ADRIAN PERRIG & TORSTEN HOEFLER

Networks and Operating Systems (252-0062-00)
Chapter 9: I/O Subsystems

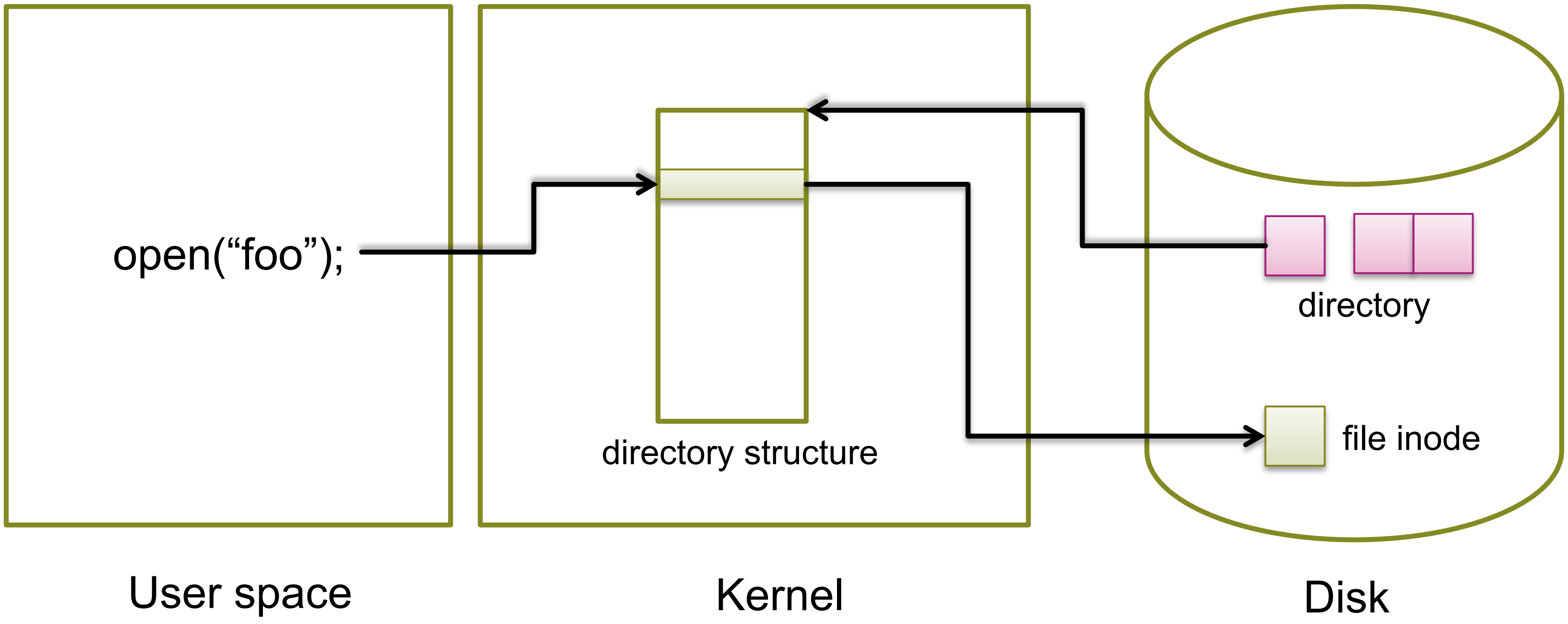
Our Small Quiz

- **True or false (raise hand)**
  - Directories can never contain cycles
  - Access control lists scale to large numbers of principals
  - Capabilities are stored with the principals revocation can be complex
  - POSIX (Unix) access control is scalable to large numbers of files
  - Named pipes are special files in Unix
  - Memory mapping improves sequential file access
  - Accessing different files on disk has different speeds
  - The FAT filesystem enables fast random access
  - FFS enables fast random access for small files
  - The minimum storage for a file in FFS is 8kB (4kB inode + block)
  - Block groups in FFS are used to simplify the implementation
  - Multiple hard links in FFS are stored in the same inode
  - NTFS stores files that are contiguous on disk more efficiently than FFS
  - The volume information in NTFS is a file in NTFS

In-memory data structures

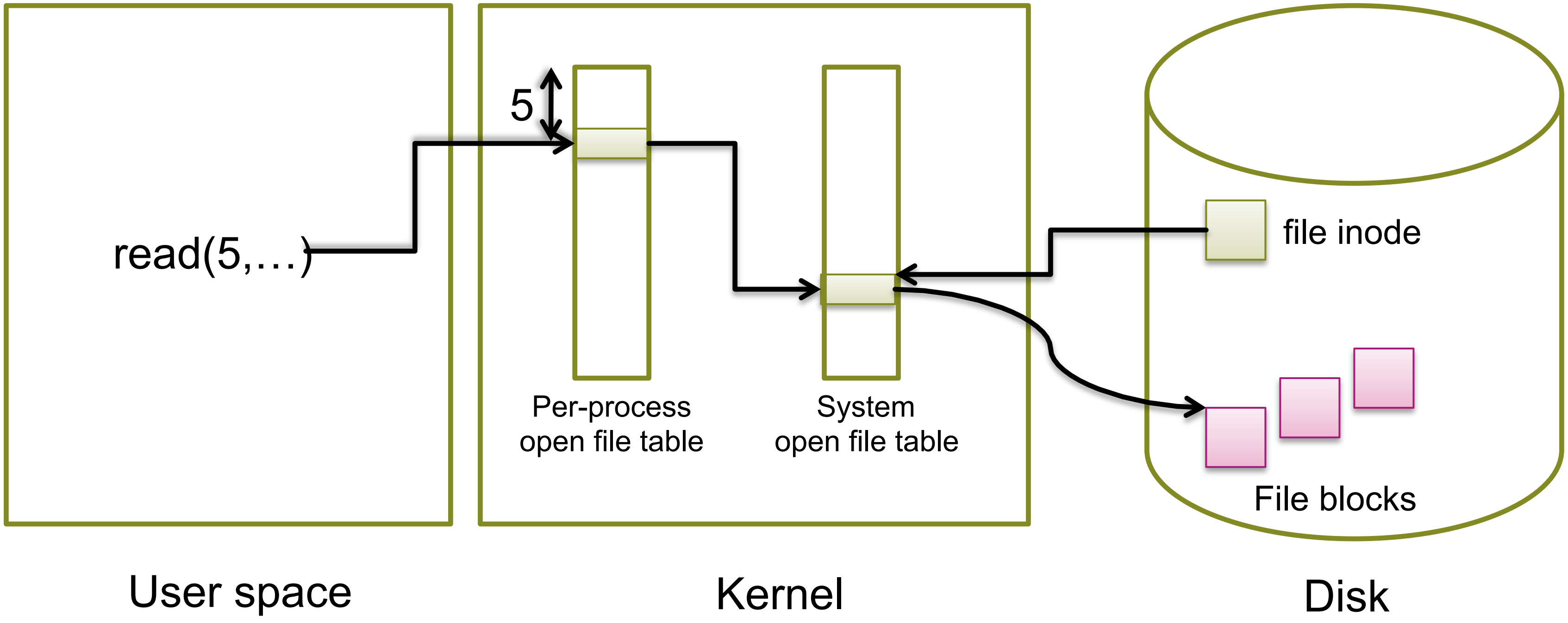
Opening a file

- **Directories translated into kernel data structures on demand:**



Reading and writing

- **Per-process open file table → index into…**
- **System open file table → cache of inodes**



Efficiency and Performance

- **Efficiency dependent on:**
  - disk allocation and directory algorithms
  - types of data kept in file's directory entry

- **Performance**
  - disk cache – separate section of main memory for frequently used blocks
  - free-behind and read-ahead – techniques to optimize sequential access
  - improve PC performance by dedicating section of memory as virtual disk, or RAM disk

Page Cache

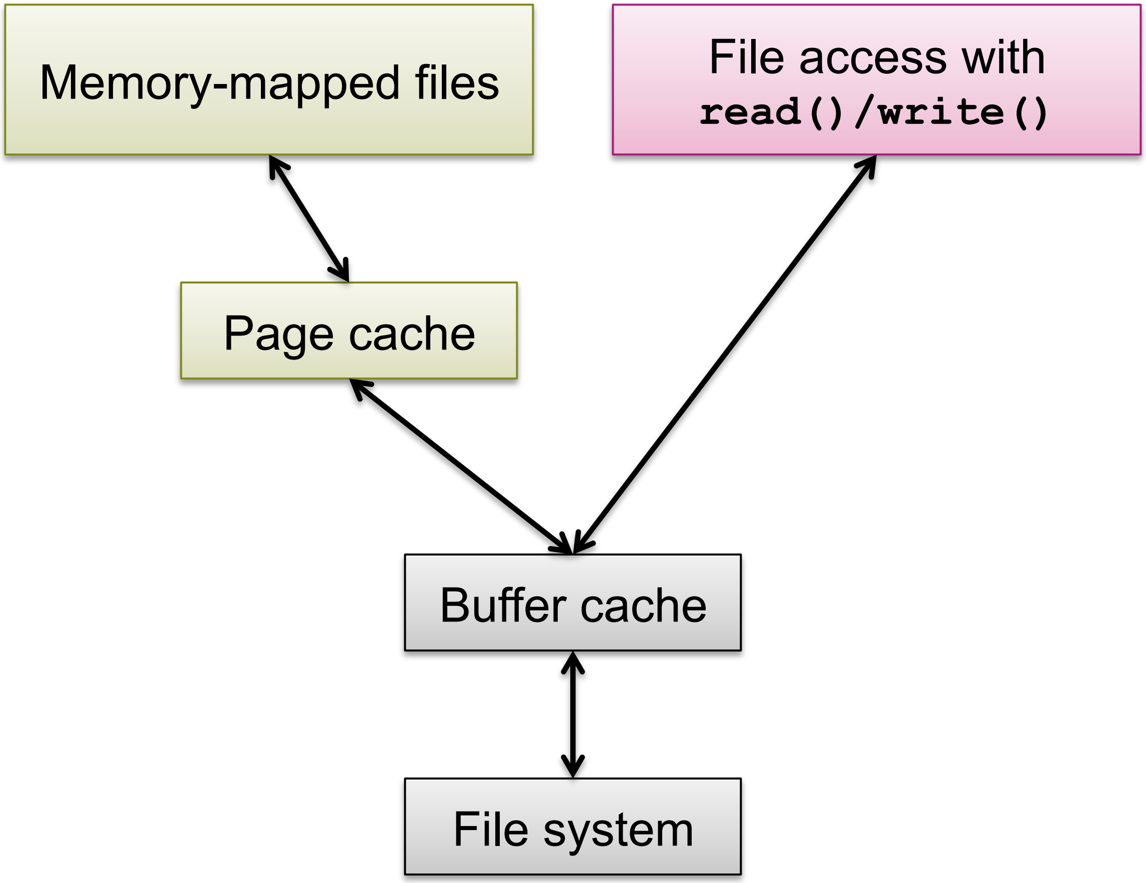
- A page cache caches pages rather than disk blocks using virtual memory techniques

- Memory-mapped I/O uses a page cache

- Routine I/O through the file system uses the buffer (disk) cache

- This leads to the following figure

2 layers of caching?


Memory-mapped files

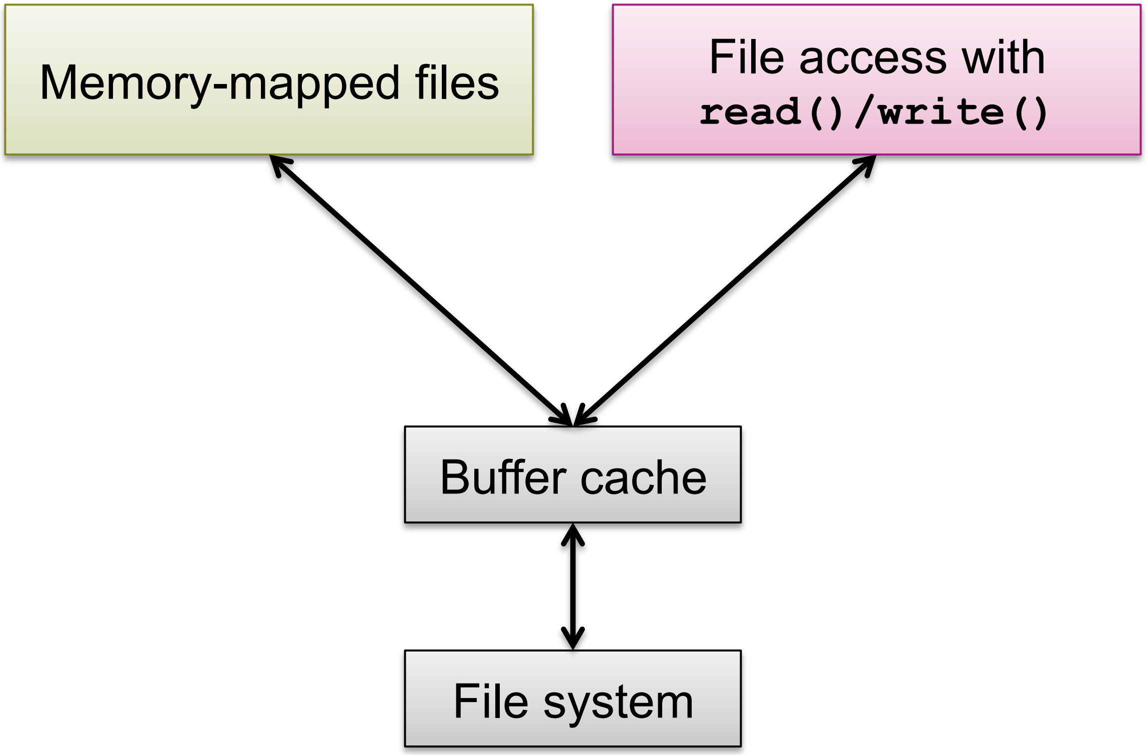
File access with
`read()/write()`

Page cache

Buffer cache

File system

Unified Buffer Cache



Memory-mapped files

File access with `read()/write()`

Buffer cache

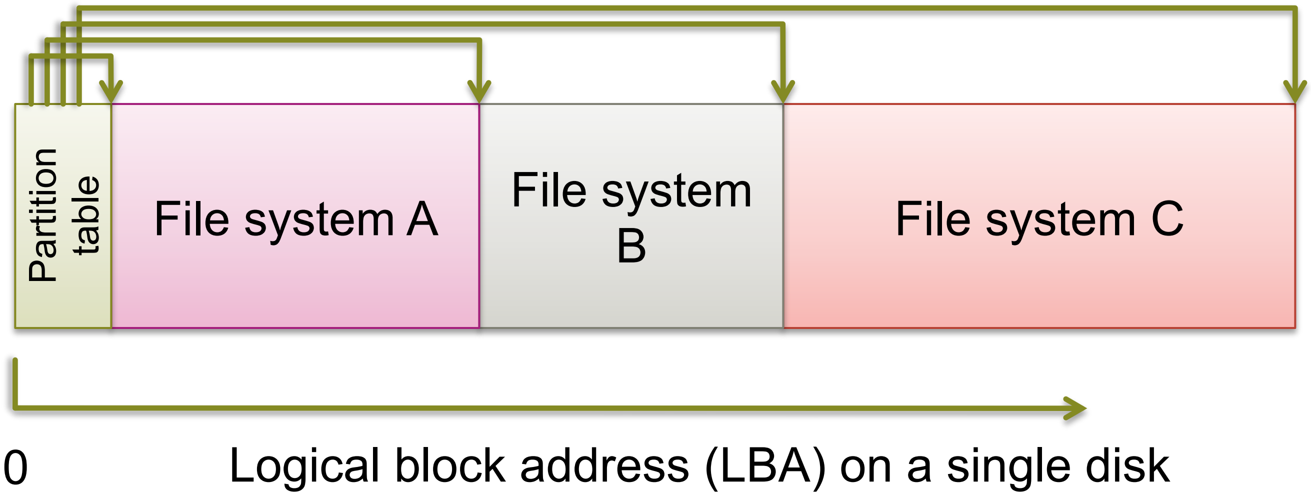
File system

Filesystem Recovery

- Consistency checking – compares data in directory structure with data blocks on disk, and tries to fix inconsistencies
- Use system programs to back up data from disk to another storage device (floppy disk, magnetic tape, other magnetic disk, optical)
- Recover lost file or disk by restoring data from backup

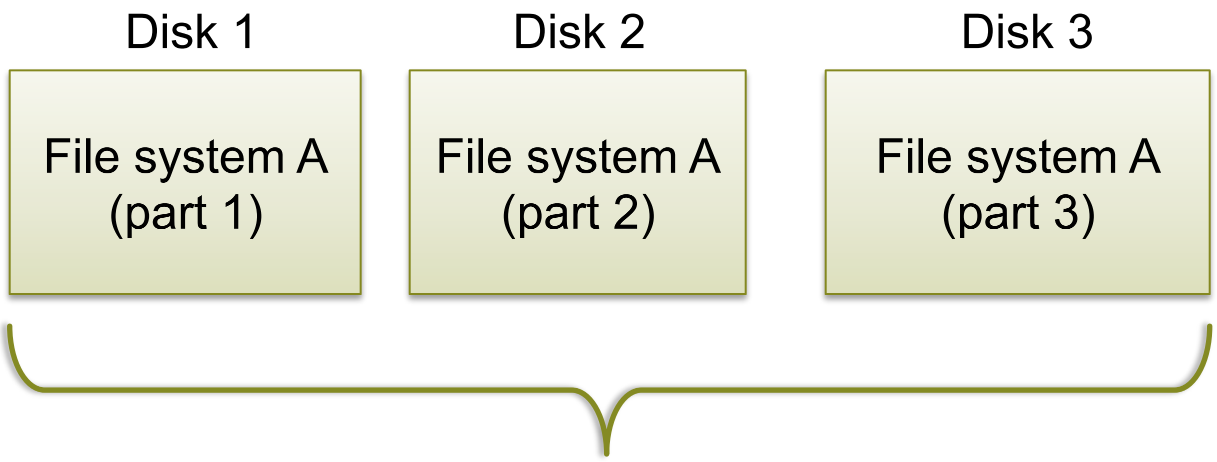
Disks, Partitions and Logical Volumes

Partitions



- **Multiplex single disk among >1 file systems**
- **Contiguous block ranges per FS**

Logical volumes



Disk 1

Disk 2

Disk 3

File system A (part 1)

File system A (part 2)

File system A (part 3)

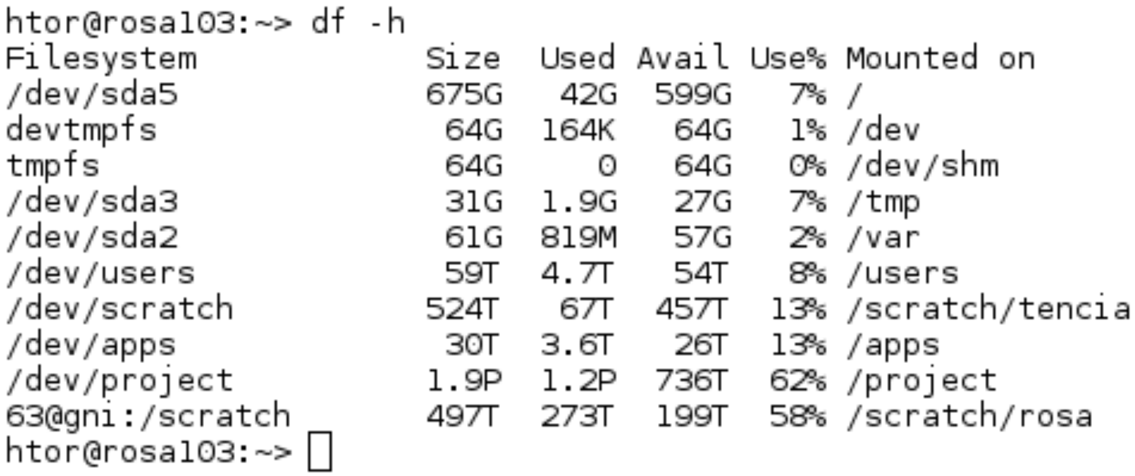
Single *logical volume* with file system A

- ***Emulate*** **1 virtual disk from >1 physical ones**
- **Single file system spanning >1 disk**

Multiple file systems

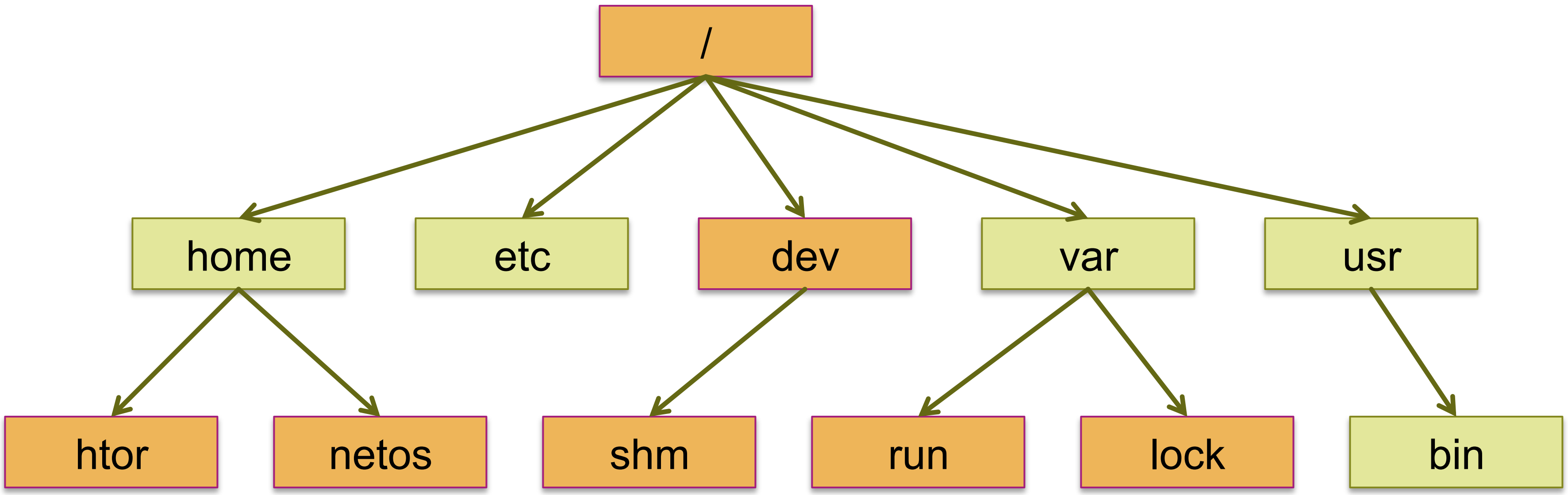
- **How to name files in multiple file systems?**
- **Top-level volume names:**
  - Windows C:, D:, etc.
  - \\fs-systems.ethz.ch\
- **Bind "mount points" in name space**
  - Unix, etc.

Mount points



```
htor@rosa103:~> df -h
Filesystem             Size  Used Avail Use% Mounted on
/dev/sda5              675G   42G  599G   7% /
devtmpfs                64G  164K   64G   1% /dev
tmpfs                   64G     0   64G   0% /dev/shm
/dev/sda3               31G  1.9G   27G   7% /tmp
/dev/sda2               61G  819M   57G   2% /var
/dev/users              59T  4.7T   54T   8% /users
/dev/scratch           524T   67T  457T  13% /scratch/tencia
/dev/apps               30T  3.6T   26T  13% /apps
/dev/project           1.9P  1.2P  736T  62% /project
63@gni:/scratch        497T  273T  199T  58% /scratch/rosa
htor@rosa103:~> 
```

File hierarchy with mounts



Mount point 🟧

Normal directory 🟨

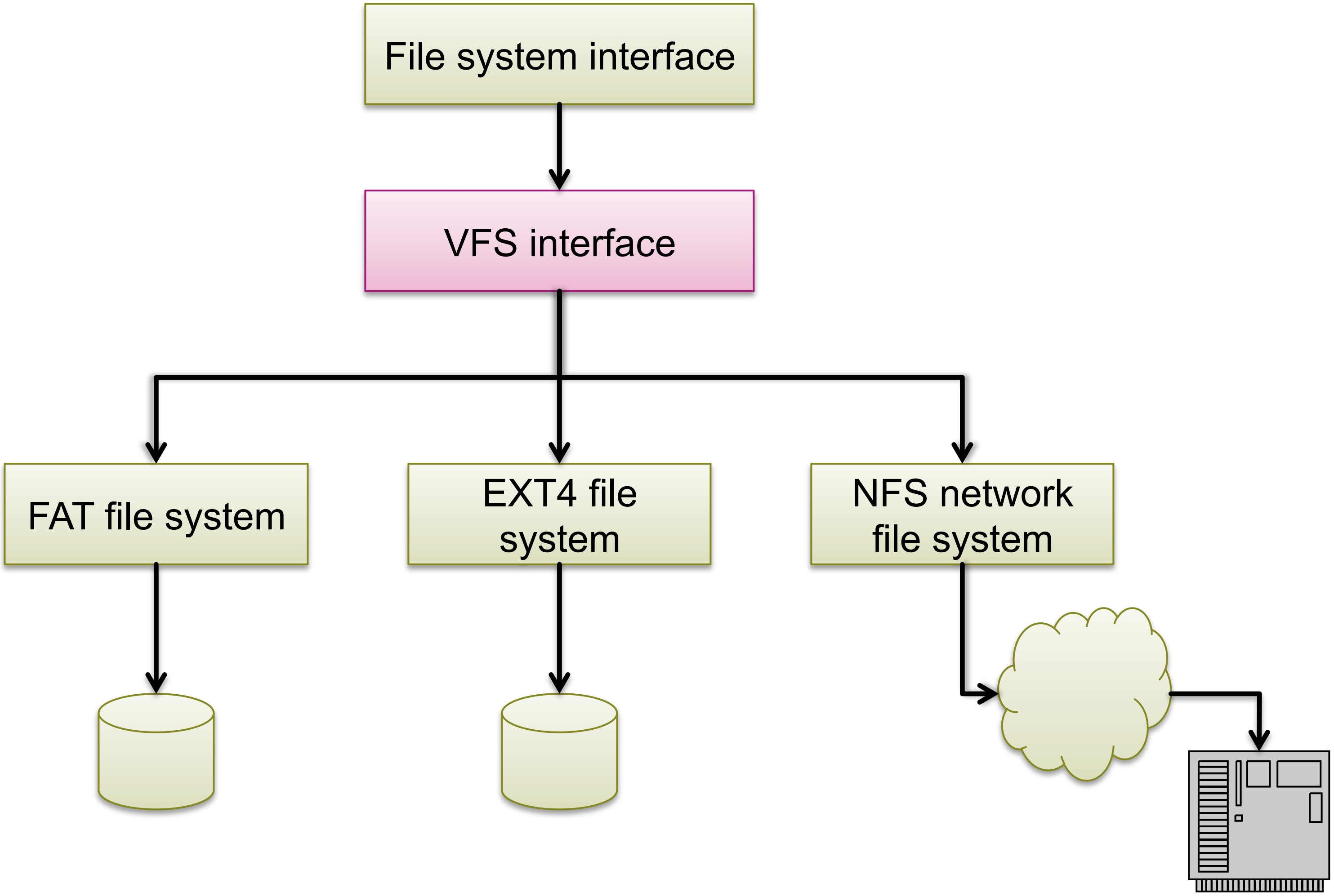
Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.

- VFS allows the same system call interface (the API) to be used for different types of file systems.

- The API is to the VFS interface, rather than any specific type of file system.

Virtual File System



File system interface

VFS interface

FAT file system

EXT4 file system

NFS network file system

Rest of today: I/O

1. Recap: what devices look like
2. Device drivers
3. The I/O subsystem

Recap from CASP:
What does a device look like?

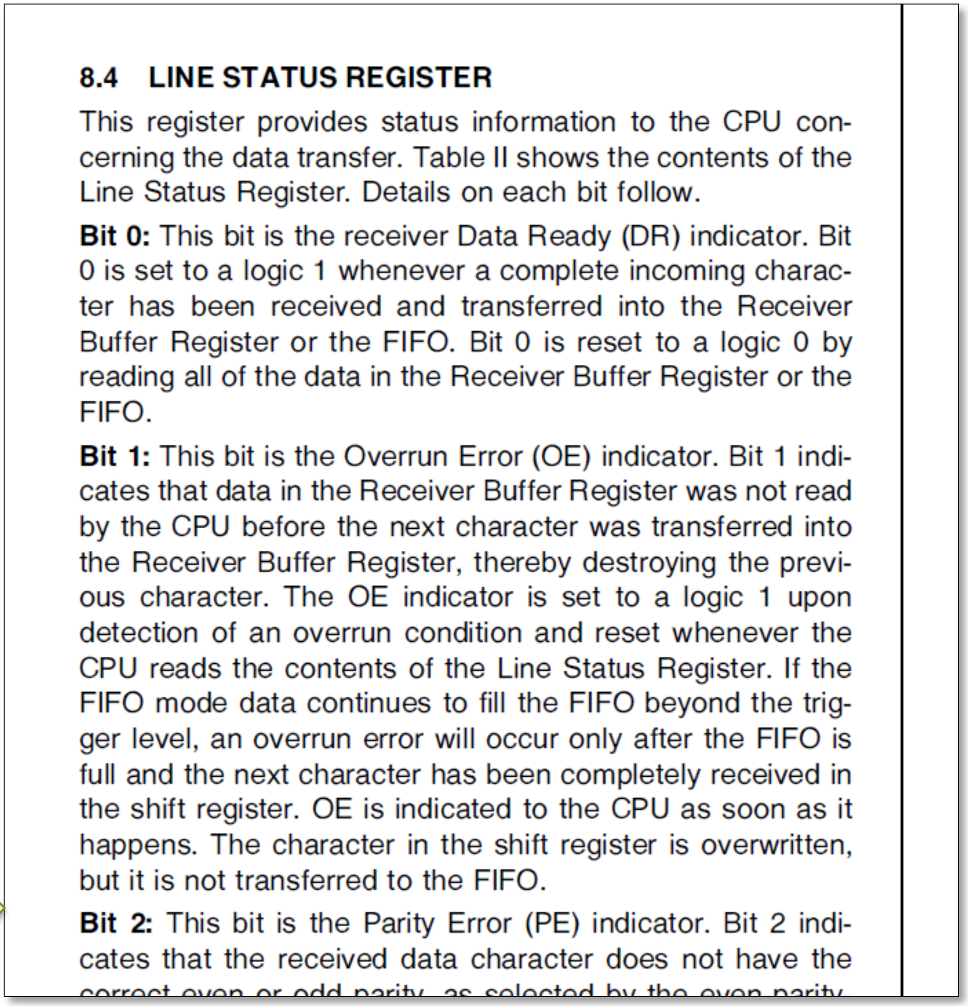
Recap: What is a device?

Specifically, to an OS programmer:

- **Piece of hardware visible from software**
- **Occupies some location on a bus**
- **Set of registers**
  - Memory mapped or I/O space
- **Source of interrupts**
- **May initiate Direct Memory Access transfers**

Recap: Registers

- **Details of registers given in chip "datasheets" or "data books"**
- **Information is rarely trusted by OS programmers ☺**



8.4 LINE STATUS REGISTER

This register provides status information to the CPU concerning the data transfer. Table II shows the contents of the Line Status Register. Details on each bit follow.

Bit 0: This bit is the receiver Data Ready (DR) indicator. Bit 0 is set to a logic 1 whenever a complete incoming character has been received and transferred into the Receiver Buffer Register or the FIFO. Bit 0 is reset to a logic 0 by reading all of the data in the Receiver Buffer Register or the FIFO.

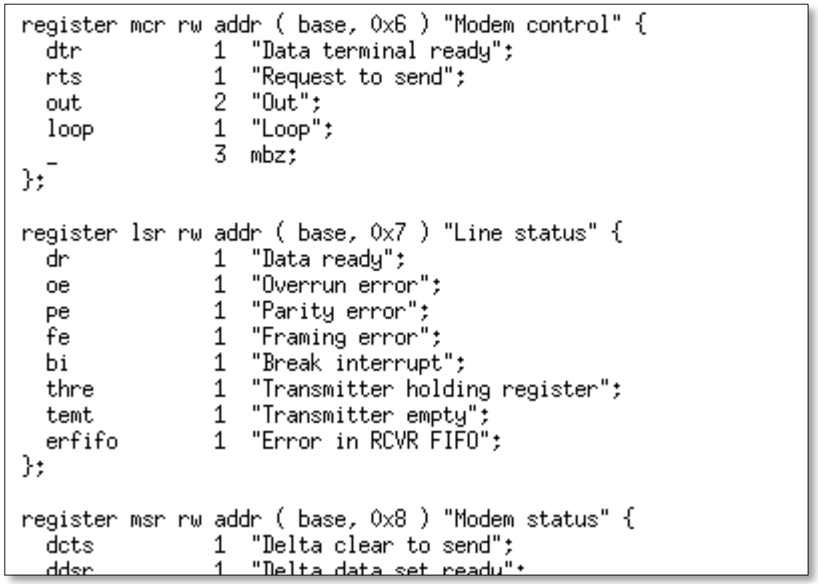
Bit 1: This bit is the Overrun Error (OE) indicator. Bit 1 indicates that data in the Receiver Buffer Register was not read by the CPU before the next character was transferred into the Receiver Buffer Register, thereby destroying the previous character. The OE indicator is set to a logic 1 upon detection of an overrun condition and reset whenever the CPU reads the contents of the Line Status Register. If the FIFO mode data continues to fill the FIFO beyond the trigger level, an overrun error will occur only after the FIFO is full and the next character has been completely received in the shift register. OE is indicated to the CPU as soon as it happens. The character in the shift register is overwritten, but it is not transferred to the FIFO.

Bit 2: This bit is the Parity Error (PE) indicator. Bit 2 indicates that the received data character does not have the

From the data sheet for the PC16550 UART (standard PC serial port)

Registers

- **Slightly more readable version:**
    - From Barrelfish, in a language called “Mackerel”
    - Compiler generates code to do the “bit-banging”



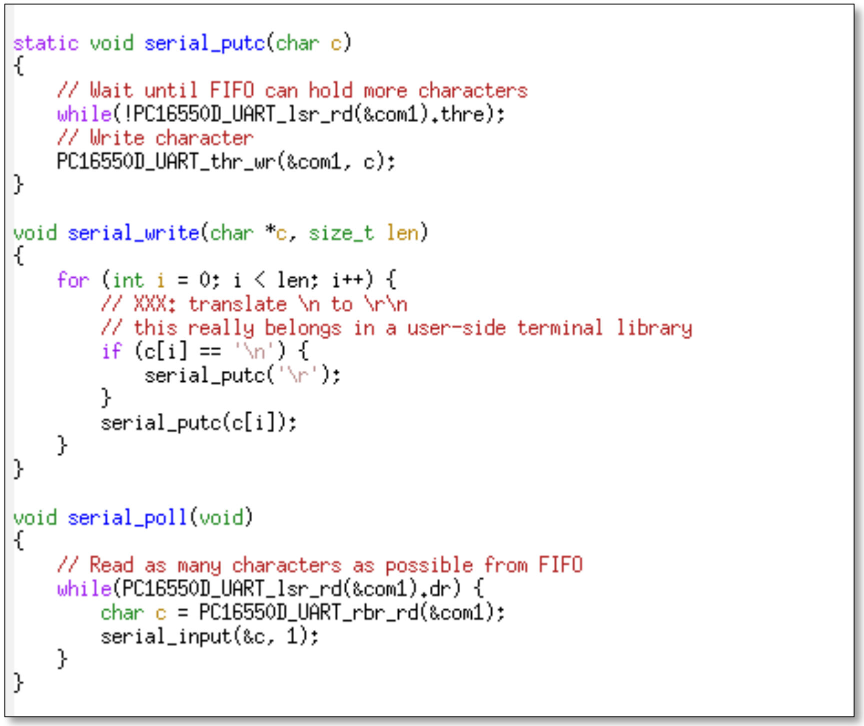
```
register mcr rw addr ( base, 0x6 ) "Modem control" {
  dtr           1  "Data terminal ready";
  rts           1  "Request to send";
  out           2  "Out";
  loop          1  "Loop";
  _             3  mbz;
};

register lsr rw addr ( base, 0x7 ) "Line status" {
  dr            1  "Data ready";
  oe            1  "Overrun error";
  pe            1  "Parity error";
  fe            1  "Framing error";
  bi            1  "Break interrupt";
  thre          1  "Transmitter holding register";
  temt          1  "Transmitter empty";
  erfifo        1  "Error in RCVR FIFO";
};

register msr rw addr ( base, 0x8 ) "Modem status" {
  dcts          1  "Delta clear to send";
  ddsr          1  "Delta data set ready";
```

Using registers

- **From the Barrelfish console driver**
  - Very simple!
- **Note the issues:**
  - Polling loop on send
  - Polling loop on receive

    Only a good idea for debug
  - CPU must write all the data

    not much in this case



```
static void serial_putc(char c)
{
    // Wait until FIFO can hold more characters
    while(!PC16550D_UART_lsr_rd(&com1).thre);
    // Write character
    PC16550D_UART_thr_wr(&com1, c);
}

void serial_write(char *c, size_t len)
{
    for (int i = 0; i < len; i++) {
        // XXX: translate \n to \r\n
        // this really belongs in a user-side terminal library
        if (c[i] == '\n') {
            serial_putc('\r');
        }
        serial_putc(c[i]);
    }
}

void serial_poll(void)
{
    // Read as many characters as possible from FIFO
    while(PC16550D_UART_lsr_rd(&com1).dr) {
        char c = PC16550D_UART_rbr_rd(&com1);
        serial_input(&c, 1);
    }
}
```

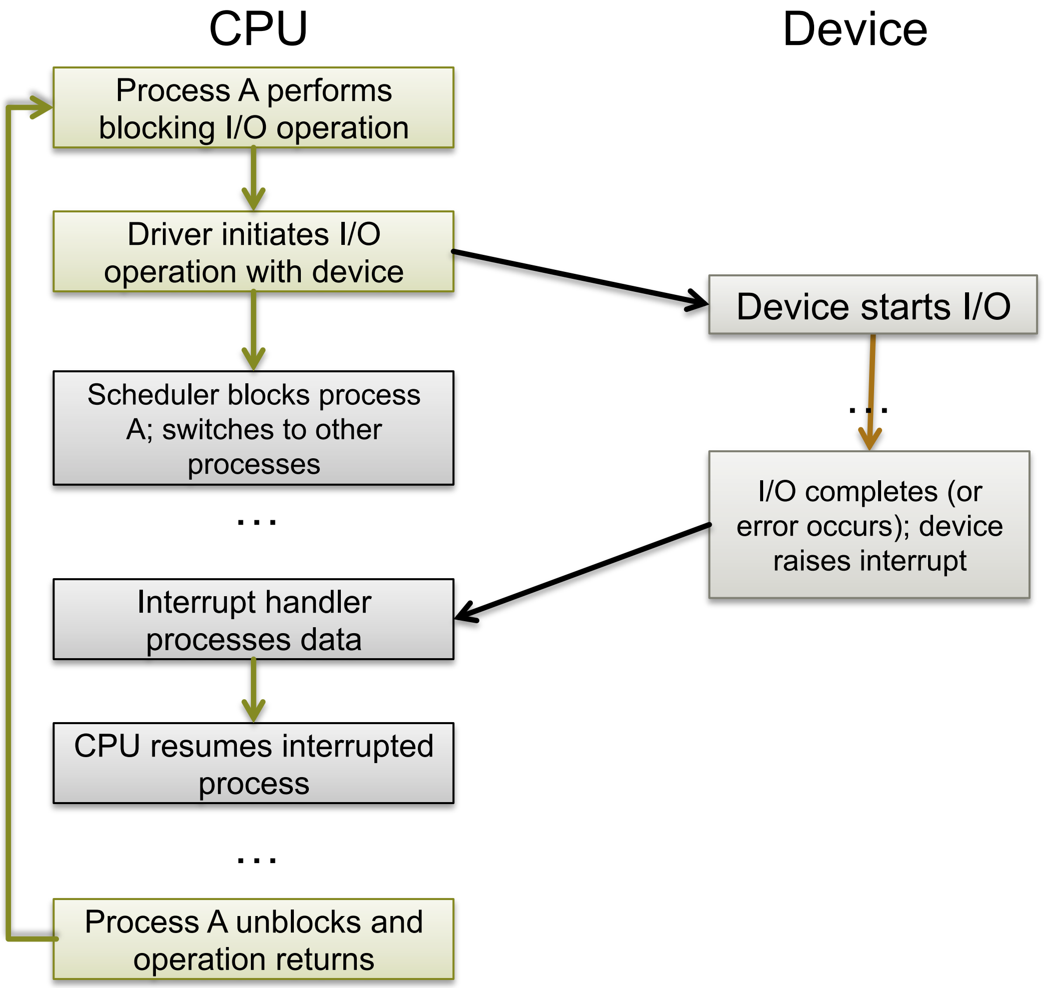
Very simple UART driver

- **Actually, far too simple!**
  - But this is how the first version always looks…
- **No initialization code, no error handling.**
- **Uses *Programmed I/O* (PIO)**
  - CPU explicitly reads and writes all values to and from registers
  - All data must pass through CPU registers
- **Uses *polling***
  - CPU polls device register waiting before send/receive
    Tight loop!
  - Can’t do anything else in the meantime
    Although could be extended with threads and care…
  - Without CPU polling, no I/O can occur

Recap: Interrupts

- **CPU Interrupt-request line triggered by I/O device**
- **Interrupt handler receives interrupts**
- **Maskable to ignore or delay some interrupts**
- **Interrupt vector to dispatch interrupt to correct handler**
  - Based on priority
  - Some **nonmaskable**
- **Interrupt mechanism also used for exceptions**

Interrupt-Driven I/O Cycle


CPU

Device

Process A performs blocking I/O operation

Driver initiates I/O operation with device

Device starts I/O

…

Scheduler blocks process A; switches to other processes

…

I/O completes (or error occurs); device raises interrupt

Interrupt handler processes data

CPU resumes interrupted process

…

Process A unblocks and operation returns

Recap: Direct Memory Access

- **Avoid *programmed I/O* for lots of data**
  - E.g. fast network or disk interfaces
- **Requires *DMA controller***
  - Generally built-in these days
- **Bypasses CPU to transfer data directly between I/O device and memory**
  - Doesn't take up CPU time
  - Can save memory bandwidth
  - Only one interrupt per transfer

I/O Protection

I/O operations can be dangerous to normal system operation!

- **Dedicated I/O instructions usually privileged**
- **I/O performed via system calls**
  - Register locations must be protected
- **DMA transfers must be carefully checked**
  - Bypass memory protection!
  - IOMMUs are beginning to appear…

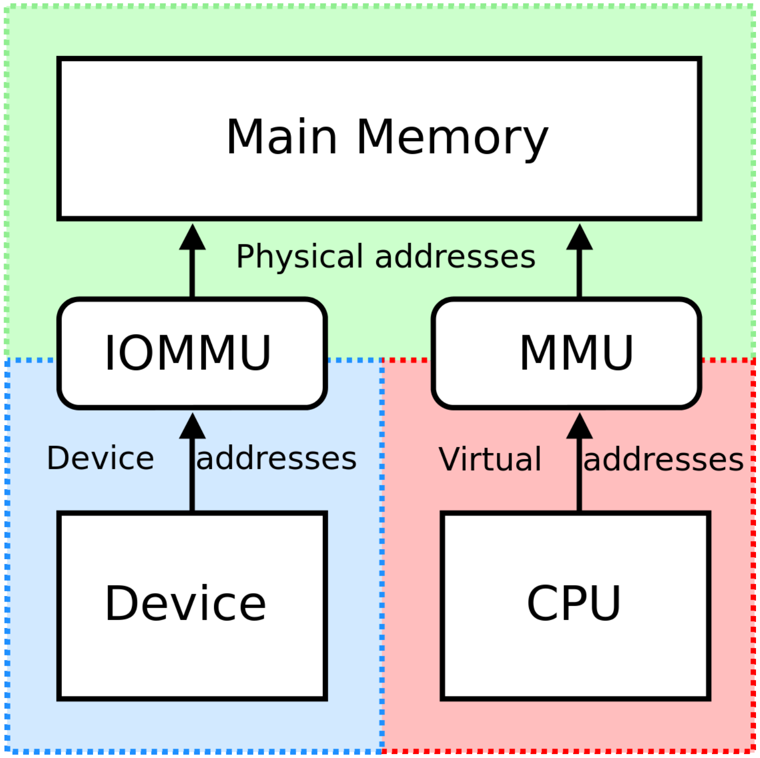
IOMMUs

IOMMU does the same for the I/O devices as MMU does for the CPU!

- Translates device adresses (so called DVAs) into physical ones,
- Uses so called IOTLB (I/O TLB)
- Works for DMA-capable devices :-)

- Examples:
  - Intel VT-d
  - AMD IOMMU

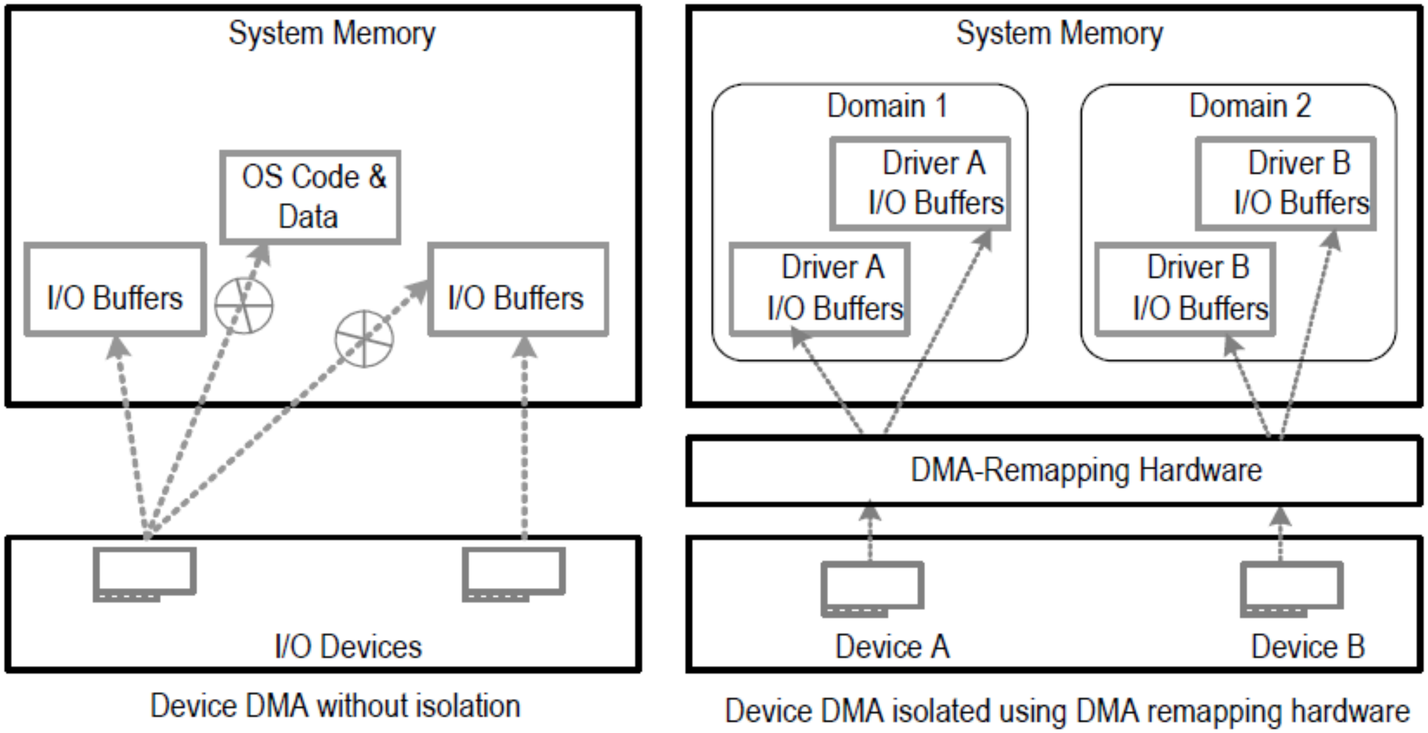
  - ...very similar in functionality



Source: Wikipedia

IOMMUs

- **Security features for VMs**
  - Possibility to assign different devices to different address **domains**
  - By address remapping we can isolate the domains from one another, thus 'sandboxing' untrusted devices



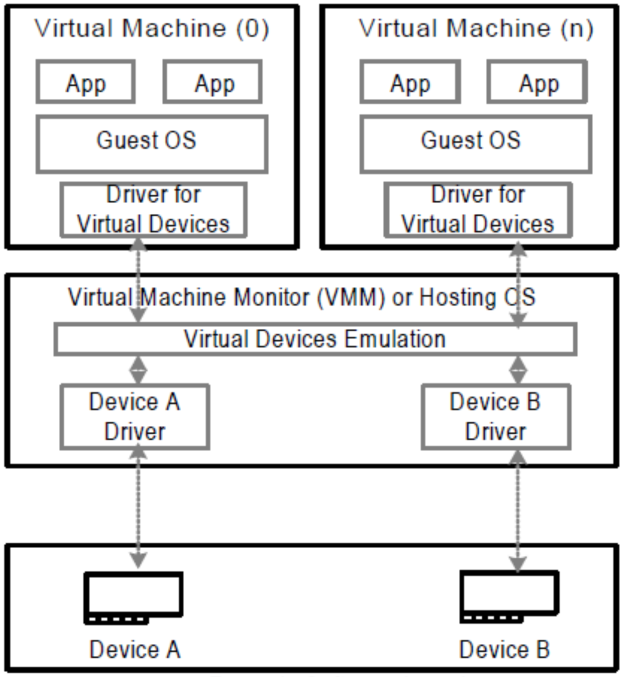
Device DMA without isolation

Device DMA isolated using DMA remapping hardware

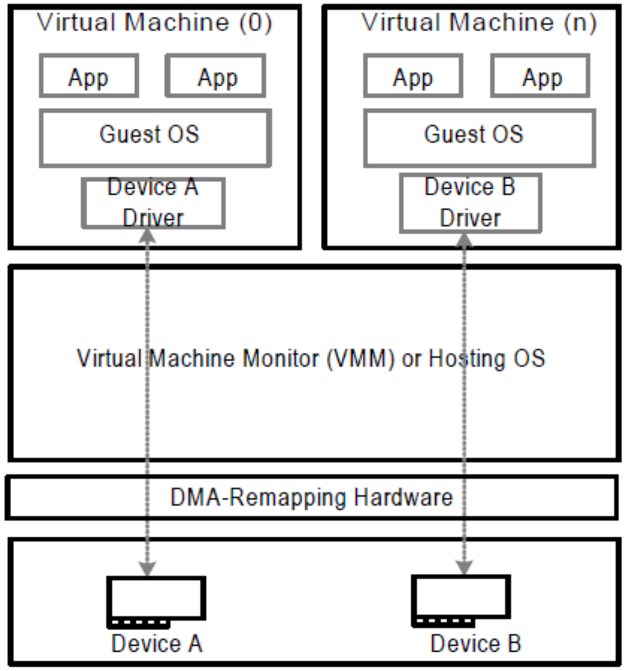
Source: Intel VT-d specification

IOMMUs

- **IOMMUs were basically designed for enhancing virtualization**
  - All the remapping & security features can be applied to guest virtual machines
  - Better performance than software-based I/O virtualization



Example Software-based I/O Virtualization



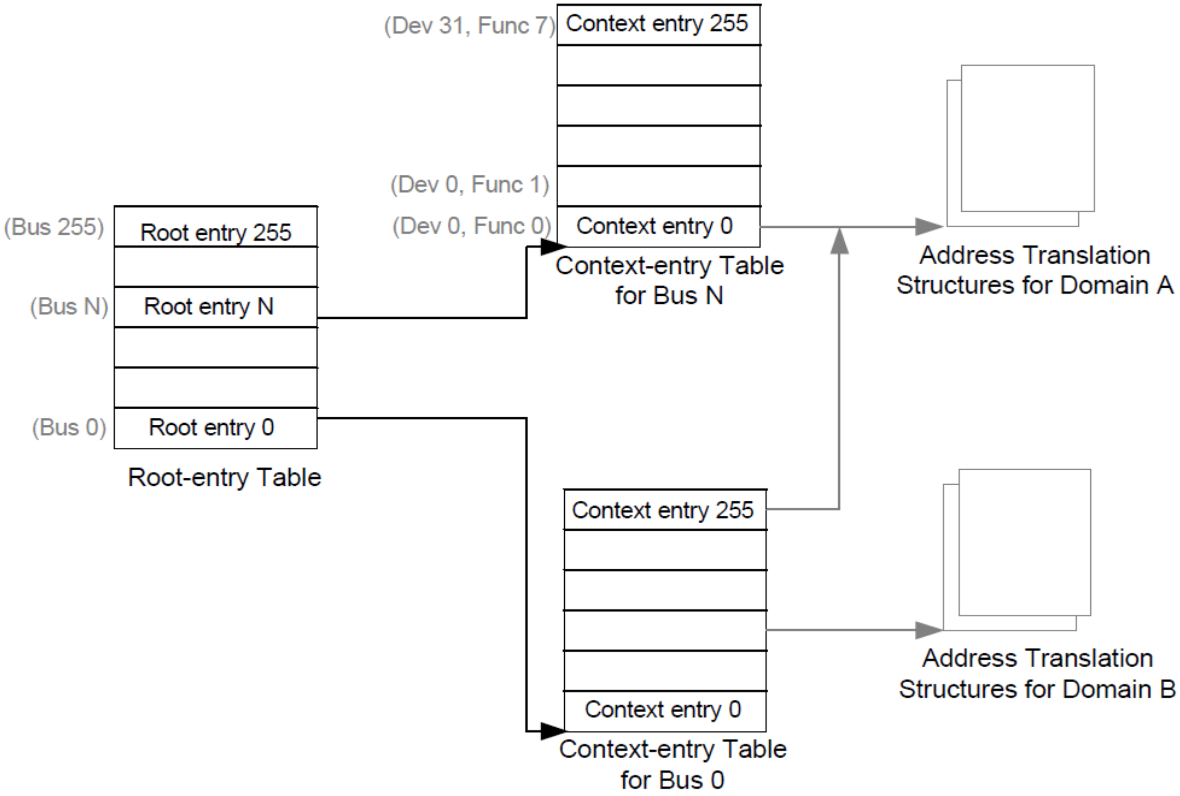
Direct Assignment of I/O Devices

Source: Intel VT-d specification

IOMMUs

- **IOMMUs take as the 'input request' the ID consisting of:**
  - *Bus ID, stored in* ***root tables*** *(support for multiple buses),*
  - *Device ID, stored in* ***context tables*** *(support for multiple devices within each bus)*
  - *Function ID, also stored in* ***context tables*** *(support for multiple func. within each device)*

- **Different page table per I/O device**



Source: Intel VT-d specification

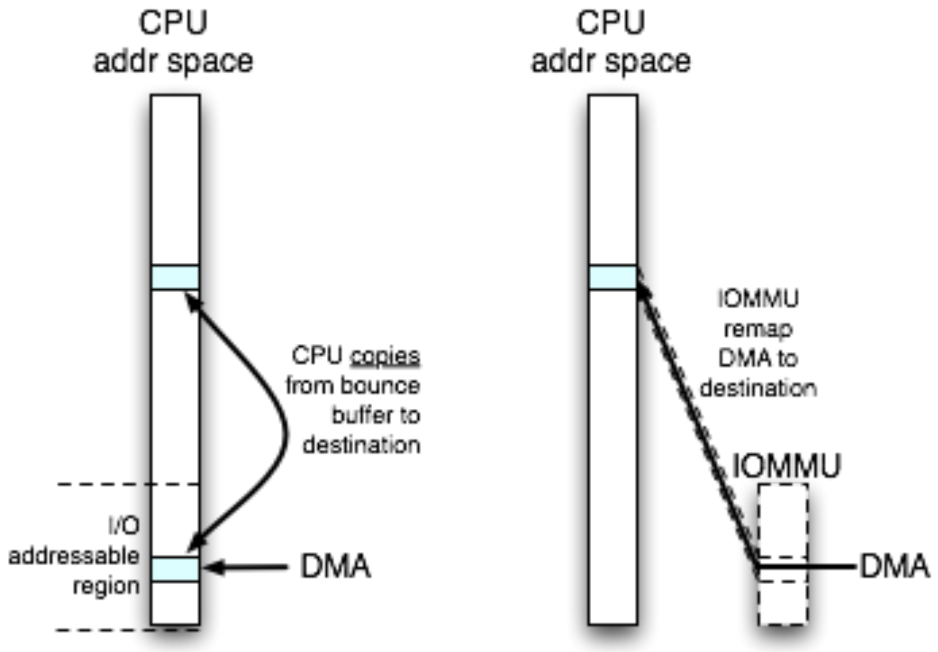
IOMMUs - Address remapping

- **IOMMUs support page remapping**
  - Some PCI devices use 32 bit addressing

- **IOMMU Page Tables**
  - Similar to 'standard' multi-level page tables
  - Write-only / read-only bits
  - Support for huge pages
  - Currently no support for more extended features (e.g., reference bits)

bounce buffers

IOMMU



Source: http://codingrelic.geekhold.com/

IOMMUs

- **IOMMUs are much broader topic**
- **They provide also:**
  - Interrupt remapping (you can control interrupts in a similar way as memory accesses)
  - Device I/O TLBs (Intel VT-d)
  - Fault logging
  - …

- **You can think of many interesting use cases for them :-)**
  - **Interested? New ideas?**

Device drivers

Device drivers

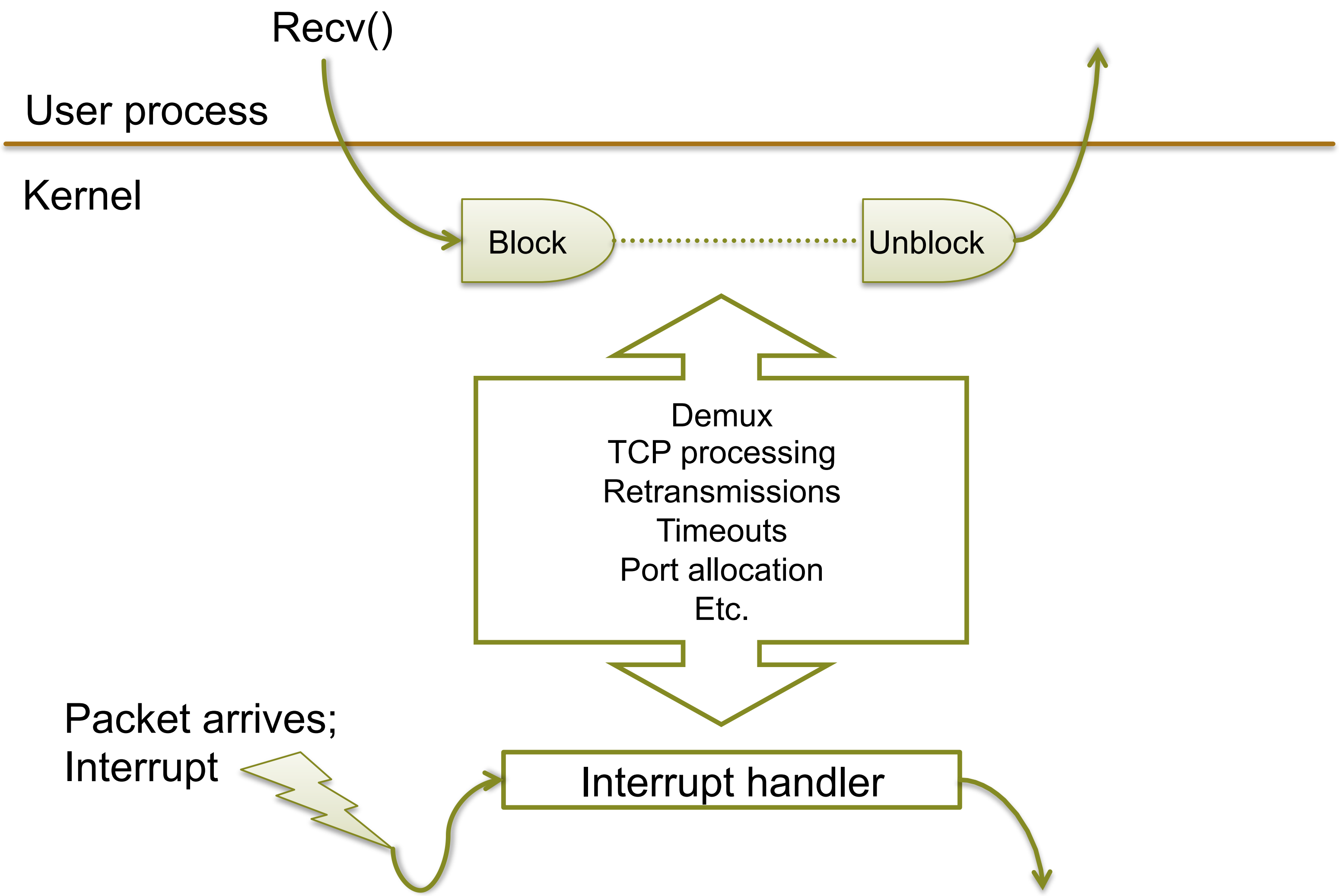
- **Software object (module, object, process, hunk of code) which abstracts a device**
    - Sits between hardware and rest of OS
    - Understands device registers, DMA, interrupts
    - Presents uniform interface to rest of OS

- **Device abstractions (“driver models”) vary…**
    - Unix starts with “block” and “character” devices

Device driver structure: the basic problem

- **Hardware is *interrupt driven*.**
  - System must respond to unpredictable I/O events (or events it is expecting, but doesn't know when)
- **Applications are (often) *blocking***
  - Process is waiting for a specific I/O event to occur
- **Often considerable processing *in between***
  - TCP/IP processing, retries, etc.
  - File system processing, blocks, locking, etc.

Example: network receive



Recv()

User process

Kernel

Block

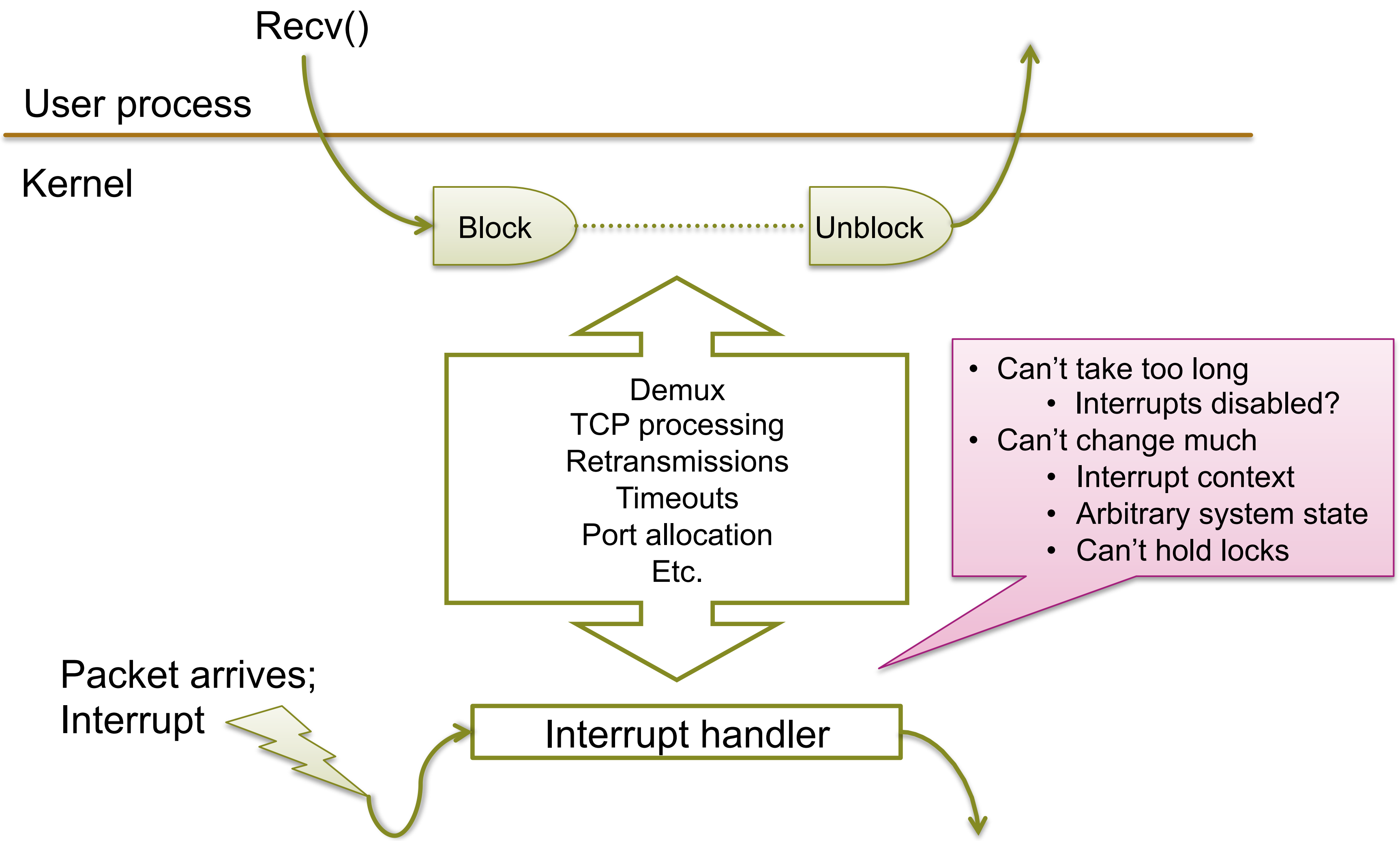
Unblock

Demux
TCP processing
Retransmissions
Timeouts
Port allocation
Etc.

Packet arrives;
Interrupt

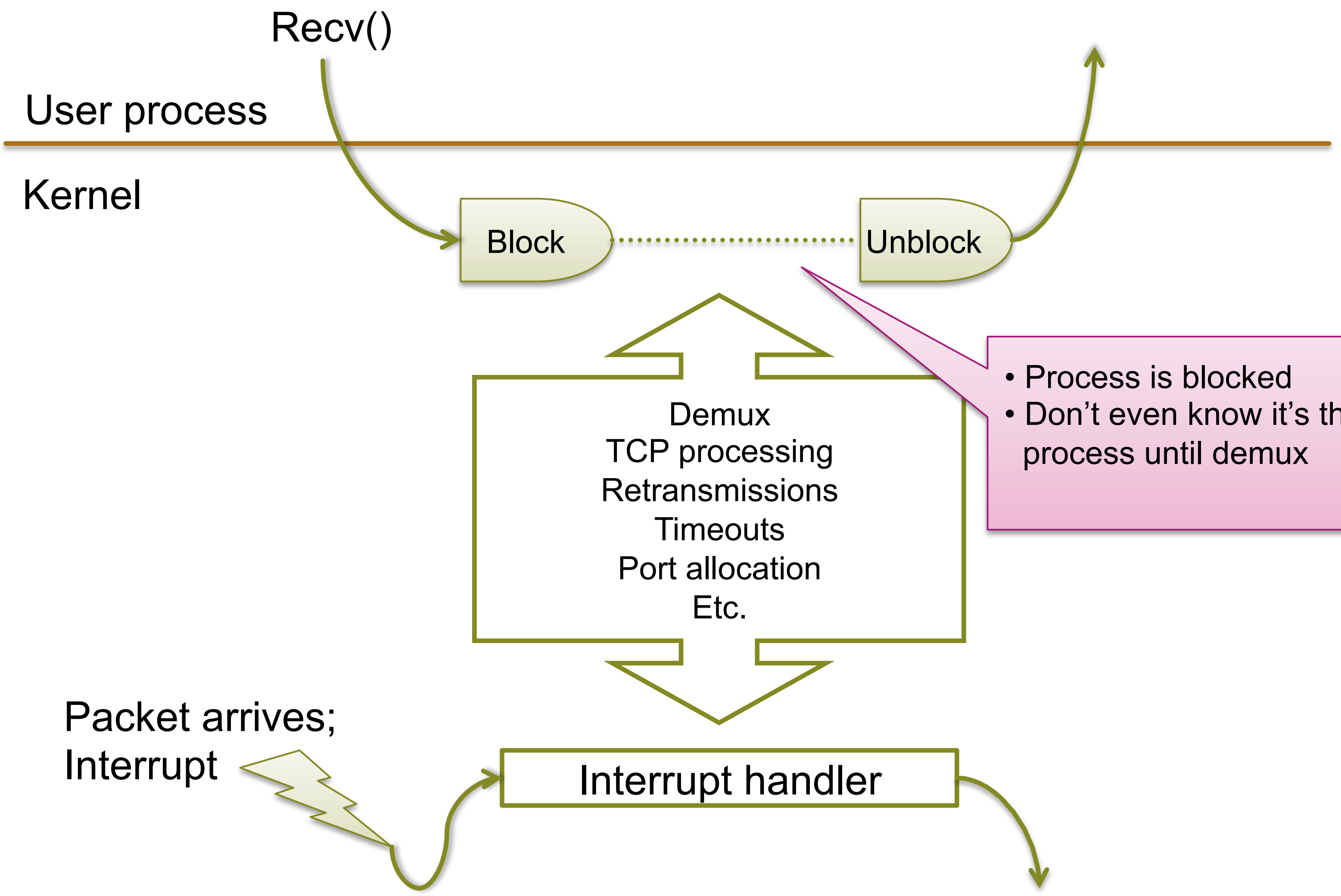
Interrupt handler

Example: network receive



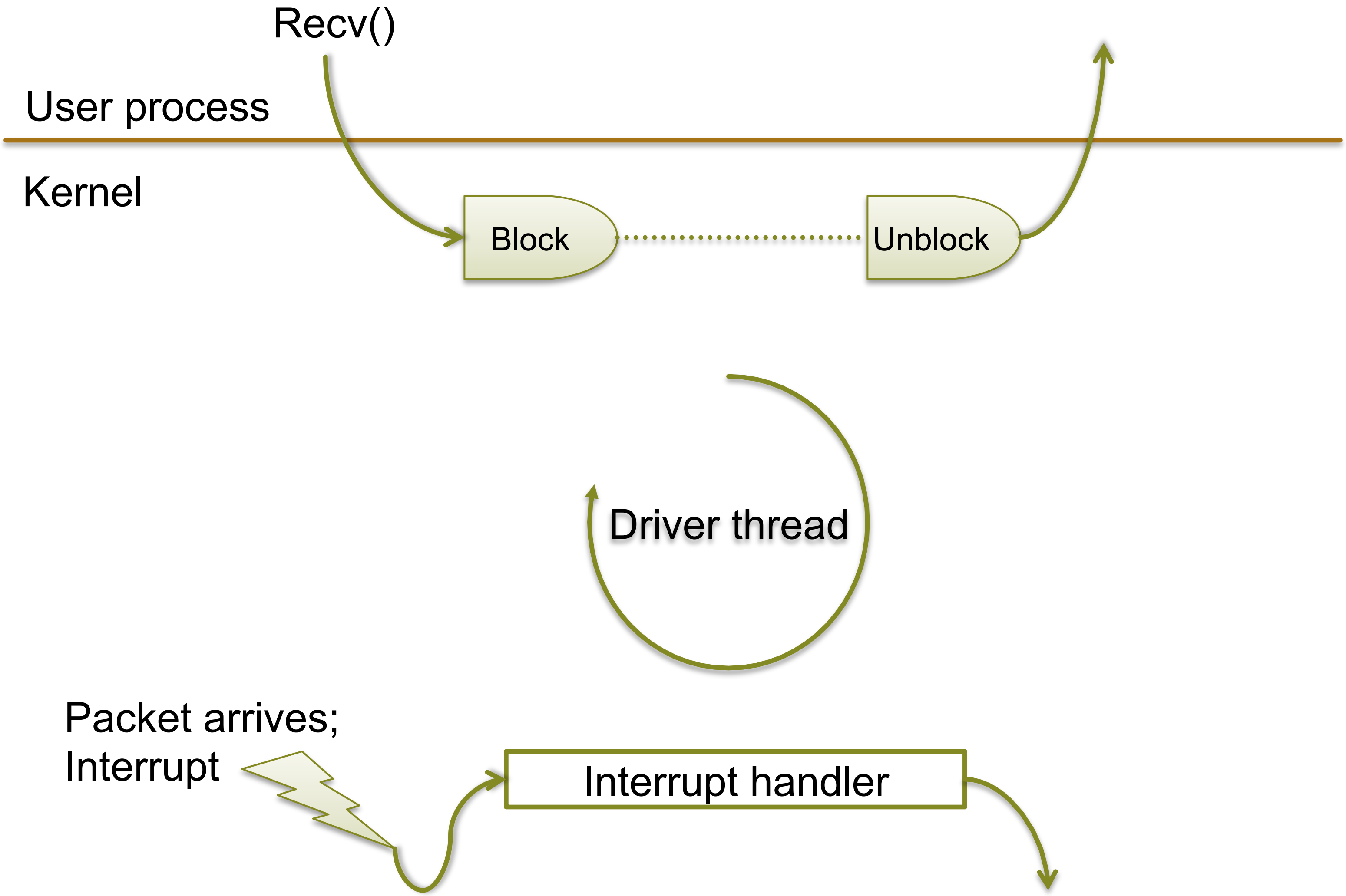
- Can't take too long
  - Interrupts disabled?
- Can't change much
  - Interrupt context
  - Arbitrary system state
  - Can't hold locks

Example: network receive

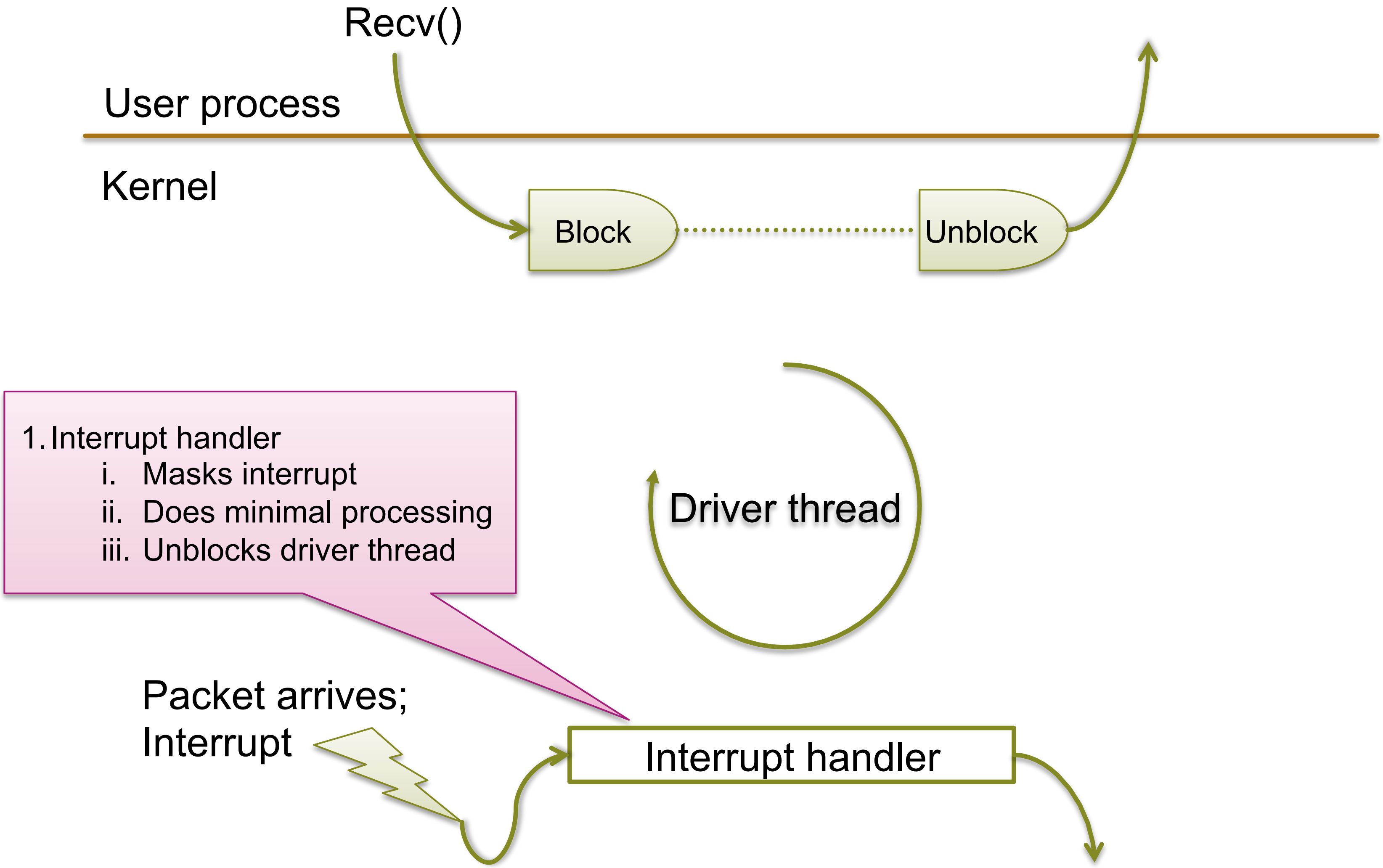


- Process is blocked
- Don't even know it's this process until demux

Solution 1: driver threads

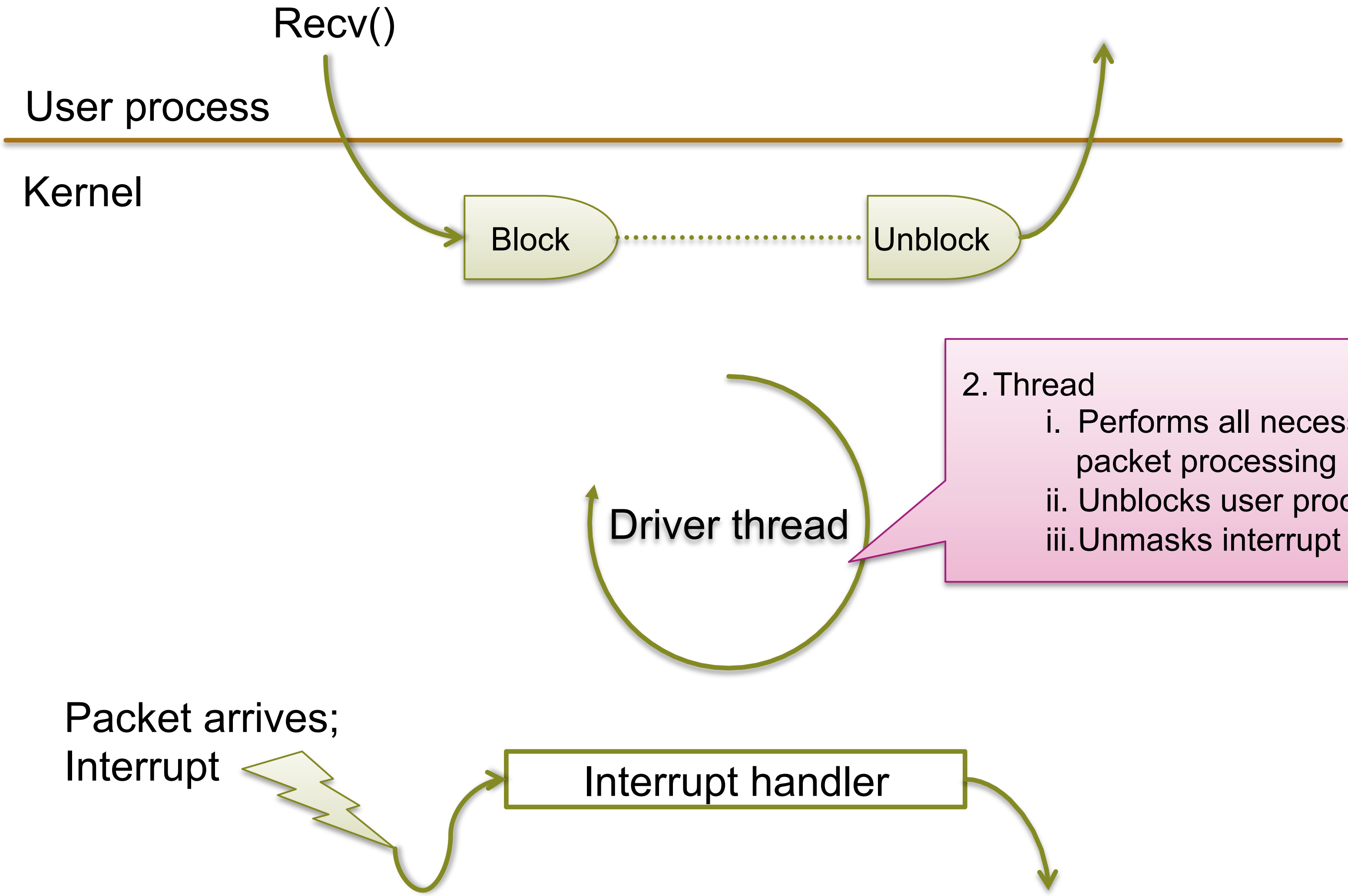


Solution 1: driver threads



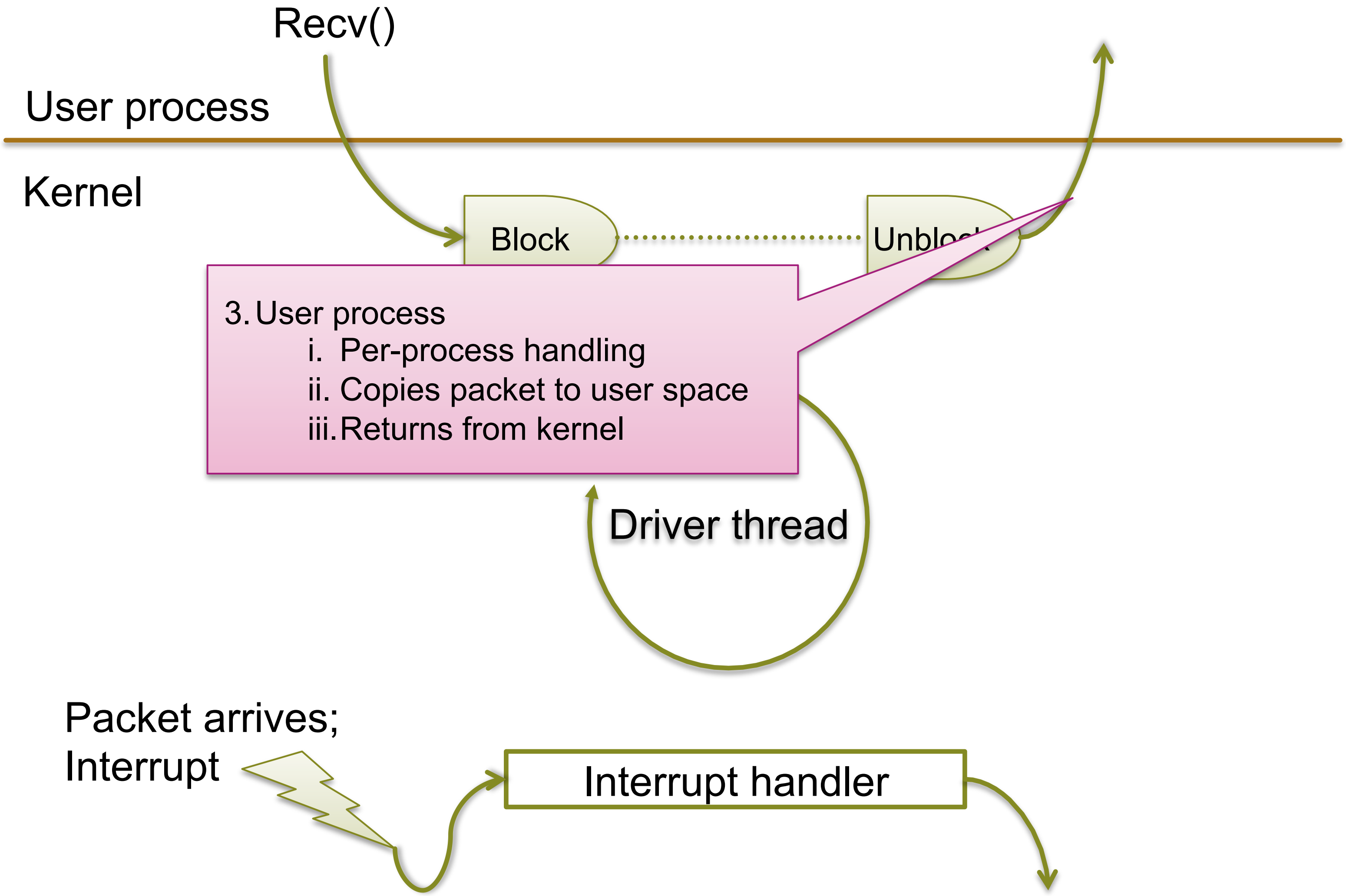
1. Interrupt handler
    1. Masks interrupt
    2. Does minimal processing
    3. Unblocks driver thread

Solution 1: driver threads



2. Thread
    i. Performs all necessary packet processing
    ii. Unblocks user processes
    iii. Unmasks interrupt

Solution 1: driver threads



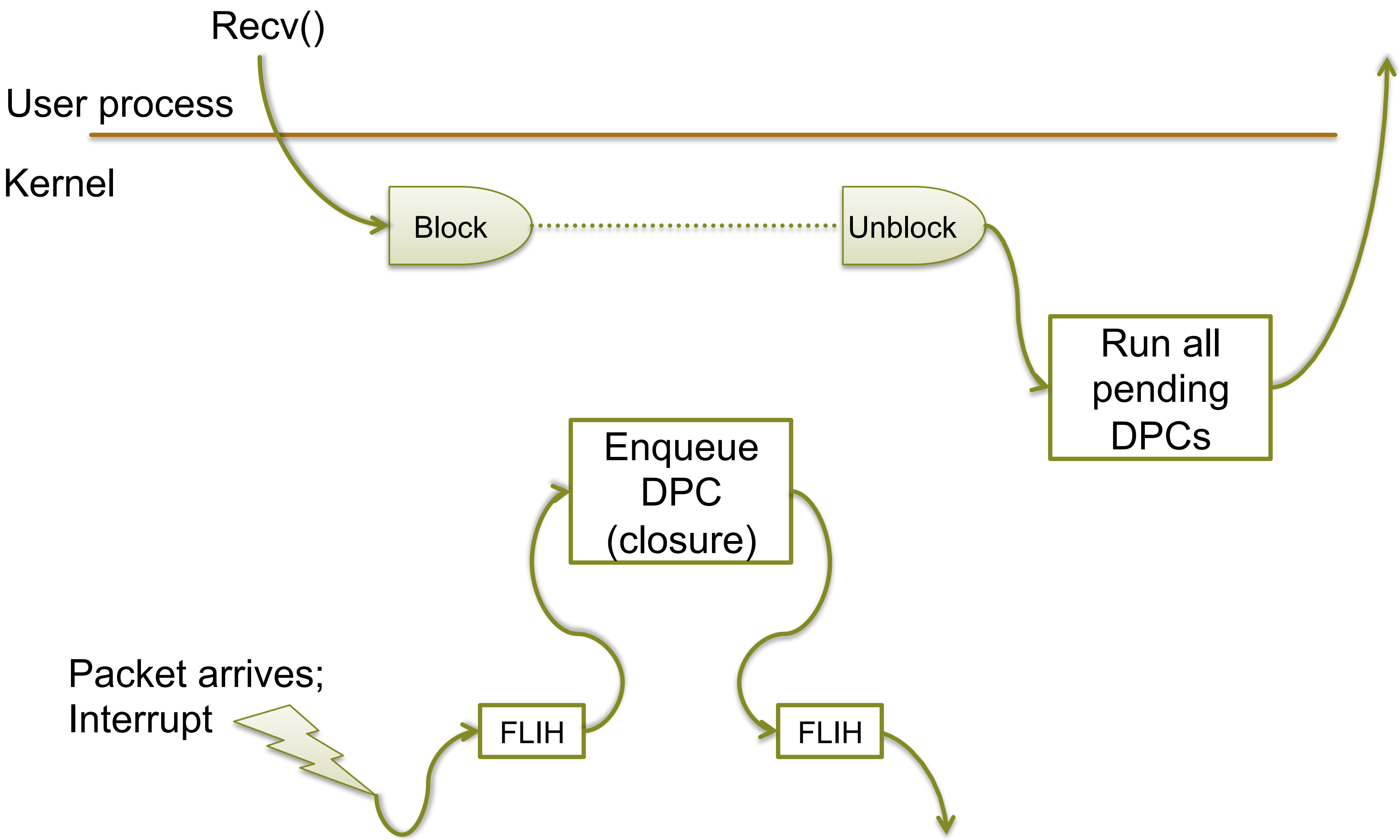
3. User process
    i. Per-process handling
    ii. Copies packet to user space
    iii. Returns from kernel

Terminology – very confused!

- **1^{st}-level Interrupt Handler (FLIH)**
  - Linux calls this the “top half”.
  - In contrast to every other OS on the planet.

- **Thread is an “interrupt handler thread” in Solaris**
  - Other names in other systems… ☹

Solution 2: deferred procedure calls (DPCs)



Deferred Procedure Calls

- **Instead of using a thread, execute on the *next* process to be dispatched**
  - Before it leaves the kernel

- **Solution in most versions of Unix**
  - Don't need kernel threads
  - Saves a context switch
  - Can't account processing time to the right process

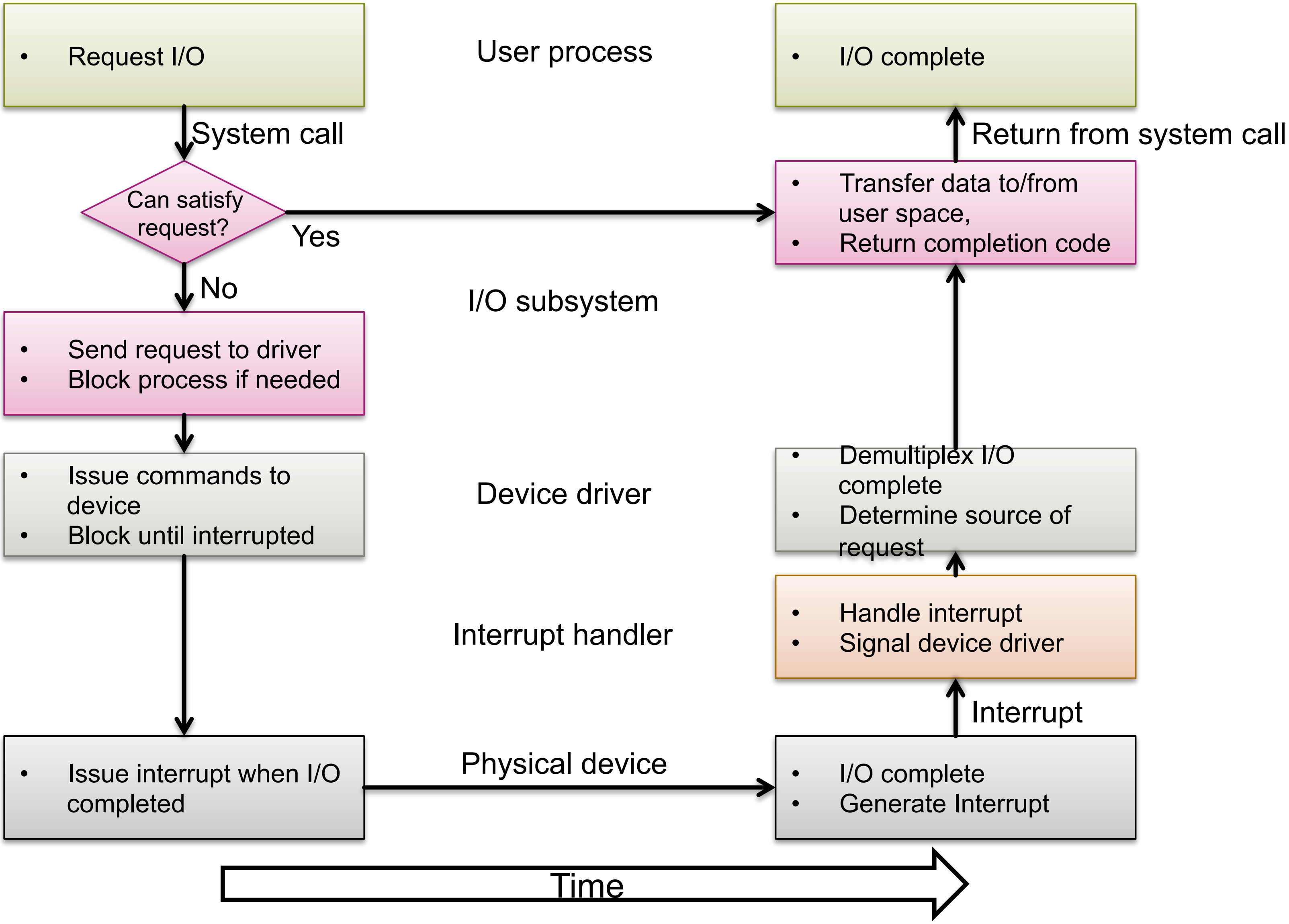
- **∃ 3rd solution: demux early, run in user space**
  - Covered in Advanced OS Course!

More confusing terminology

- **DPCs: also known as:**
  - 2nd-level interrupt handlers
  - Soft interrupt handlers
  - Slow interrupt handlers
  - In Linux ONLY: bottom-half handlers

- **Any non-Linux OS (the way to think about it):**
  - Bottom-half = FLIH + SLIH, called from “below”
  - Top-half = Called from user space (syscalls etc.), “above”

Life Cycle of An I/O Request



The I/O subsystem

Generic I/O functionality

- **Device drivers essentially move data to and from I/O devices**
  - Abstract hardware
  - Manage asynchrony

- **OS I/O subsystem includes generic functions for dealing with this data**
  - Such as…

The I/O Subsystem

- **Caching - fast memory holding copy of data**
  - Always just a copy
  - Key to performance

- **Spooling - hold output for a device**
  - If device can serve only one request at a time
  - E.g., printing

The I/O Subsystem

- **Scheduling**
  - Some I/O request ordering via per-device queue
  - Some OSs try fairness

- **Buffering - store data in memory while transferring between devices or memory**
  - To cope with device speed mismatch
  - To cope with device transfer size mismatch
  - To maintain “copy semantics”

Naming and Discovery

- **What are the devices the OS needs to manage?**
  - Discovery (bus enumeration)
  - Hotplug / unplug events
  - Resource allocation (e.g. PCI BAR programming)
- **How to match driver code to devices?**
  - Driver instance ≠ driver module
  - One driver typically manages many models of device
- **How to name devices inside the kernel?**
- **How to name devices outside the kernel?**

Matching drivers to devices

- **Devices have unique (model) identifiers**
  - E.g. PCI vendor/device identifiers
- **Drivers recognize particular identifiers**
  - Typically a list…
- **Kernel offers a device to each driver in turn**
  - Driver can “claim” a device it can handle
  - Creates driver instance for it.

Naming devices in the Unix kernel

(Actually, naming *device driver instances*)

- **Kernel creates identifiers for**
  - Block devices
  - Character devices
  - *[Network devices – see later…]*

- **Major device number:**
  - Class of device (e.g. disk, CD-ROM, keyboard)

- **Minor device number:**
  - Specific device within a class

Unix Block Devices

- **Used for “structured I/O”**
    - Deal in large “blocks” of data at a time

- **Often look like files (seekable, mappable)**
    - Often use Unix’ shared buffer cache

- **Mountable:**
    - File systems implemented above block devices

Character Devices

- **Used for "unstructured I/O"**
  - Byte-stream interface – no block boundaries
  - Single character or short strings get/put
  - Buffering implemented by libraries

- **Examples:**
  - Keyboards, serial lines, mice

- **Distinction with block devices somewhat arbitrary…**

Naming devices outside the kernel

- **Device files: special type of *file***
  - Inode encodes <type, major num, minor num>
  - Created with `mknod()` system call

- **Devices are traditionally put in `/dev`**
  - `/dev/sda` – First SCSI/SATA/SAS disk
  - `/dev/sda5` – Fifth partition on the above
  - `/dev/cdrom0` – First DVD-ROM drive
  - `/dev/ttyS1` – Second UART

Pseudo-devices in Unix

- **Devices with no hardware!**
- **Still have major/minor device numbers. Examples:**

```
/dev/stdin
/dev/kmem
/dev/random
/dev/null
/dev/loop0
```

etc.

Old-style Unix device configuration

- **All drivers compiled into the kernel**
- **Each driver probes for any supported devices**
- **System administrator populates `/dev`**
    - Manually types `mknod` when a new device is purchased!
- **Pseudo devices similarly hard-wired in kernel**

Linux device configuration today

- **Physical hardware configuration readable from `/sys`**
  - Special fake file system: `sysfs`
  - Plug events delivered by a special socket
- **Drivers dynamically loaded as kernel modules**
  - Initial list given at boot time
  - User-space daemon can load more if required
- **`/dev` populated dynamically by `udev`**
  - User-space daemon which polls `/sys`

Next time:

- Network stack implementation
- Network devices and network I/O
- Buffering
- Memory management in the I/O subsystem