ADRIAN PERRIG & TORSTEN HOEFLER

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

Never underestimate the KISS principle!

52131 Issue with opening Class 377 doors on the Thameslink route

January 2014 in Train Operations

Equipment Infrastructure Rolling stock Station and platform South East

Concerns have been raised about intermittent faults when opening the doors of the Class 377 trains at certain stations on the Thameslink route.

It is reported that at certain times when Drivers attempt to release the doors at the station, the Train Management System (TMS) indicates that the beacons cannot be detected, preventing the doors from opening. The location of the train then needs to be inputted into the TMS, allowing the doors to open. In some instances, even this will not release the doors, and trains have needed to be rebooted. This can take in excess of five minutes, leaving passengers on the train without an exit route.

It is noted that this has happened at many stations on the Brighton to Bedford route, but occurs most frequently at St. Pancras International, City Thameslink, Farringdon, Blackfriars and Brighton. There are concerns that this could delay an emergency exit if an incident were to occur, leaving passengers at risk.

Response from First Capital Connect

FCC would like to thank the reporter for bringing this matter to our attention.

Operation of Class 377 train doors require a Global Positioning Satellite (GPS) signal to identify that the train is in a station to allow the Driver to open the doors. Effectively this prevents the doors being operated in error when the train is not at a station and as such is a safety feature of the trains.

Where the stations are in tunnels, for example St Pancras International low level, and the train cannot receive a GPS signal directly, additional GPS repeater beacons are fitted to the track to relay the signal to the train to enable the Driver to release the doors.

A considerable amount of work has already been done with Network Rail to improve the efficiency of the beacons and this work has also caused a massive reduction in the number of times that the doors do not release first time.

However, we are aware that there are still occasional problems, which results in the Driver having to either manually tell the train where it is via the "location not found" option in the TMS, or in the event of that not working, using the emergency door release option in the train management system.

Initial investigations are pointing towards the signal from the beacon being distorted by an unknown source.

Cache re-load and a magic trick

- **Last time**
  - On-disk data structures
    File representation
    Block allocation
    Directories
  - FAT32 case study
    Very simple block interface
    Single table
  - FFS case study
    Blocked interface
    Uses inodes, direct, (single, double, triple …) indirect blocks
  - NTFS case study
    Extent interface
    Direct and indirect extent pointers

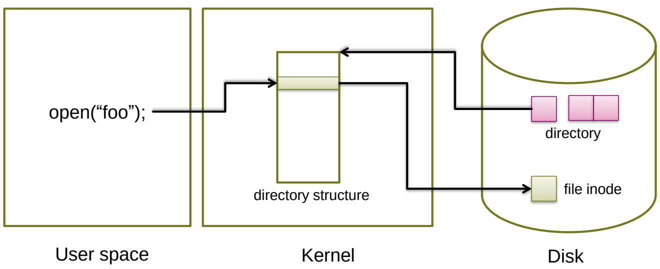
Our small quiz

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

In-memory data structures

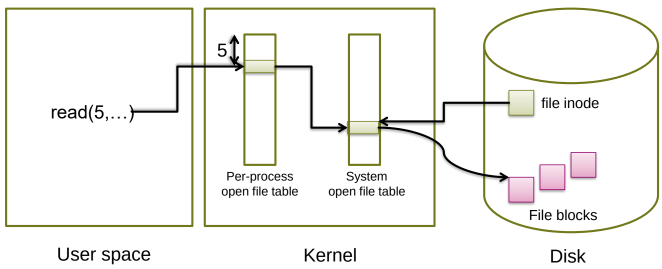
Opening a file

- **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**



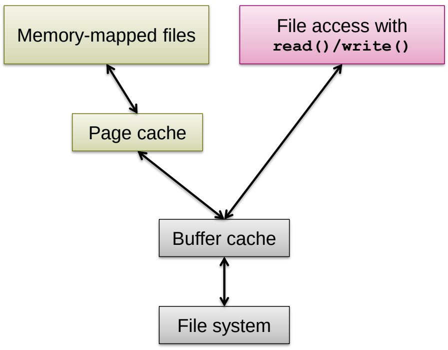
Efficiency and Performance

- **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

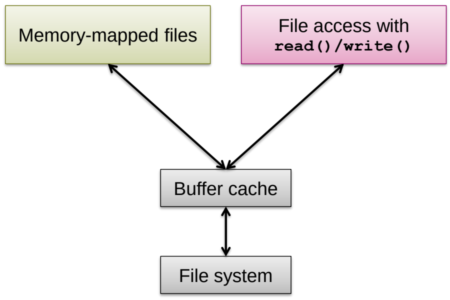
Page Cache

- **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**

Two layers of caching?



Unified Buffer Cache

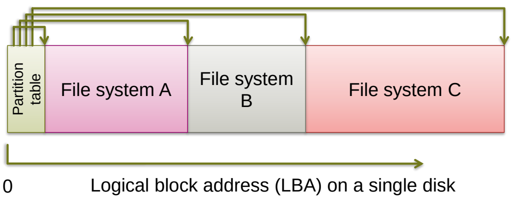


Filesystem Recovery

- **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**

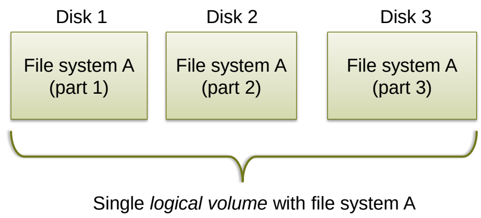
Disks, Partitions and Logical Volumes

Partitions



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

Logical volumes



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

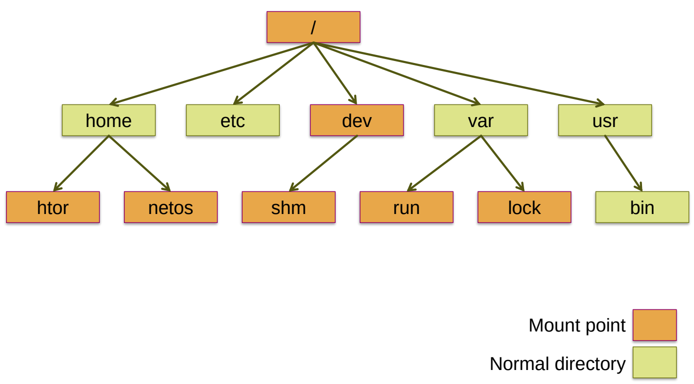
Multiple file systems

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

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:~> 
```

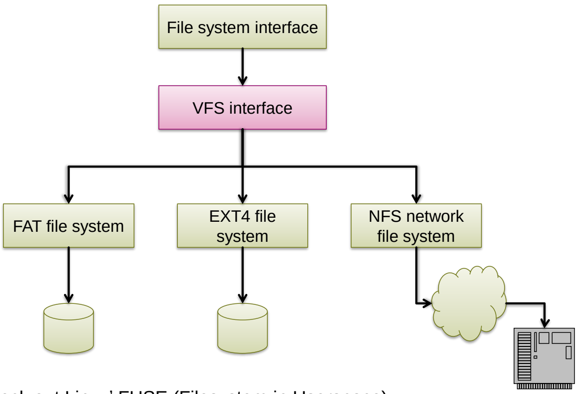

File hierarchy with mounts



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



Advanced: check out Linux' FUSE (Filesystem in Userspace)

ETH zürich spcl.inf.ethz.ch @spcl_eth

Rest of today: I/O

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

ETH zürich spcl.inf.ethz.ch @spcl_eth

Recap from CASP: What does a device look like?

ETH zürich spcl.inf.ethz.ch @spcl_eth

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**

ETH zürich spcl.inf.ethz.ch @spcl_eth

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)

ETH zürich spcl.inf.ethz.ch @spcl_eth

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";
```

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

```c
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);
    }
}
```

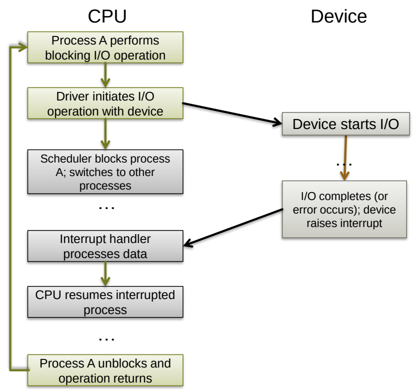

Very simple UART driver

- **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



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!

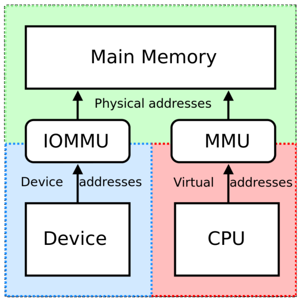
- **Dedicated I/O instructions usually privileged**
- **I/O performed via system calls**
  - Register locations must be protected
- **DMA transfers must be carefully checked**
  - Bypass memory protection!
  - How can that happen today?
    Multiple operating systems on the same machine (e.g., virtualized)
  - 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 :-)

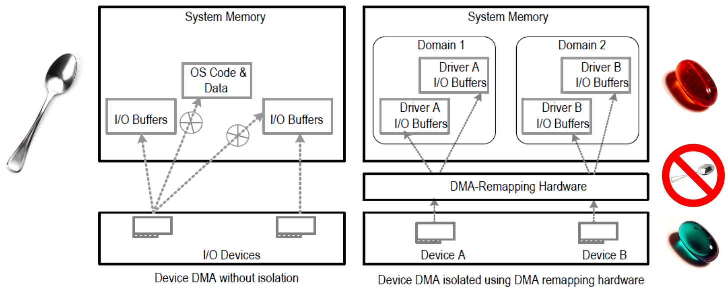
- Examples:
  - Intel VT-d
  - AMD IOMMU
  - ...very similar in functionality



Source: Wikipedia

IOMMUs

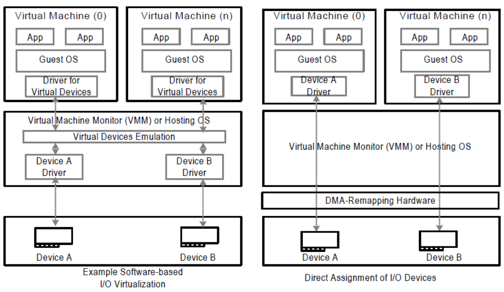
- **Security features for VMs**
  - Possibility to assign different devices to different address **domains**
  - By address remapping we can isolate the domains from one another, thus 'sandboxing' untrusted devices



Source: Intel VT-d specification

IOMMUs

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

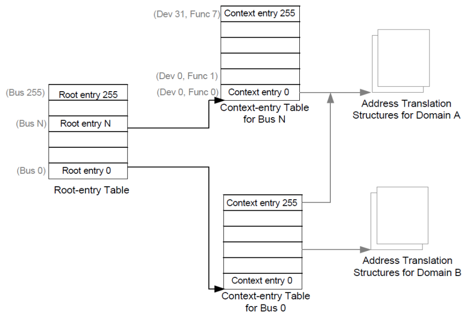


Source: Intel VT-d specification

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**

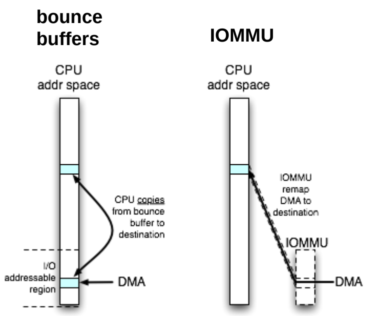


Source: Intel VT-d specification

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)



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

ETH zürich spcl.inf.ethz.ch @spcl_eth

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

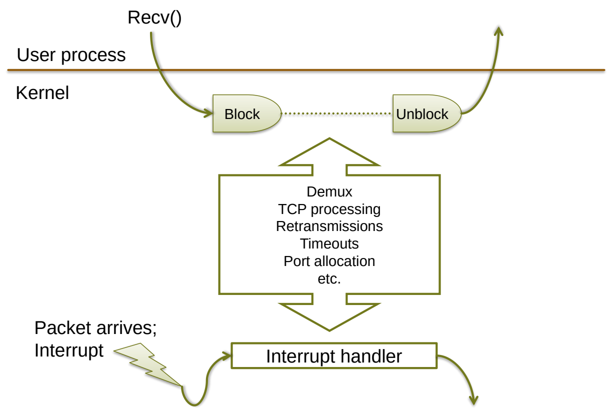
ETH zürich spcl.inf.ethz.ch @spcl_eth

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.

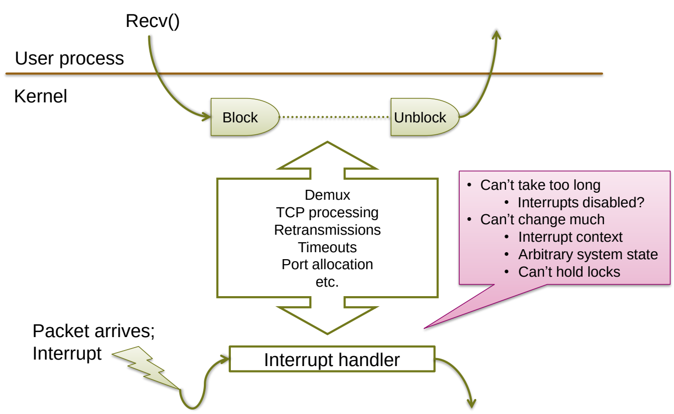
ETH zürich spcl.inf.ethz.ch @spcl_eth

Example: network receive



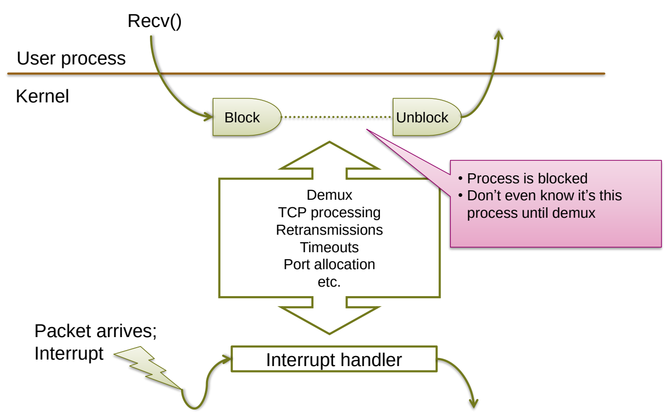
ETH zürich spcl.inf.ethz.ch @spcl_eth

Example: network receive

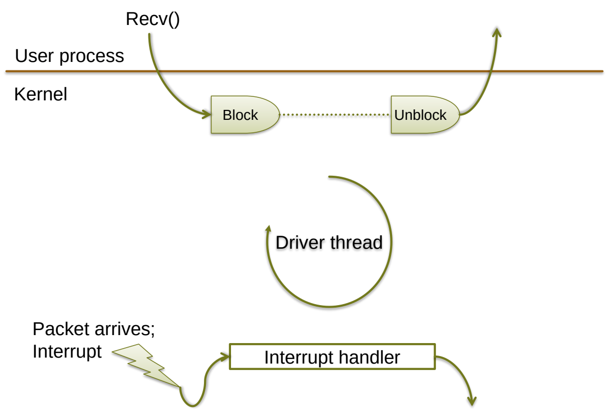


ETH zürich spcl.inf.ethz.ch @spcl_eth

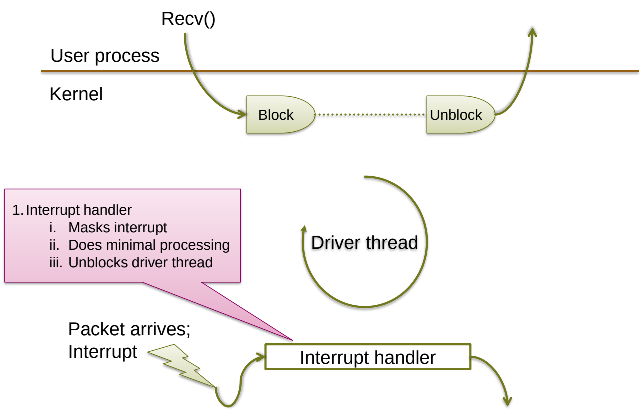
Example: network receive



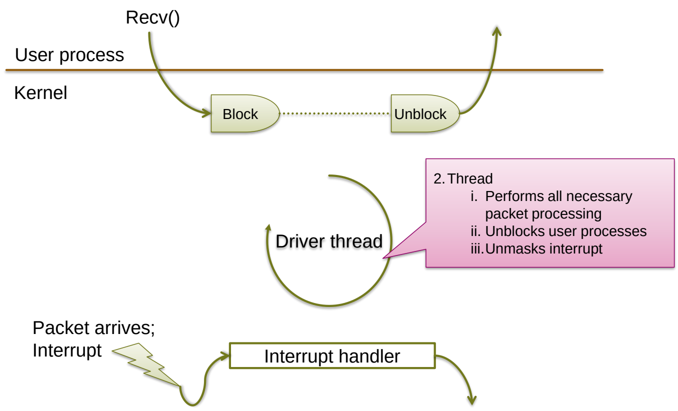
Solution 1: driver threads



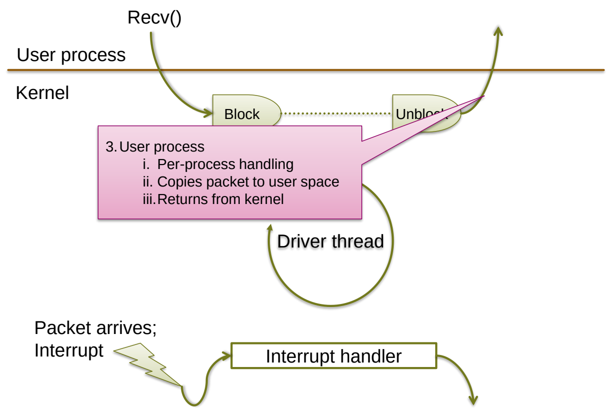
Solution 1: driver threads



Solution 1: driver threads



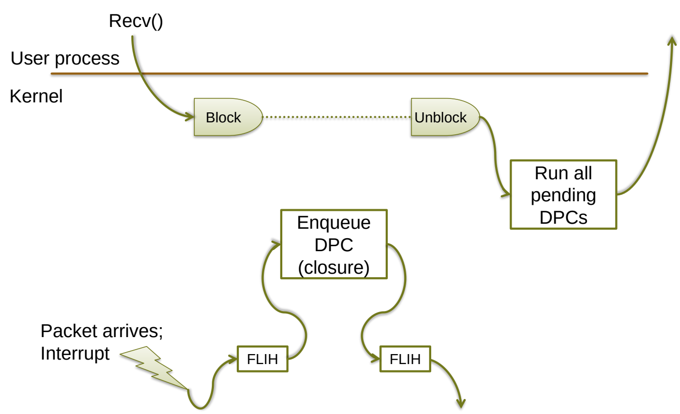
Solution 1: driver threads



Terminology – very confused!

- **1st-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)



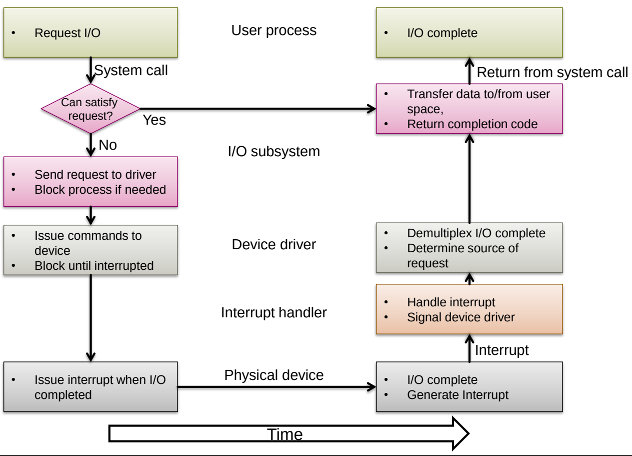
Deferred Procedure Calls

- **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"

ETH zürich spcl.inf.ethz.ch @spcl_eth

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?**

ETH zürich spcl.inf.ethz.ch @spcl_eth

Matching drivers to devices

- **Devices have unique (model) identifiers**
  - E.g., PCI vendor/device identifiers
    Format in Linux (lspci): bus:device.function
- **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

ETH zürich spcl.inf.ethz.ch @spcl_eth

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

ETH zürich spcl.inf.ethz.ch @spcl_eth

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

ETH zürich spcl.inf.ethz.ch @spcl_eth

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 (not a real device anymore)
/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**