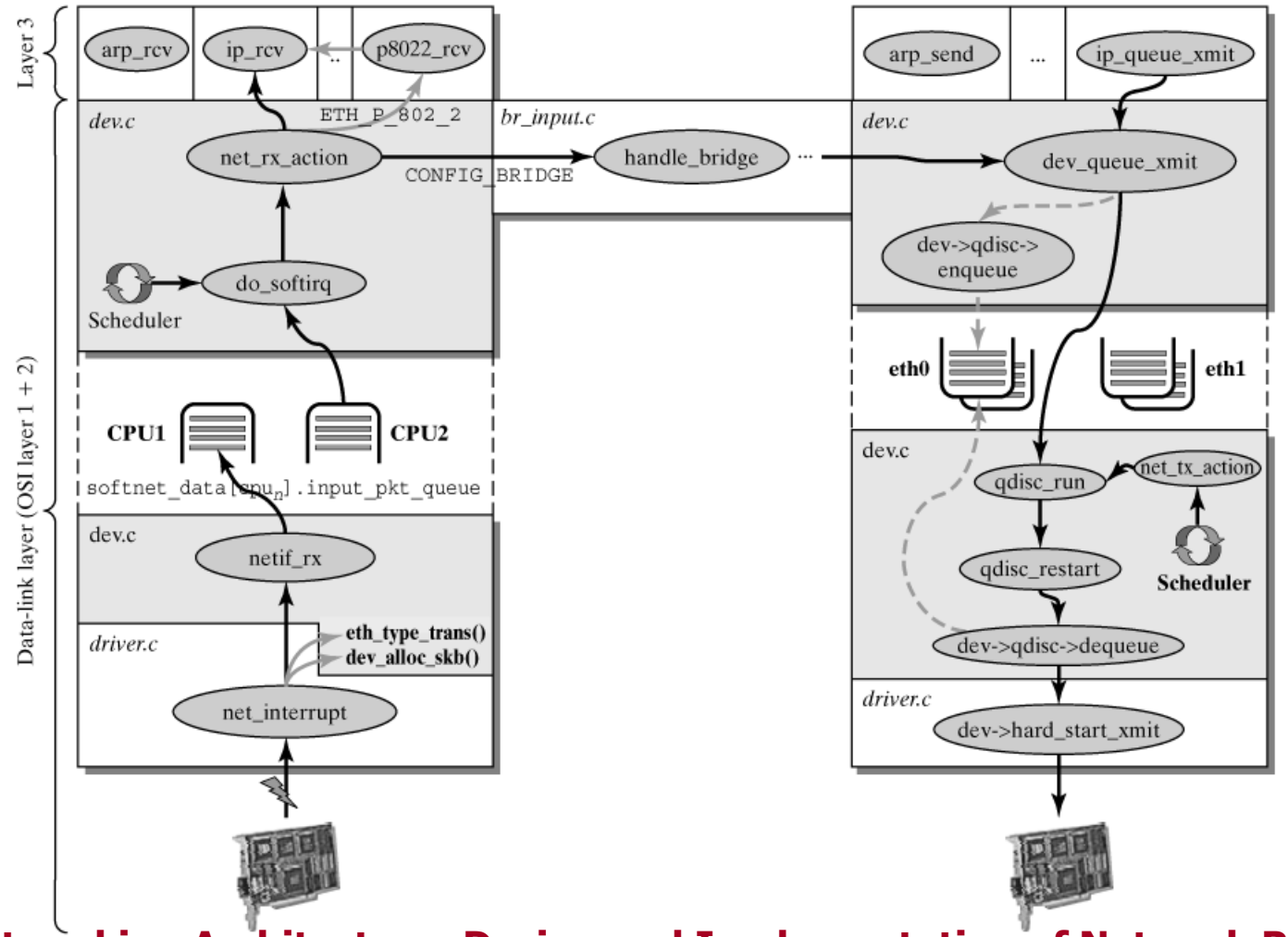
Top Halves vs. Bottom Halves

- Top halves
  - interrupt handlers (top halves), are executed by the kernel asynchronously in immediate response to a hardware interrupt
  - ASAP
- Bottom halves
  - to perform any interrupt-related work not performed by the interrupt handler itself
  - Process all interrupt related functions

Top Halves vs. Bottom Halves

Bottom Halves

Top Halves



The Linux® Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel By Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller, Marc Bechler

Interrupt vs. exception

- Interrupts:
  - Maskable interrupts: IRQ
  - Nonmaskable interrupts: reset
- Exceptions:
  - Processor-detected exceptions: divide by 0
  - Faults: page faults
  - Traps: debug (no need to execute old instruction)
  - Aborts: hardware failure
  - Programmed exceptions: software interrupt

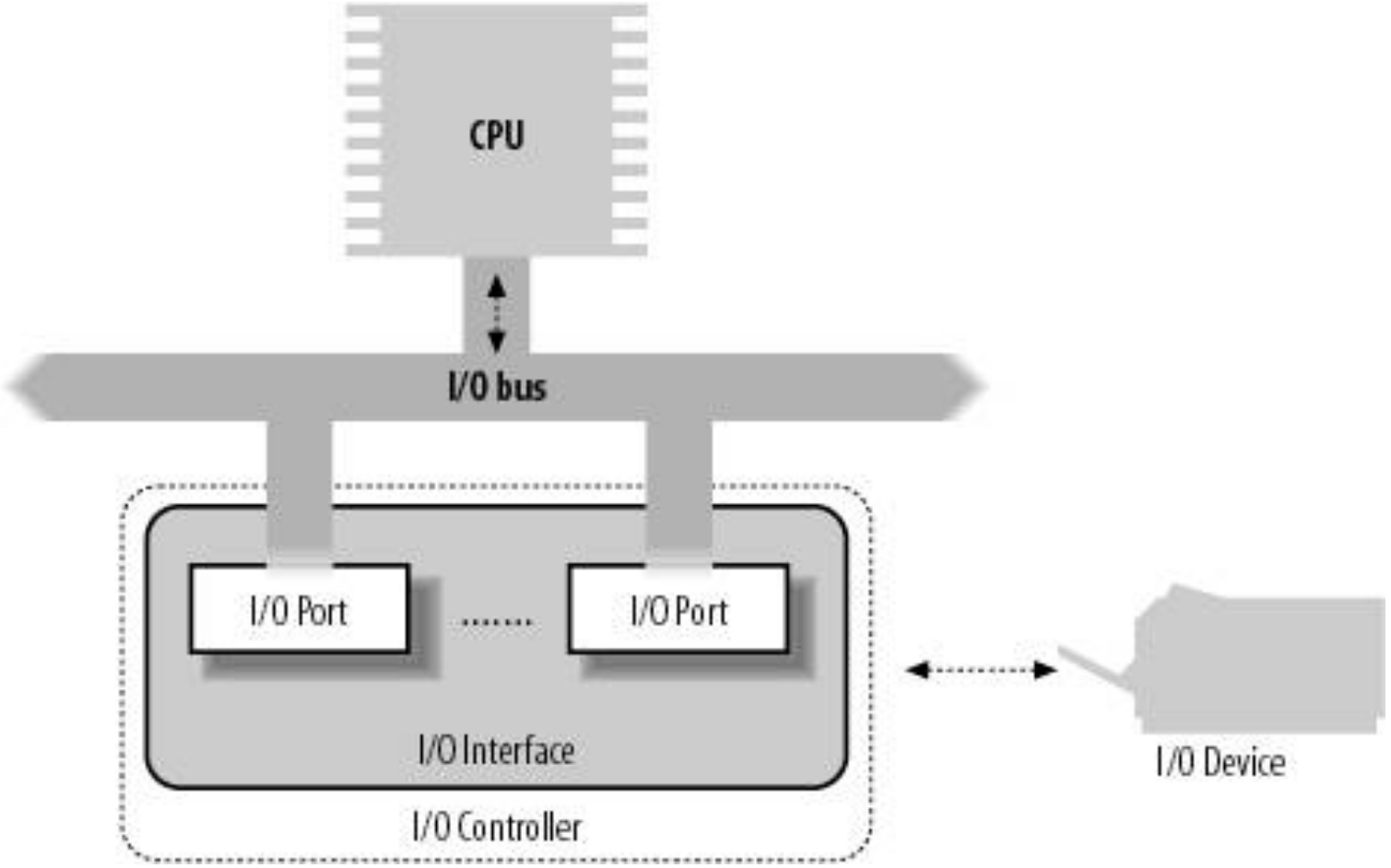
Linux Device Driver

Outline

- Basic Concept of Device Driver
- Linux Implementation
  - IO subsystem
  - Module
  - Characteristic device driver
  - Block device driver
  - Network device driver

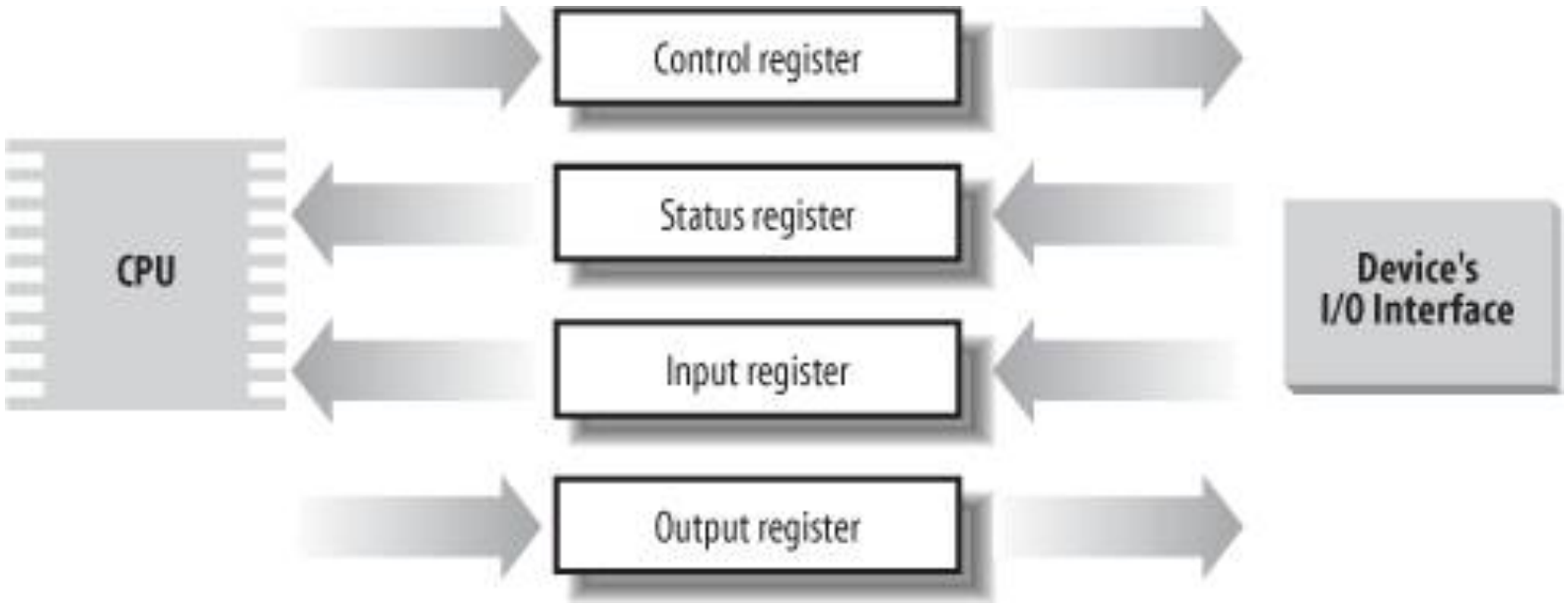
Basic Concept of Device Driver

Connect interrupt and device driver



Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 3rd Edition

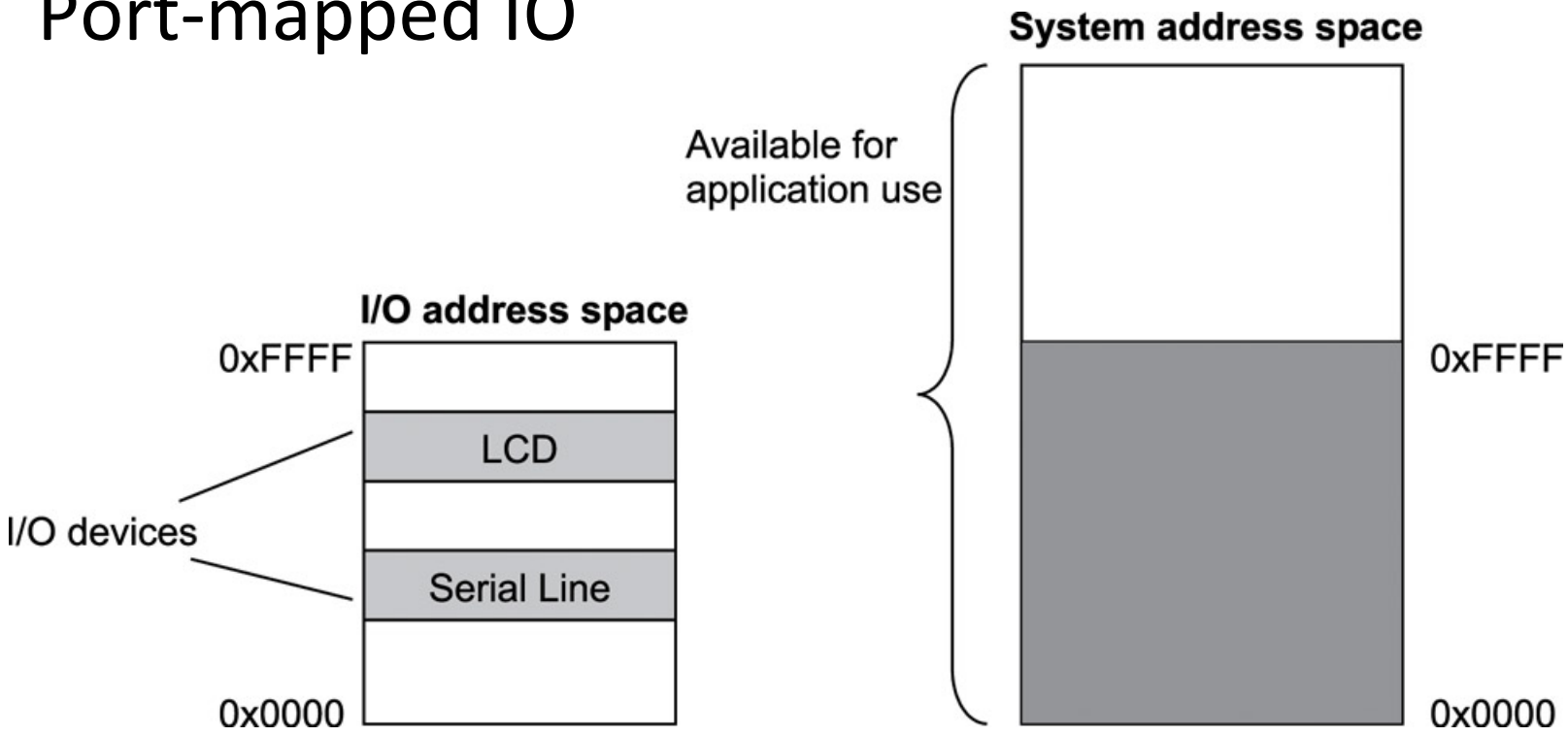
Connect interrupt and device driver



Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 3rd Edition

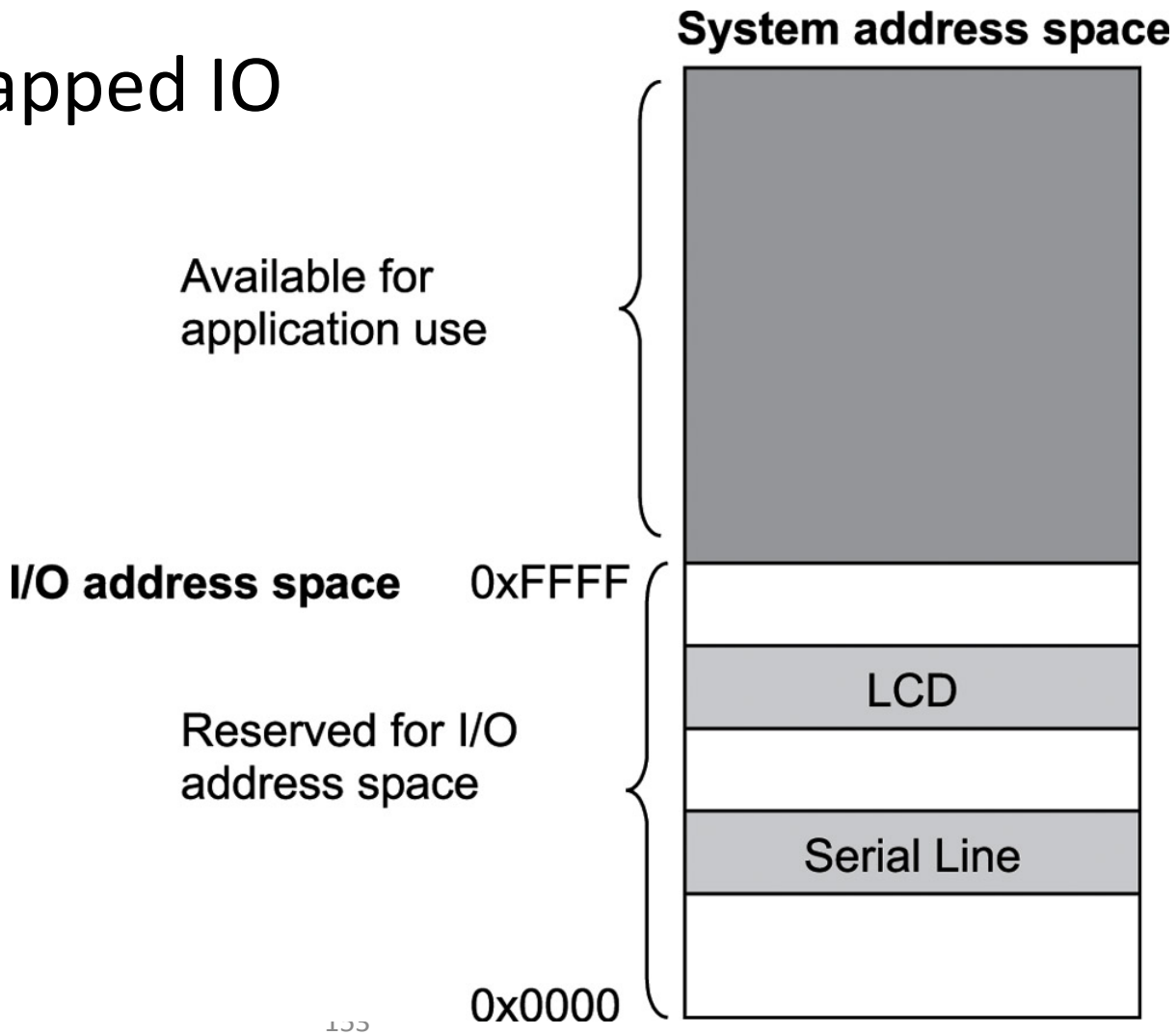
Device Driver Basics

- Port-mapped IO



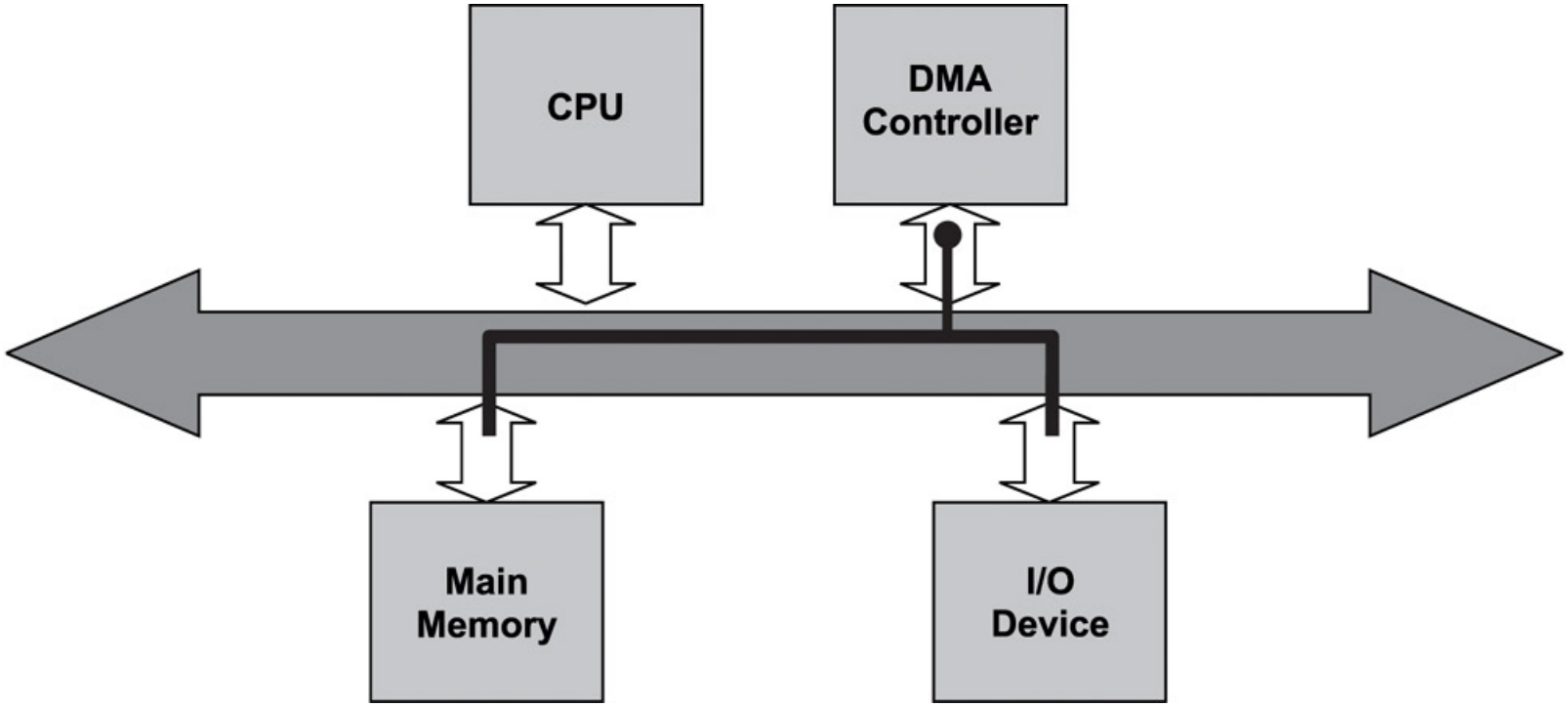
Device Driver Basics

- Memory-mapped IO

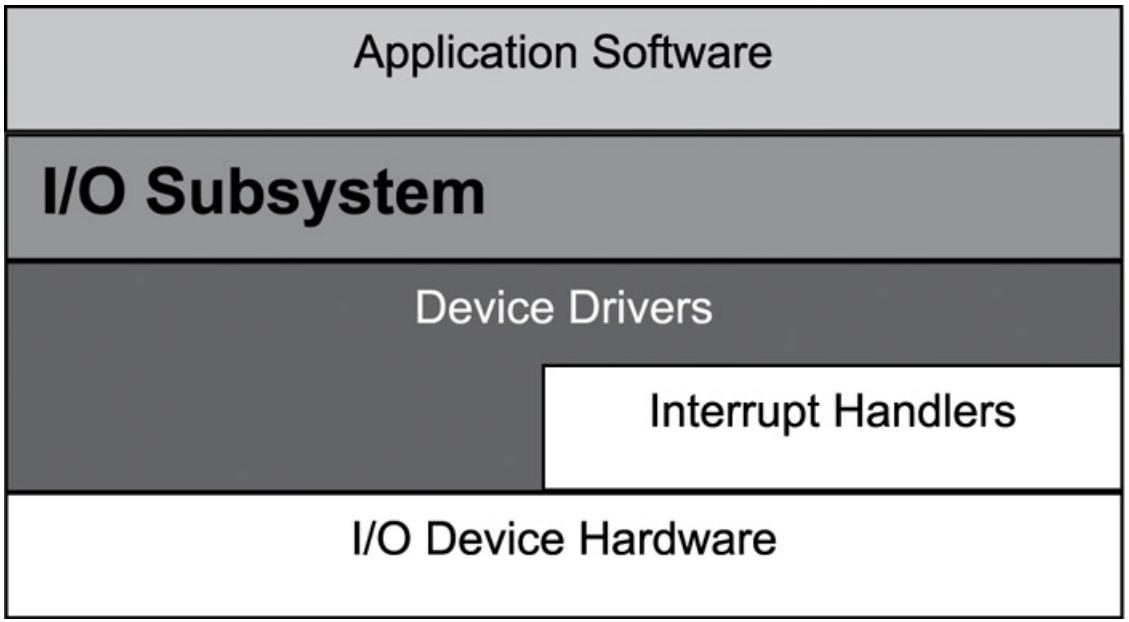


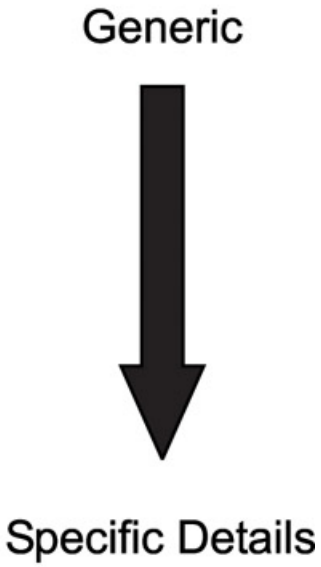
"Real-Time Concepts for Embedded Systems," by Qing Li and Carolyn Yao

Device Driver Basics



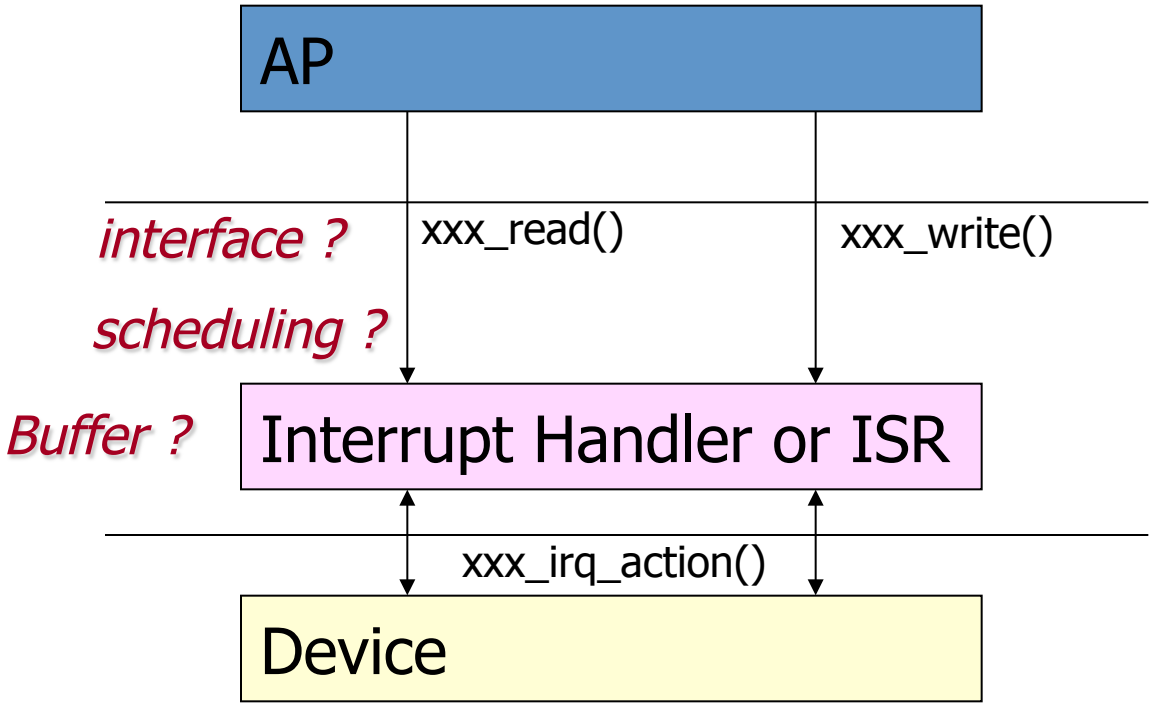
Device Driver Basics





Connect interrupt and device driver

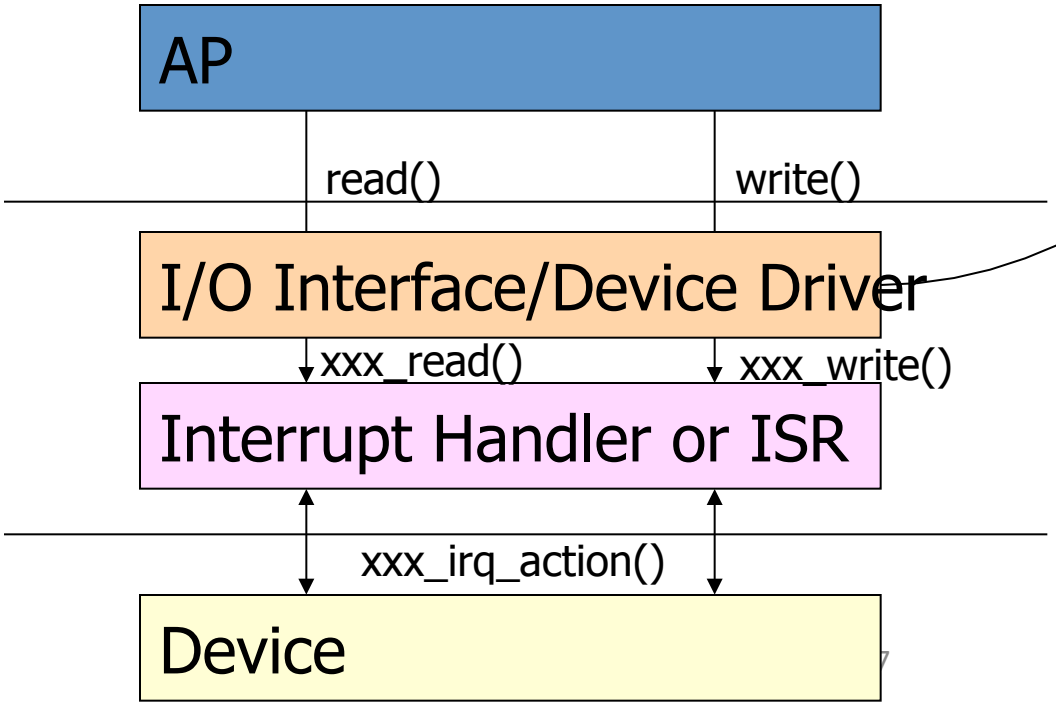
- AP + interrupt service routine



Connect interrupt and device driver

- AP + device driver + interrupt service routine

device file	open	read	write	close	...
tty0	tty0_open()	tty0_read()	tty0_write()	tty0_close()	
lp	lp_open()	lp_read()	lp_write()	lp_close()	...
...	...	...	...	...	...

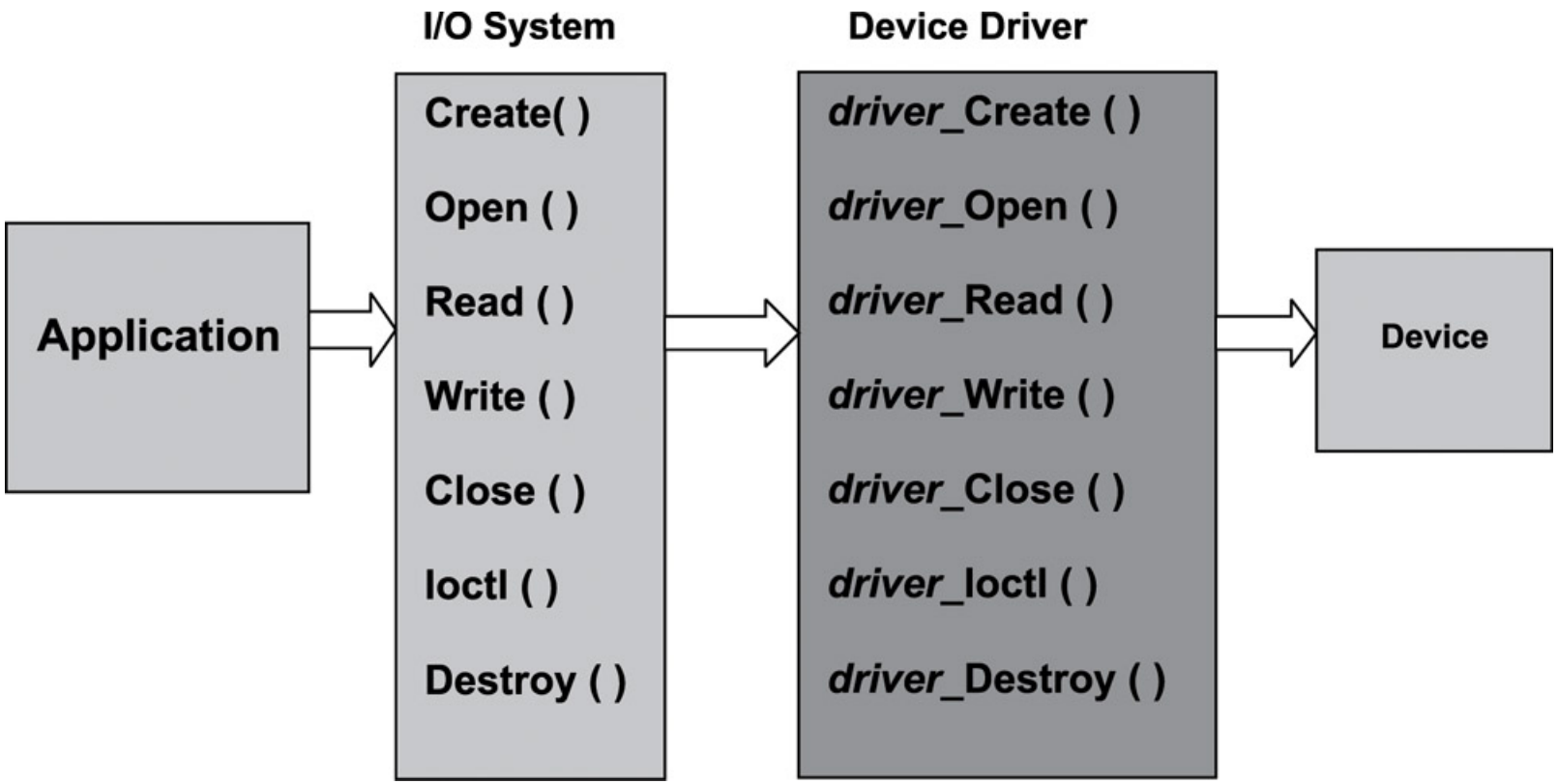


Connect interrupt and device driver

- Device file concept
  - Old-style device file

Name	Type	Major	Minor	Description
/dev/fd0	block	2	0	Floppy disk
/dev/hda	block	3	0	First IDE disk
/dev/hda2	block	3	2	Second primary partition of first IDE disk
/dev/hdb	block	3	64	Second IDE disk
/dev/hdb3	block	3	67	Third primary partition of second IDE disk
/dev/ttyp0	char	3	0	Terminal
/dev/console	char	5	1	Console
/dev/lp1	char	6	1	Parallel printer
/dev/ttyS0	char	4	64	First serial port
/dev/rtc	char	10	135	Real-time clock
/dev/null	char	1	3	Null device (black hole)

Device Driver Basics



"Real-Time Concepts for Embedded Systems," by Qing Li and Carolyn Yao

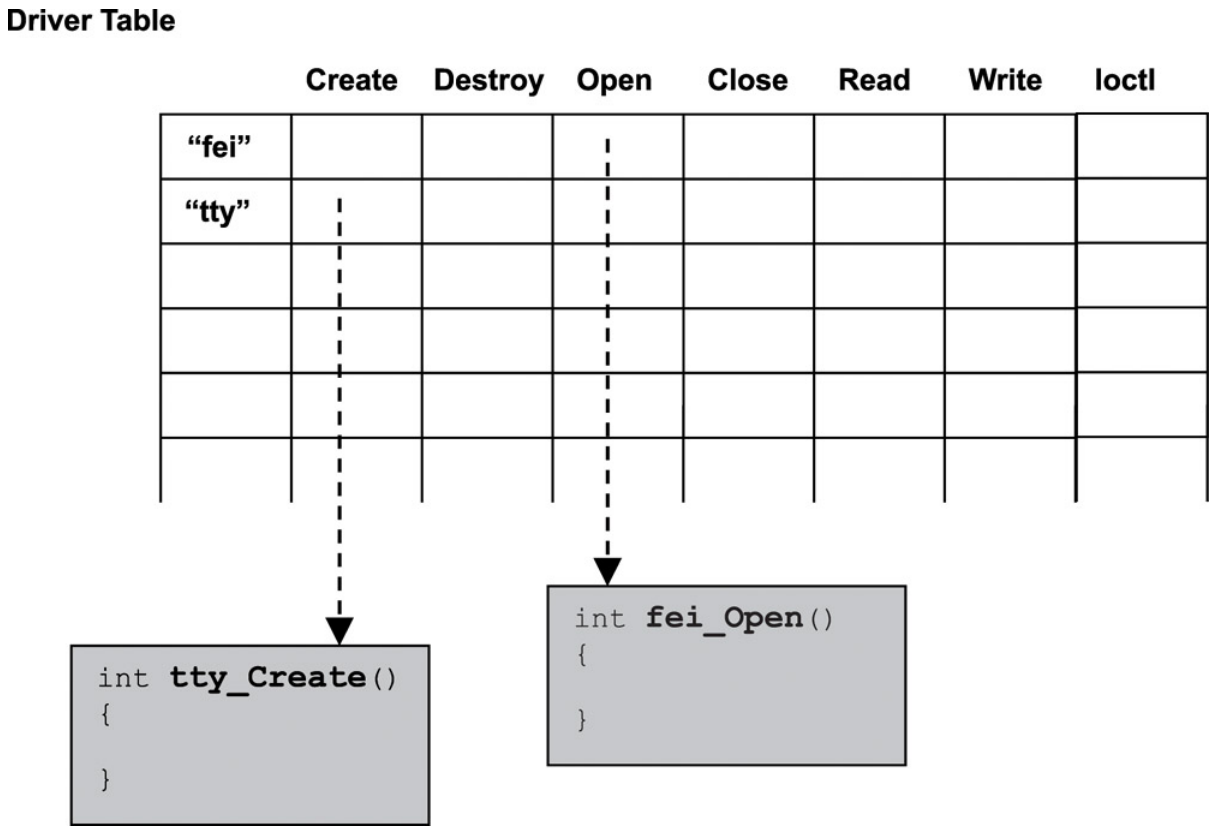
I/O subsystem (Cont.)

- Mapping generic functions to driver functions

```
typedef struct
{
        int (*Create)( );
        int (*Open) ( );
        int (*Read)( );
        int (*Write) ( );
        int (*Close) ( );
        int (*Ioctl) ( );
        int (*Destroy) ( );
} UNIFORM_IO_DRV;
```

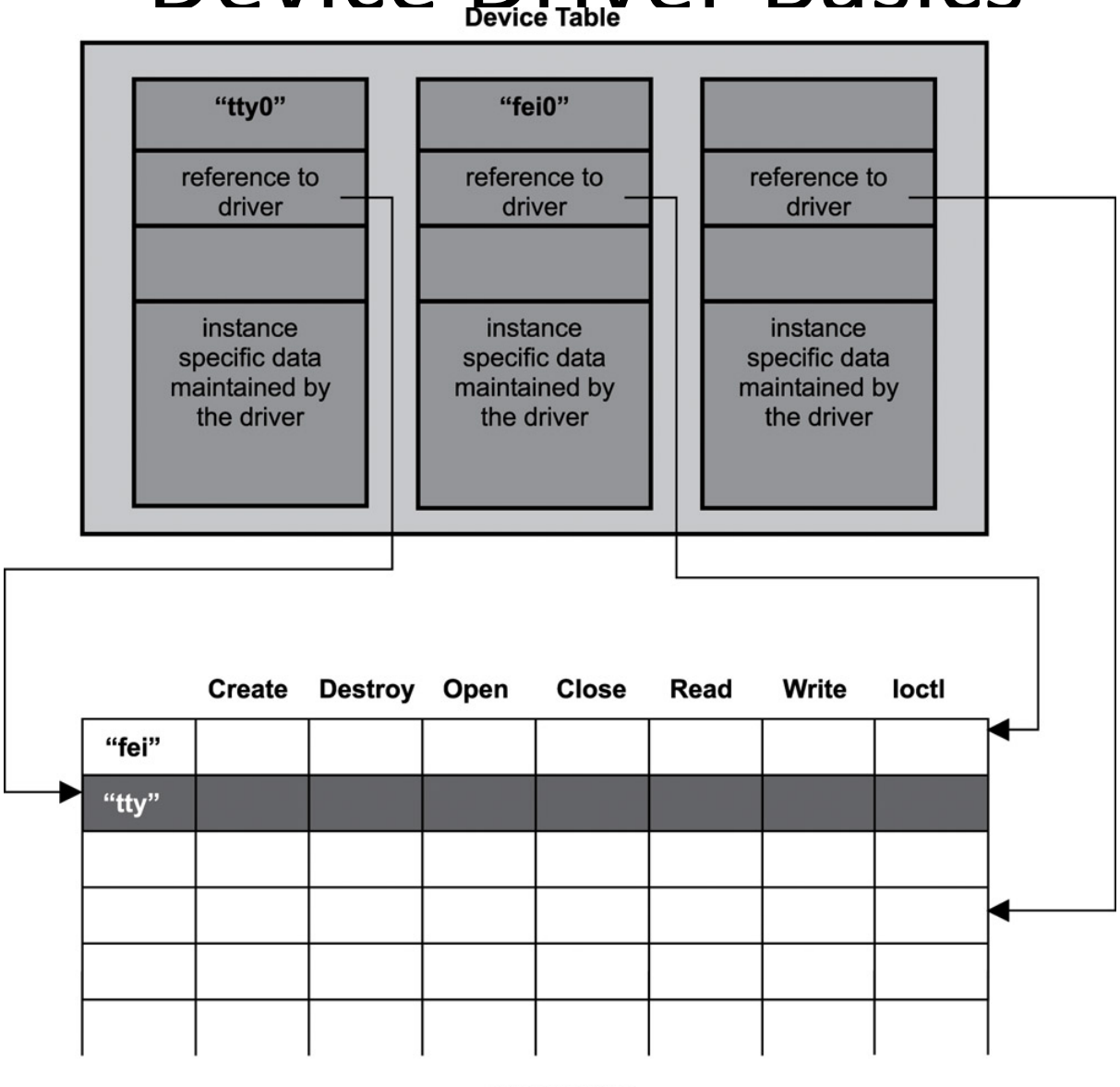
```
UNIFORM_IO_DRV ttyIOdrv;
ttyIOdrv.Create = tty_Create;
ttyIOdrv.Open = tty_Open;
ttyIOdrv.Read = tty_Read;
ttyIOdrv.Write = tty_Write;
ttyIOdrv.Close = tty_Close;
ttyIOdrv.Ioctl = tty_Ioctl;
ttyIOdrv.Destroy = tty_Destroy;
```

Device Driver Basics



"Real-Time Concepts for Embedded Systems," by Qing Li and Carolyn Yao

Device Driver Basics



"Real-Time Concepts for Embedded Systems," by Qing Li and Carolyn Yao

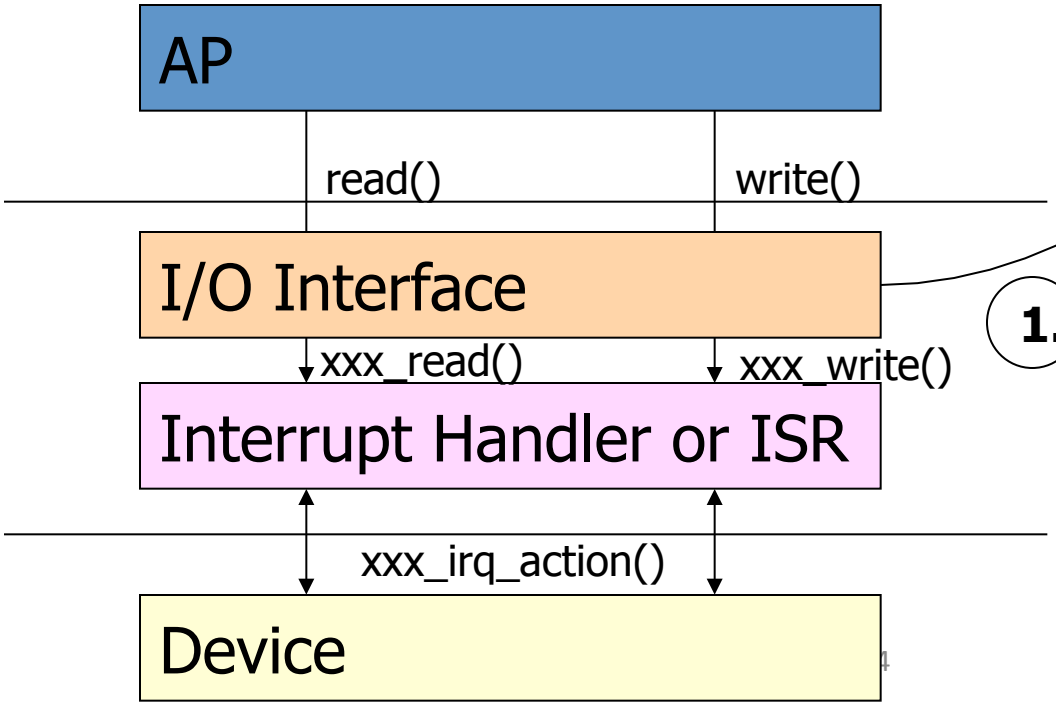
Connect interrupt and device driver

- Level of kernel support
  - No support at all
    - The application program interacts directly with the device's I/O ports by issuing suitable in and out assembly language instructions.
  - Minimal support
    - The kernel does not recognize the hardware device, but does recognize its I/O interface. User programs are able to treat the interface as a sequential device capable of reading and/or writing sequences of characters.
  - Extended support
    - The kernel recognizes the hardware device and handles the I/O interface itself. In fact, there might not even be a device file for the device.

Connect interrupt and device driver

- Registering a device driver

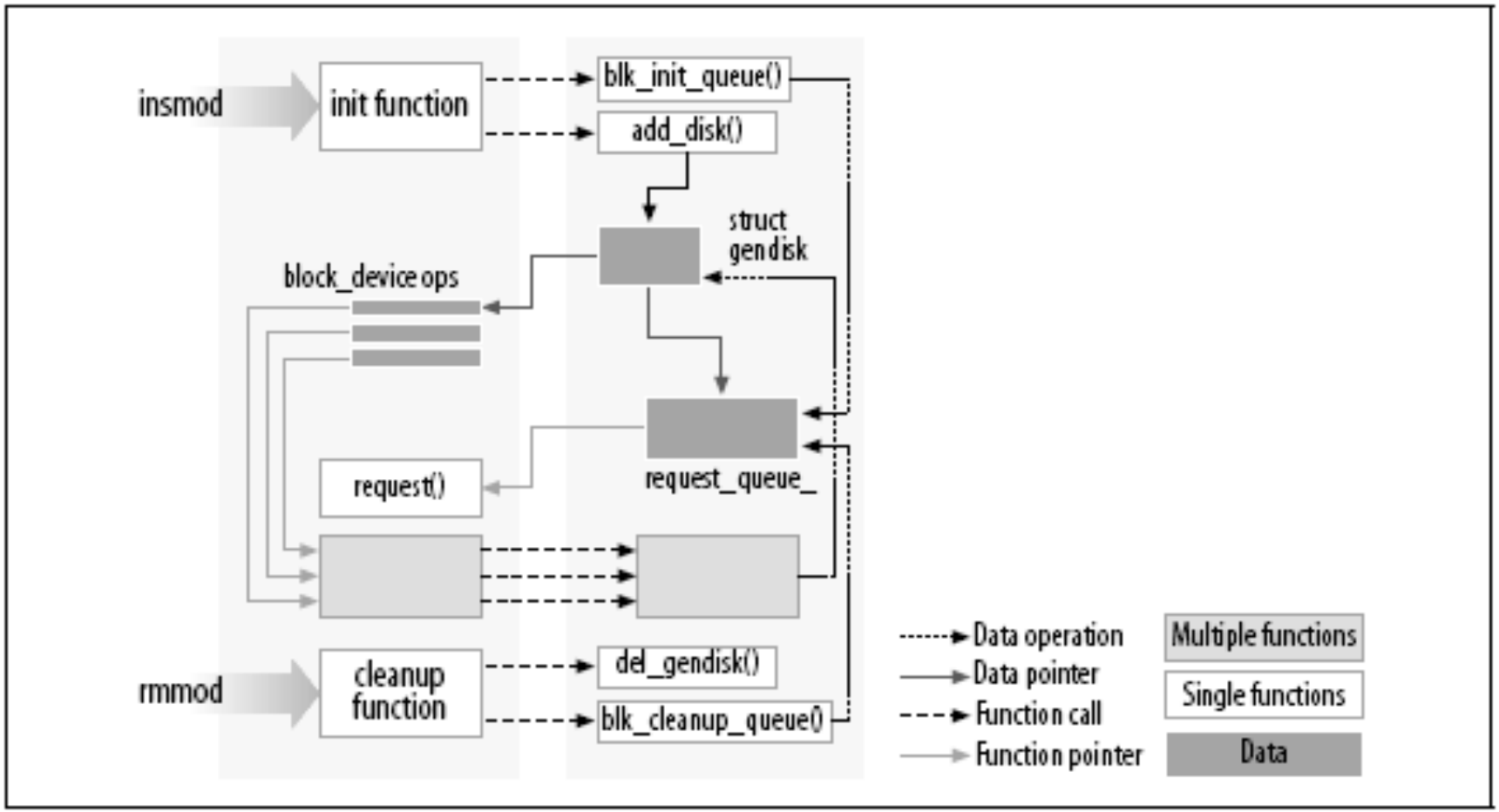
device file	open	read	write	close	...
tty0	tty0_open()	tty0_read()	tty0_write()	tty0_close()	
lp	lp_open()	lp_read()	lp_write()	lp_close()	...
...	...	...	...	...	...



2. Mapping interface

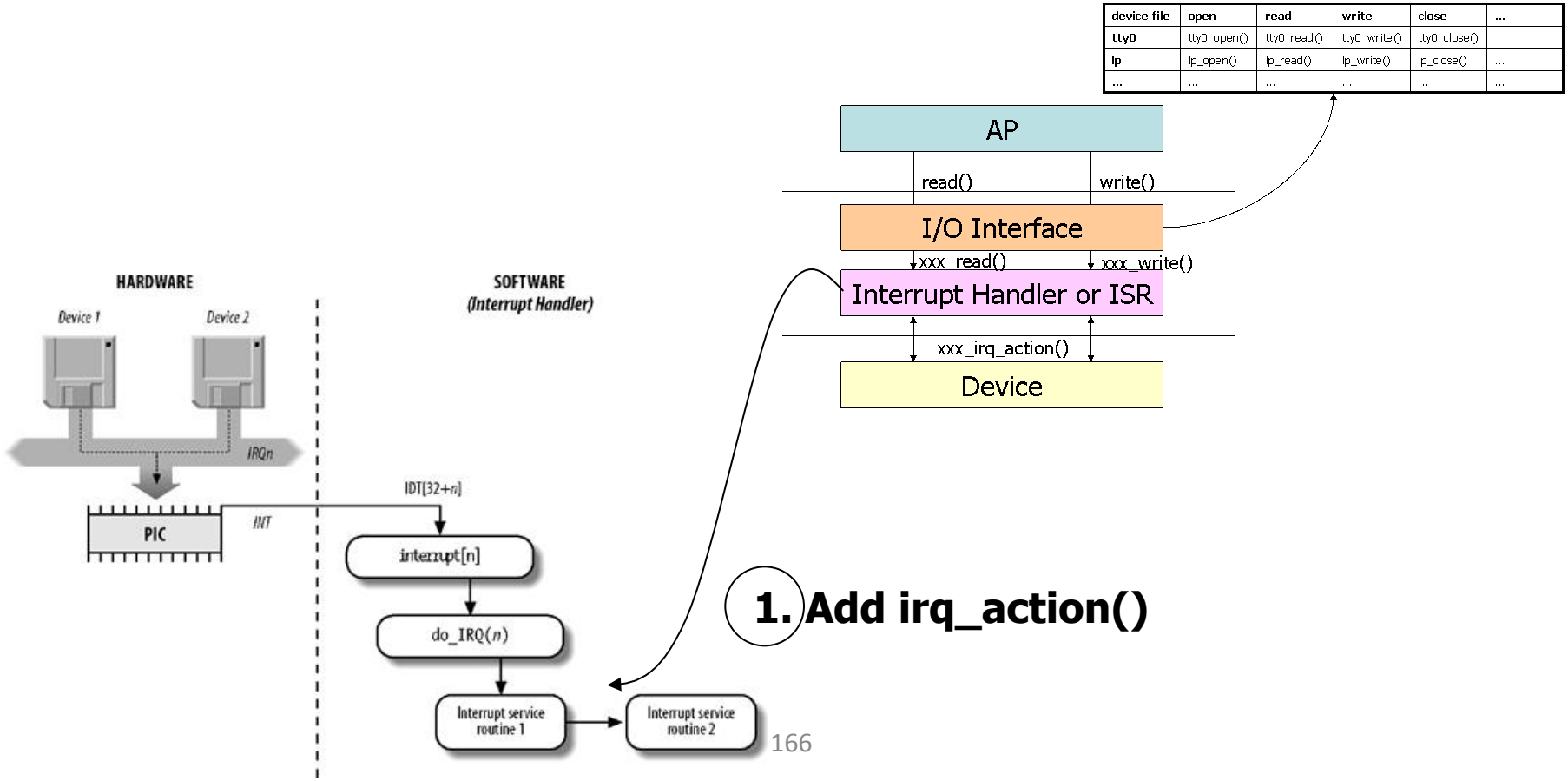
1. Register device driver

Connect interrupt and device driver



Connect interrupt and device driver

- Initializing a device driver



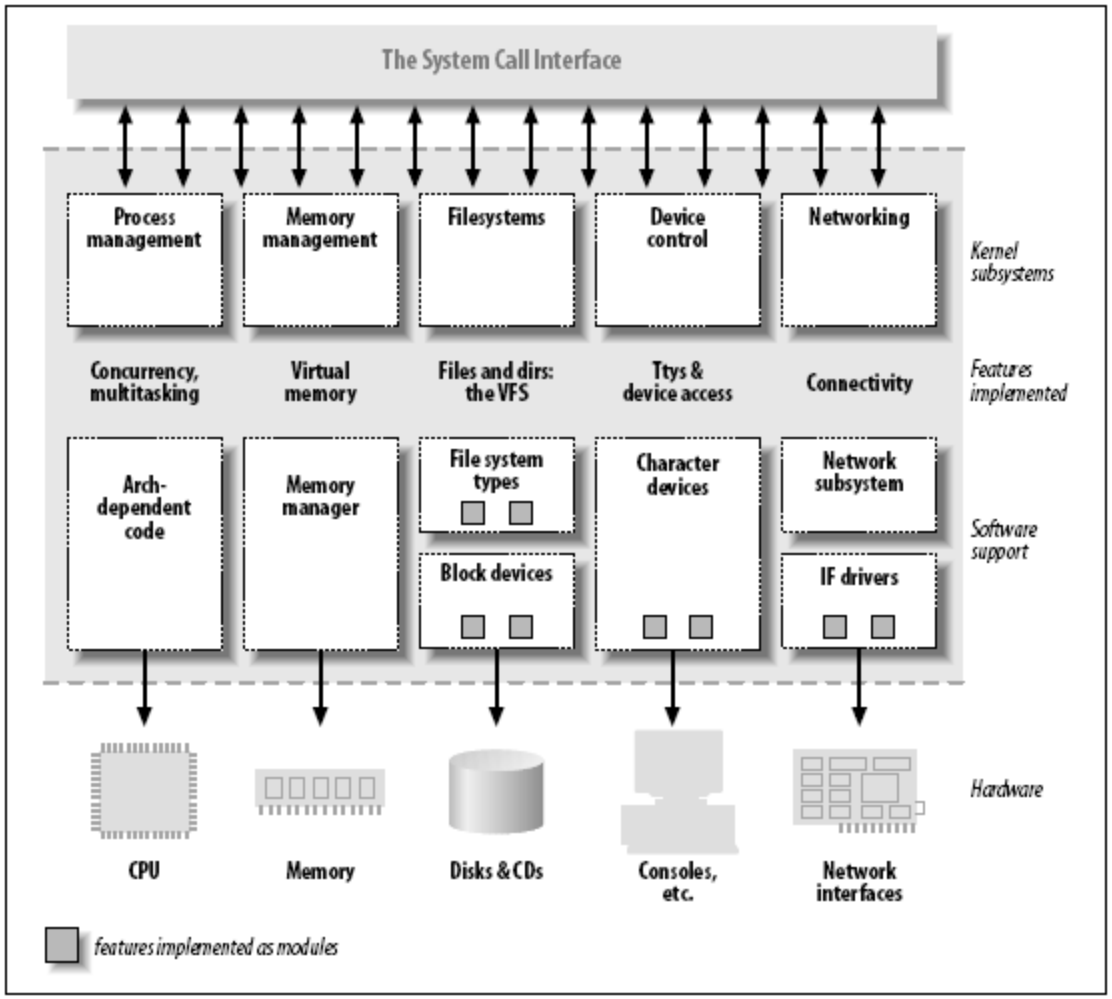
Device Driver Basics

- Character-mode devices
  - for unstructured data transfers
  - data transfers typically take place in serial fashion
  - usually simple devices

- Block-mode devices
  - transfer data one block at time
  - underlying hardware imposes the block size
  - structure must be imposed on the data

Linux Implementation

Linux I/O Architecture



Device categories

- Character devices
  - can be accessed as a stream of bytes
  - implements at least the open, close, read, and write system calls
  - The text console (/dev/console) and the serial ports (/dev/ttyS0 and friends) are examples

Device categories

- Block devices
  - A block device is a device (e.g., a disk) that can host a filesystem
  - a block device can only handle I/O operations that transfer one or more whole blocks
  - Block drivers have a completely different interface to the kernel than char drivers

Device categories

- Network interfaces
  - is able to exchange data with other hosts
  - still by assigning a unique name to them (such as eth0), but that name doesn't have a corresponding entry in the filesystem
  - Communication between the kernel and a network device driver is completely different from that used with char and block drivers

Register device drivers

- 2.4
  - register_chrdev()
  - register_blkdev()
  - Example: register_chrdev(6, “lp”, &lp_fops)
- 2.6
  - device_register( )
  - driver_register( )

Device file operations

```
struct file_operations scull_fops = {
    .owner =    THIS_MODULE,
    .llseek =   scull_llseek,
    .read =     scull_read,
    .write =    scull_write,
    .ioctl =    scull_ioctl,
    .open =     scull_open,
    .release =  scull_release,
};
```

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

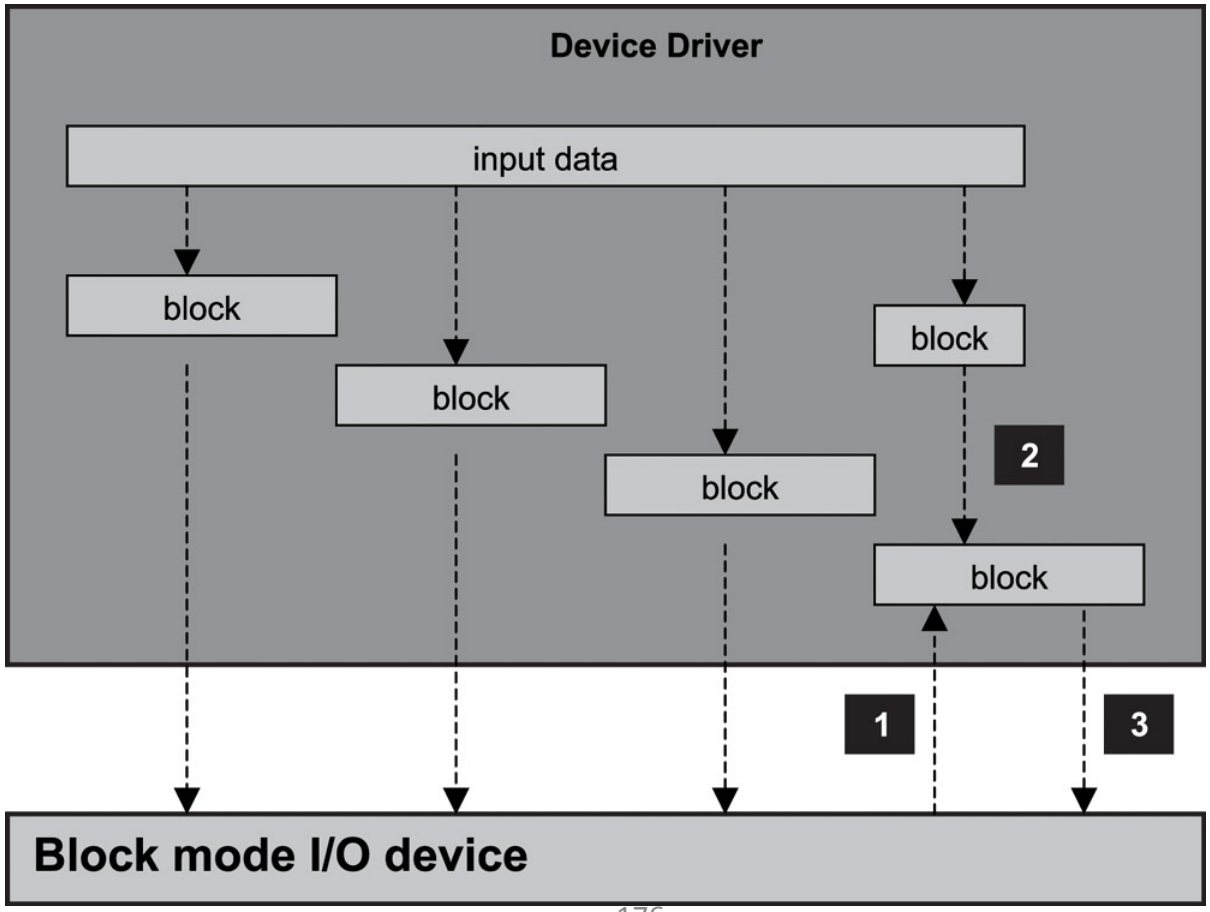
```
struct scull_dev {
    struct scull_qset *data;  /* Pointer to first quantum set */
    int quantum;              /* the current quantum size */
    int qset;                 /* the current array size */
    unsigned long size;       /* amount of data stored here */
    unsigned int access_key;  /* used by sculluid and scullpriv */
    struct semaphore sem;     /* mutual exclusion semaphore     */
    struct cdev cdev;     /* Char device structure      */
};
```

Device file operations

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
    int err, devno = MKDEV(scull_major, scull_minor + index);

    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Fail gracefully if need be */
    if (err)
    printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}

int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* device information */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* for other methods */

    /* now trim to 0 the length of the device if open was write-only */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
        scull_trim(dev); /* ignore errors */
    }
    return 0;          /* success */
}
```

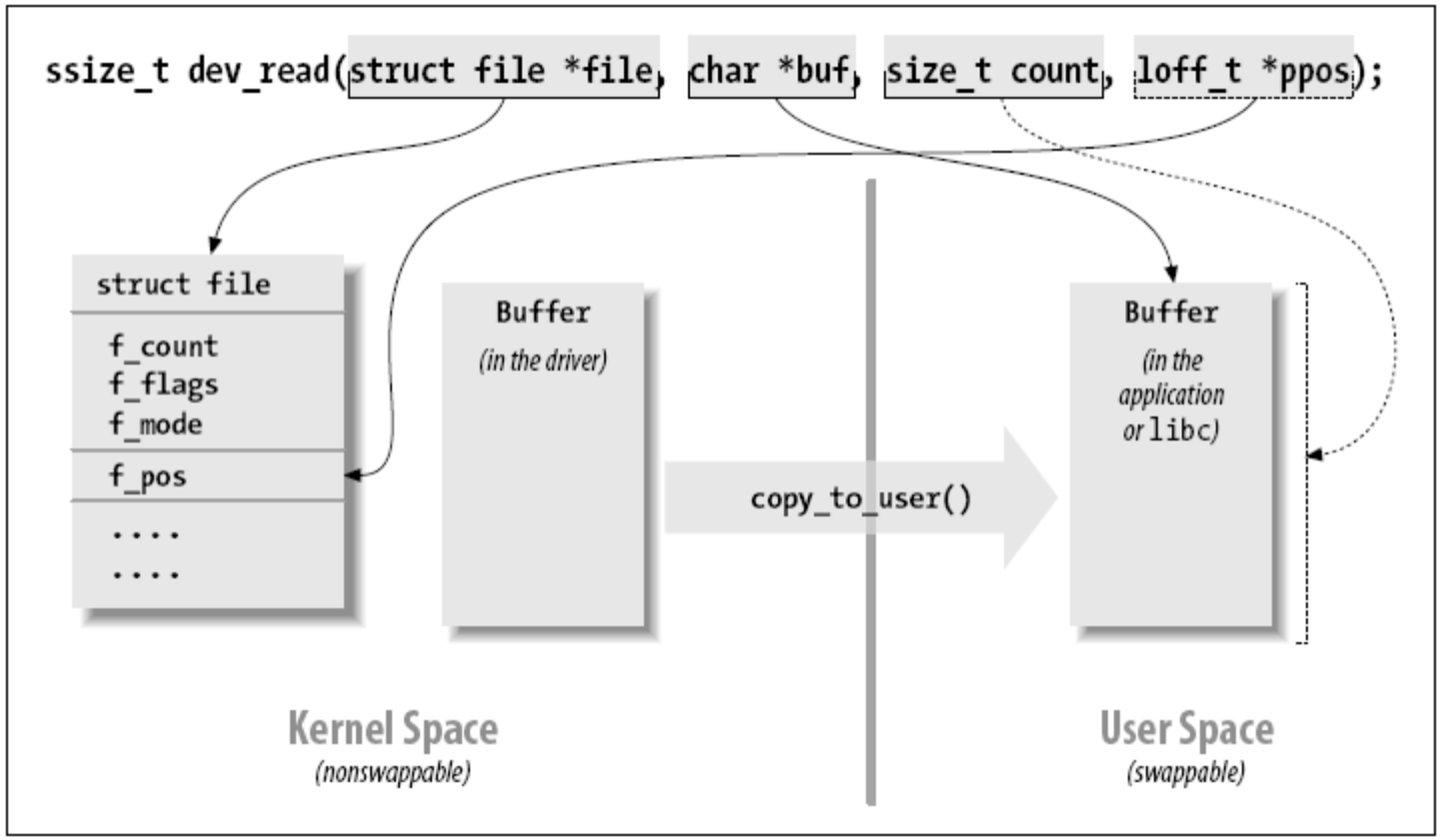
Device Driver Basics

- Block mode device



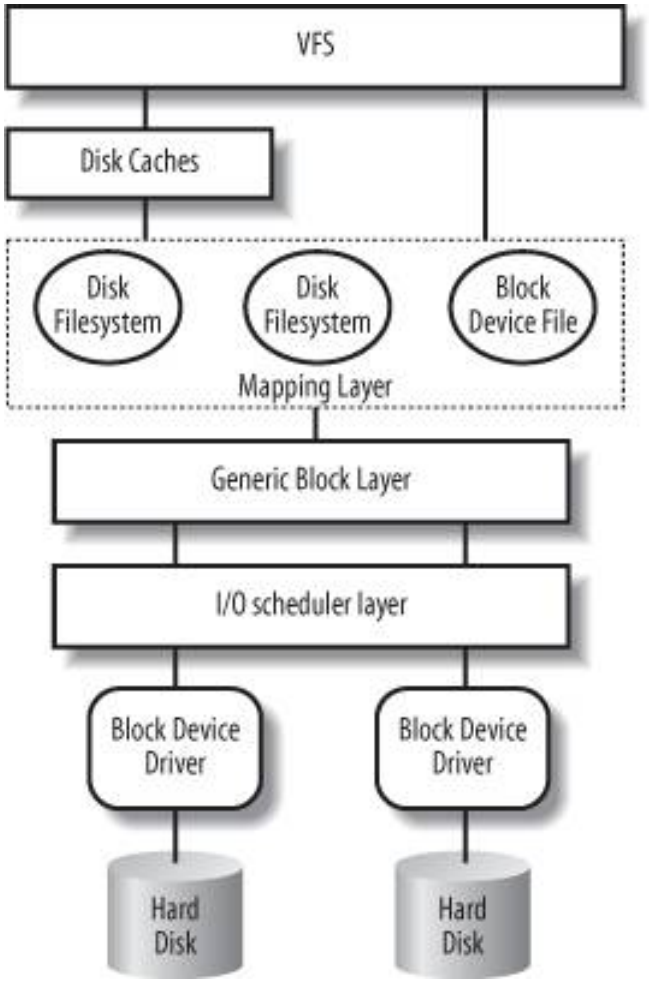
"Real-Time Concepts for Embedded Systems," by Qing Li and Carolyn Yao

Memory copy


ssize_t dev_read(struct file *file, char *buf, size_t count, loff_t *ppos);
struct file
f_count
f_flags
f_mode
f_pos
....
....
Buffer
(in the driver)
copy_to_user()
Buffer
(in the application or libc)
Kernel Space
(nonswappable)
User Space
(swappable)

Block device driver

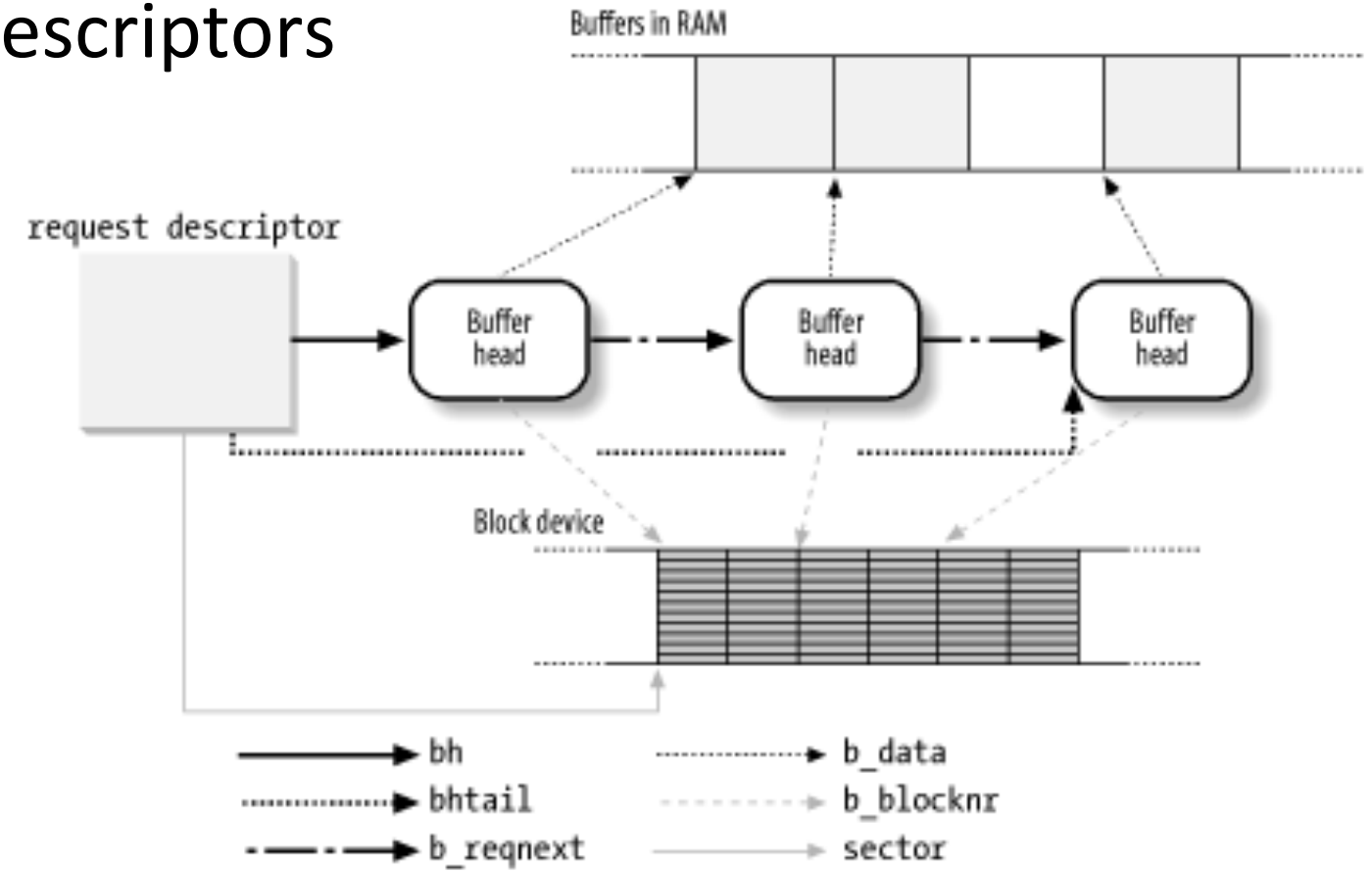
- Architecture



Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 3rd Edition

Block device driver

- Request descriptors, and request queue descriptors



Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel," 2nd Edition