Overview

- This Lecture
  - Interrupts and exceptions
  - Source: ULK ch 4, ELDD ch1, ch2 & ch4

Three reasons for interrupts

- System calls
- Program/hardware faults
- External device interrupts
- Note interrupts, exceptions and traps are similar and very often confusing due to their subtle differences
  - We use interrupts to include them all here.

Event-driven programming

- OS is largely based on event-driven programming for handling interrupts
- Event handlers are important parts of OS
  - Interrupt handlers, exception handlers
  - If no one kicks the ball, nothing will happen after kernel initialization
- Who kicks the ball?
  - Init, keyboard, mouse, etc

Why care about interrupts?

- Security and isolation
  - Only kernel can access devices, MMU, FS, etc
  - User program is a potential malicious adversary
- Resource sharing
  - Centralized handling facilitates sharing
- Timely response to urgent events
- Continuation of interrupted program

User & kernel spaces

- User space and kernel space
  - Different privilege level
  - Different memory mapping
  - Different set of libraries
- User space to kernel space
  - System call: for the current process
  - Interrupt: may not be relevant to the current process

How to handle interrupts?

- Save program state for future restoration
- Set up for execution in kernel (stack, segment)
- Choose the interrupt handler
- Get arguments for system calls
- Ensure security (more detail later)

ARM interrupts

- IRQ: for devices
- FIQ: for one fast device
- Abort exception: a prefetch abort or data abort exception occurs.
- Undefined exception: an undefined instruction exception occurs.

System call

- Use swi or svc to switch from user mode to kernel mode (i.e. SVC mode)
- Parameters in xv6
  - Syscall number is passed with r0
  - The number of parameters are unlimited for syscalls (verify for yourselves:-)
  - Parameters are passed with the user stack
- Return value in xv6
  - The return value of the syscall is in r0

The Linux kernel

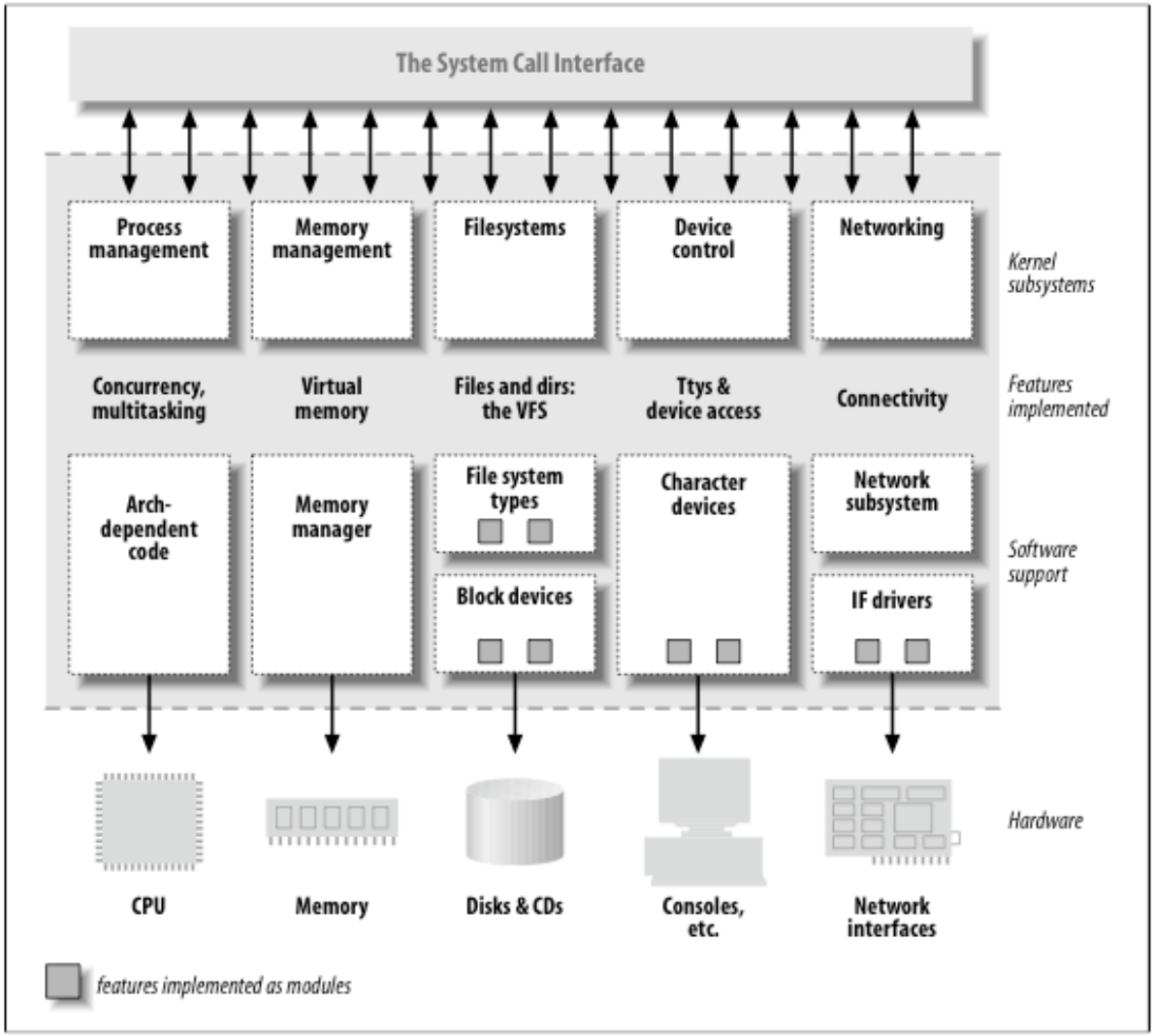

Figure 1-1. A split view of the kernel

Device interrupt handling

- Mapping between IRQ and interrupt vector
  - INT = IRQ + 32 (Intel default, allow change)

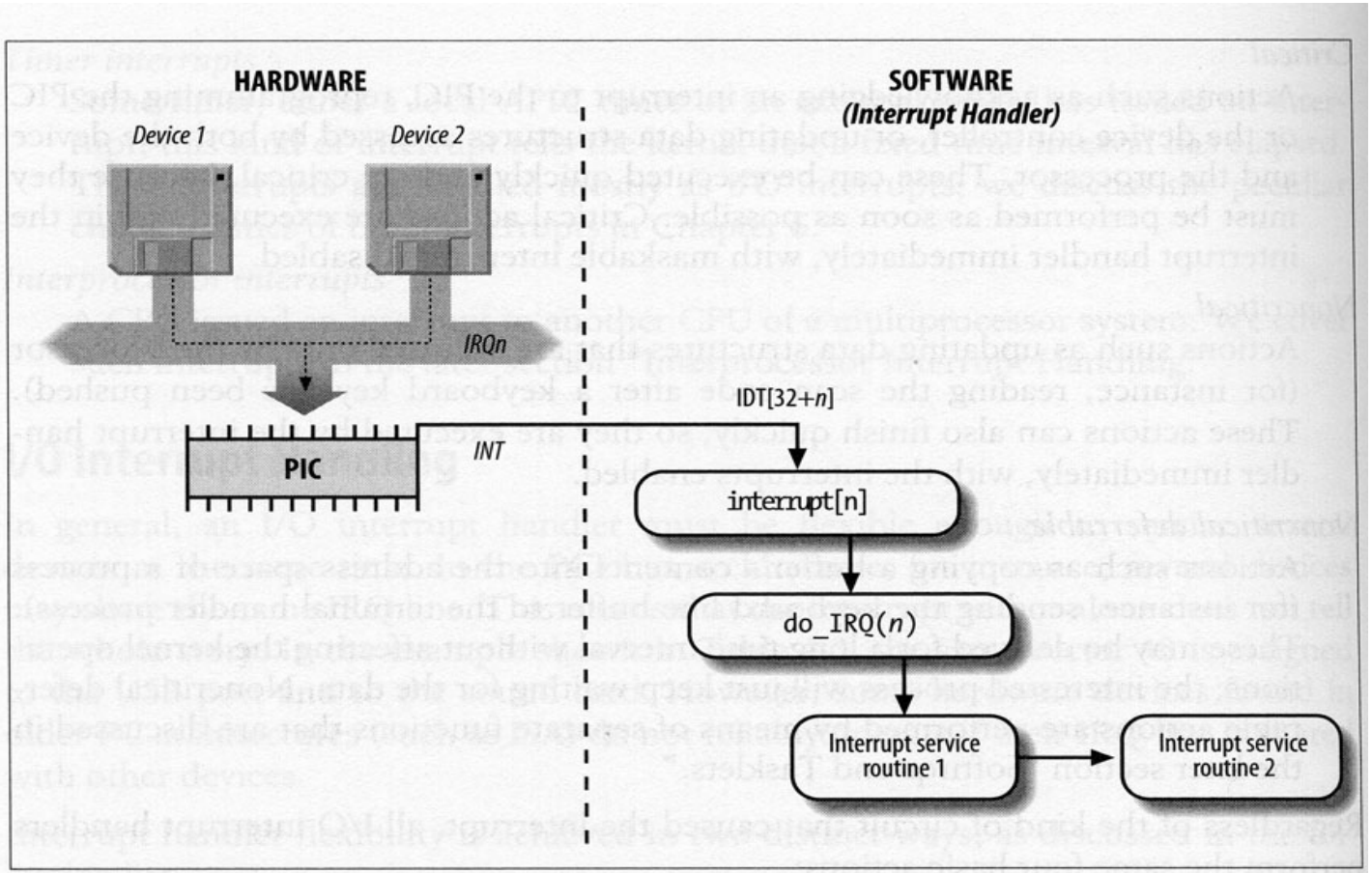

Figure 4-4. I/O interrupt handling

IRQ descriptor

- IRQ sharing

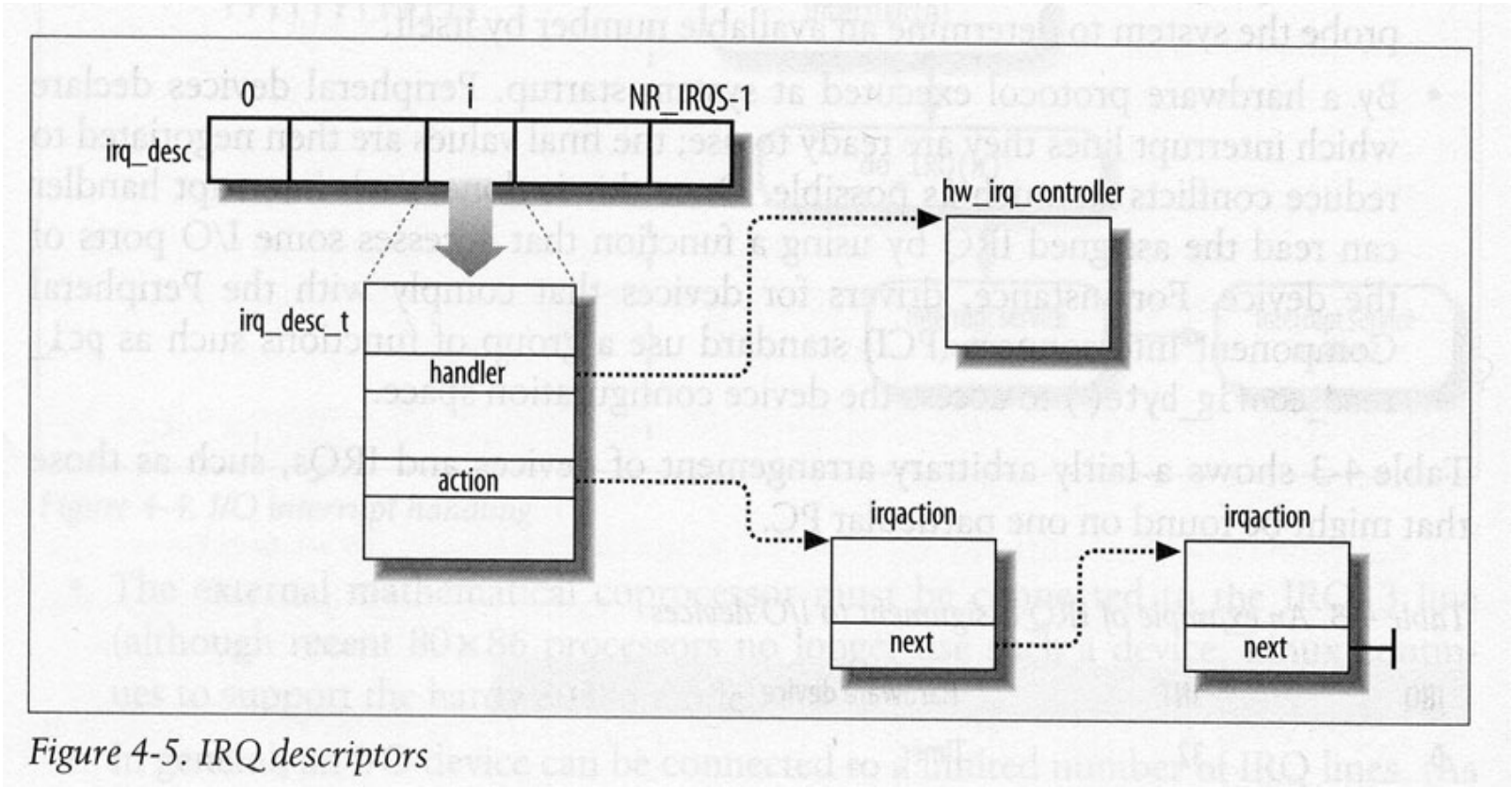

Figure 4-5. IRQ descriptors

Deferrable functions

- Non-critical work (functions) can be executed later
  - More efficient kernel
- Softirq
  - Defined at compile-time
  - Run concurrently by multiple CPUs
- Tasklet
  - Built on two softirqs
  - The same type of tasklets is executed sequentially, not executed by two CPUs at the same times
  - Good for device drivers

Deferrable functions (cont.)

- Softirqs and tasklets are executed at interrupt context
- Work queues
  - Functions in work queues run in process context
  - Run by kernel threads called worker threads
  - Can be blocked and can do work like a process except it is in kernel mode only

Module and kernel

- A module normally consists of some functions for system calls and some functions for interrupt handling
- Why make modules in kernel space?
    - Interrupts, direct memory access, direct access to I/O ports, response time, limitations

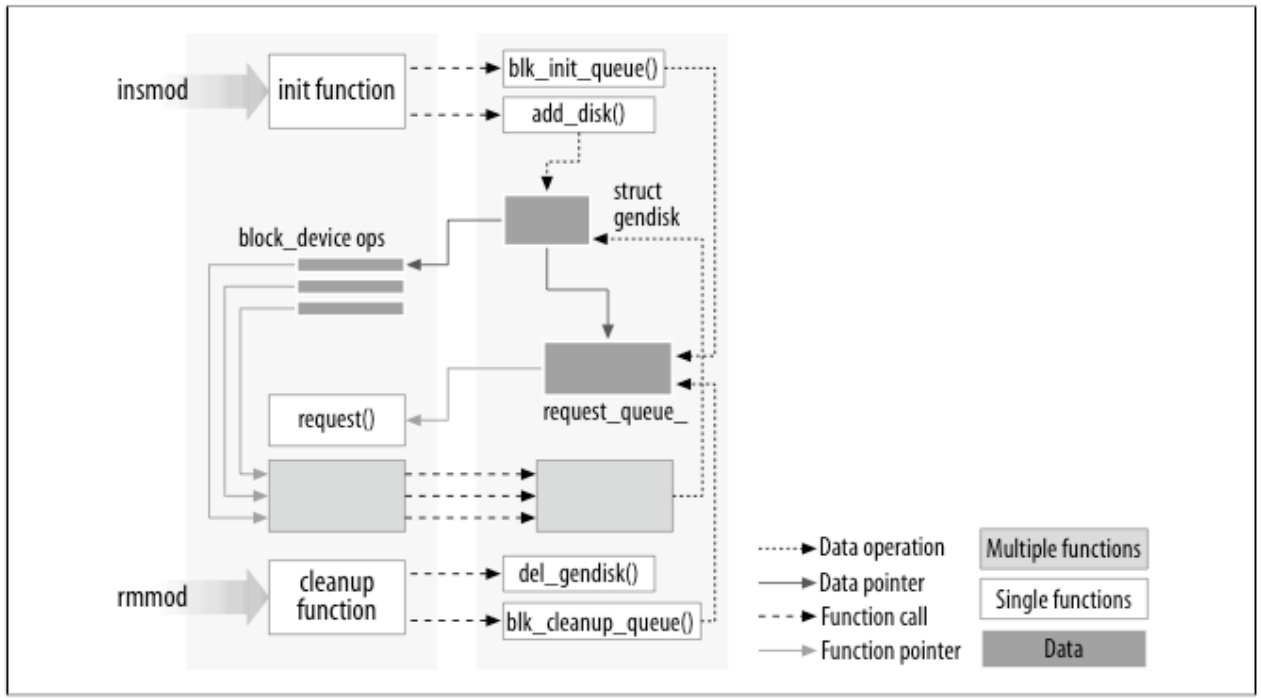

Figure 2-1. Linking a module to the kernel

A few issues

- Concurrency
- Current process
  - *current* (note: it is for SMP as well), pointer to the process descriptor of the current running process
- Small kernel stack
- Be careful of "__" prefixed functions
- No floating point in kernel

Basic kernel programming

- Export kernel symbols
  - *export_symbol(name)*
- Header files (the *include* under kernel src)
- Internal state
  - */proc* and */sys* FS entries
- Error handling
  - Undo side-effects
- Module parameters
  - *module_param(name, type, permission)*

Debugging

- Careful and robust design
- printk()
  - Check /var/log/syslog
- Run in a simulator like QEMU
  - Not suitable for hardware related code
- We shall talk about other more advanced debugging later

RCU

- Lock-free data structures are efficient
  - E.g. circular buffer
  - But sometimes too tricky, e.g. linked list
- Lock-free algorithms achieve two purposes
  - Atomically change data for readers
  - Detect and prevent conflicts between writers
- Key idea of RCU
  - Leave older data for readers, no update on it
  - Instead, update on a new copy (memory reuse is an issue)
  - Read today's article for details

Analysis of RCU

- Pros
  - Reader code is efficient and simpler by avoiding locking issues
    - Readers may see old but consistent data when there are concurrent writers, which is ok with many applications e.g. net routing table
  - Writer code is simpler than lock-free data structures
  - Good performance for read-heavy workloads
    - A factor of 2-4 due to no bus write for readers
- Cons
  - Memory is not freed immediately
  - Grace period detection at the background
  - Memory copying for updaters

RCU in Linux

- Network routing table
- Module unloading
- FD flags
- CPU array