# Computer Systems - Notes Week 4

Ruben Schenk, ruben.schenk@inf.ethz.ch

January 2, 2022

# Chapter 7: Input / Output

Every OS has an **I/O subsystem,** which handles all interaction between the machine and the outside worlds. The I/O subsystem abstracts individual hardware devices to present a more or less uniform interface, provides a way to name I/O devices, schedules I/O operations and integrates them with the rest of the system, and contains the low-level code to interface with individual hardware devices.

To an OS programmer, a **device** is a piece of hardware visible from software. It typically occupies some location on a **bus** or I/O interconnect, and exposes a set of hardware **registers** which are either **memory mapped** or in **I/O space.** A device is also usually a source of **interrupts,** and many initiate **Direct Memory Access (DMA)** transfers.

The **device driver** for a particular device is the software in the OS which understands the specific register and descriptor formats, interrupt models, and internal state machines of a given device and abstracts this to the rest of the OS. The driver can be thought of as sitting between hardware and rest of the OS.

## 7.1 Devices and Data Transfer

A **device register** is a physical address location which is used for communicating with a device using reads and writes. A hardware register is not memory, but sits in the physical address space. There is no guarantee that reading from a device register will return the same value that was last written to it.

**Programmed I/O** consists of causing input/output to occur by writing data values to hardware registers from software, or reading values from hardware registers into CPU registers in software.

```
# Algorithm 7.6: Programmed I/O
inputs
 l: the number of words to read from input
 d: buffer of size l
d <- empty buffer
while length(d) < l do
 repeat
     s <- read from status register
 until s indicates data register
 d.append(w)
end while
return
```

An **interrupt** is a signal from a device to a CPU which causes the latter to take an exception and execute an **interrupt service routine (ISR),** also known as an **interrupt handler.**

Using **Direct Memory Access** or DMA, a device can be given a pointer to buffers in main memory and transfer data to and from those buffers without further involvement from the CPU. DMA is typically performed by the device. It saves bandwidth, since the data doesn't need to be copied to through the CPU's register.

## 7.2 Dealing With Asynchrony

Device drivers have to deal with the fundamentally **asynchronous** nature of I/O: the system must respond to unexpected I/O events, or to events which it knows are going to happen, but not when. Input data arrives without warning, and an input operation takes an unknown period of time.

The **First-level Interrupt Service Routine (FLISR)** is the code that executes immediately as a result of the interrupt.

Since I/O is for the most part interrupt-driven, but data is transferred to and from processes which perform explicit operations, data must be *buffered* between the processes and the interrupt handler, and the two must somehow *rendezvous* to exchange data. There are three canonical solutions to this problem: deferred procedure calls, driver threads, and non-blocking kernels.

A **deferred procedure call,** sometimes known as a *2nd-level interrupt handler,* a *soft interrupt handler,* or a *slow interrupt handler,* is a program closure created by the 1st-level interrupt handler. It is run later by any convenient process, typically just before the kernel is exited. DPCs are extremely efficient, and a common solution to the rendezvous problem.

A **driver thread,** sometimes called and *interrupt handler thread,* serves as an intermediary between interrupt service routines and processes. The thread starts blocked waiting for a signal either from the user process or the ISR. When an interrupt occurs or a user process issues a request, the thread is unblocked, and it performs whatever I/O processing is necessary before going back to sleep.

> Remarks: Driver threads are heavyweight: even if they only run in the kernel, they still require a stack and a context switch to and from them to perform any I/O requests. They are however conceptually simple, and can be understood more intuitively than DPCs.

The third alternative, used in microkernels and exokernels, is to have the FLISR convert the interrupt into a message to be sent to the driver process. This is conceptually similar to a DPC, but is even simpler: it simply directs the process to look at the device. However, it does require the FLISR to synthesize an IPC message, which might be expensive.

The part of the device driver code which executes either in the interrupt context or as a result of the interrupt is the **bottom half.** The part of a device driver which is called “from above”, i.e. from a user or an OS process, is the **top half.**

## 7.3 Device Models

The **device model** of an OS is the set of key abstractions that define how devices are represented to the rest of the system by their individual drivers:

- A *character device* in UNIX is used for “unstructured I/O”, and presents a byte-stream interface with no block boundaries. Character devices are accessed by single byte or short string get/put operations. Examples include keyboards, mice, etc.
- A *network device* in UNIX corresponds to a real or virtual network interface adapter. It is accessed through a rather different API to character and block devices.
- A *pseudo-device* is a software service provided by the OS kernel which it is convenient to abstract as a device, even though it does not correspond to a physical piece of hardware.

*Example:* UNIX systems have a variety of pseudo-devices, such as:

- `/dev/mem`: A character device corresponding to the entire main memory of the machine.
- `/dev/random`: Generates a random number when read from.
- `/dev/null`: Anything written is discarded, read always returns end-of-file.

In older UNIX systems, devices were named inside the kernel by a pair of bytes:

- The **major device number** identified the class of device (e.g. disk, CD-ROM, keyboard, etc.)
- The **minor device number** identified a specific device within a class.

In addition, a third “bit” determined whether the device was a character or block device.

Most modern OSes perform **device discovery:** the process of finding and enumerating all hardware devices in the system and storing metadata about them in some kind of queryable data store.

## 7.4 Device Configuration

In addition to simply discovering a device, and finding out how to access it, the OS often has to configure the device and other devices in order to make it work.

*Example:* When a USB device (such as a USB thumb drive) is plugged in, a number of different devices are involved, at the very least:

- The USB drive itself.
- The USB *host bus adapter* or HBA, which interfaces the USB device network to the rest of the computer.
- The USB *hub* that the device was plugged into. This can easily not be a physically separate hub, but one integrated onto the motherboard or built into another device (such as the HBA).

Broadly speaking, when the device is plugged in, the HBA notifies the OS that something has been plugged in. The HBA driver then talks to the HBA to enumerate the devices attached to it, including the hubs - USB is organized approximately as a tree of devices where the non-leaf nodes are hubs. The HBA adapter then has to assign new bus and device identifiers to anything that has changed and reconfigure the HBA and switches. It also discovers the new device by finding out what it is - USB devices, like PCI devices, describe themselves with a 4-byte code.

**Interrupt routing** is the process of configuring *interrupt controllers* in the system to ensure that when a device raises an interrupt, it is delivered to the correct vector on the correct core. Interrupt routing is one of the things that is getting much more complex overtime. It is not unusual for a modern PC to have 4 or 5 interrupt controllers between a device and the CPU.

## 7.5 Naming Devices

One configured, an OS needs a way to refer to devices from user space. This is, of course, a naming problem, and it is important to understand what kind of problem.

On older versions of UNIX, where every device was identified by a `(major, minor)` pair of integers, devices were named using the file system by creating a special kind of file to represent each device, using the `mknod` command. As a preview, the major and minor device numbers were stored in the **inode,** meaning the "device file" took up very little space. Devices are traditionally grouped in the directory `/dev`. For example:

- `/dev/sda`: First SCSI/SATA/SAS disk
- `/dev/sda5`: Fifth partition on the above
- `/dev/cdrom0`: First DVD-ROM drive

## 7.6 Protection

Another function of the I/O subsystem is to perform protection:

- Ensuring that only authorized processes can directly access devices.
- Ensuring that only authorized processes can access the services offered by the device driver.
- Ensuring that a device cannot be configured to do something malicious to the rest of the system.

There are a number of mechanisms for achieving this. Putting device drivers in the kernel makes it easy to control access to the hardware, but you have to trust the device drivers to do the right thing since they are now part of the kernel. UNIX controls access to the drivers themselves by representing them as files, and thereby leveraging the protection model of the file system.

# Chapter 8: Memory Management and Virtual Memory

## 8.1 Segments

Before paging, there were segments. Segments evolved from basic protection mechanisms.

A **base and limit register pair** is a couple of hardware registers containing two addresses $B$ and $L$. A CPU access to an address $a$ is permitted IFF $B \leq a < L$.

A **relocation register** is an enhanced form of base register. All CPU accesses are relocated by adding the offset: a CPU access to an address $a$ is translated to $B + a$ and allowed IFF $B \leq B + a < L$.

Remarks:

- With relocation registers, each program can be compiled to run at the same address, e.g. `0x0000`.
- Relocation registers don't allow sharing code and data between processes, since each process has a single region of memory.

A **segment** is a triple $(I, B_I, L_I)$ of values specifying a contiguous region of memory address space with base $B_I$, limit $L_I$, and an associated *segment identifier* $I$ which names the segment. Memory in a segmented system uses a form of *logical addressing:* each address is a pair $(I, O)$ of segment identifier and offset. A load or store to or from a logical address $(i, o)$ succeeds IFF $0 \leq o < L_i$ and the running process is authorized to access segment $i$. If it does succeed, it will access physical address $B_i + o$.

A **segment table** is an in-memory array of base and limit values $(B_i, L_i)$ indexed by segment identifier, and possibly with additional protection information. The *Memory Management Unit (MMU)* in a segmentation system holds the location and size of this table in a **segment table base register (STBR)** and **segment table length register (STLR).** Logical memory accesses cause the MMU to look up the segment ID in this table to obtain the physical address and protection information.

Remarks:

- Segmentation is fast. As with pages, segment information can be cached in a TLB.
- Sharing is trivially easy at a segment granularity.
- The OS and hardware might have a single, system-wide segment table, or a per-process table.
- The principal downside of segmentation is that segments are still contiguous in physical memory, which leads to external fragmentation.

Paging solves the external fragmentation problem associated with segments, at some cost in efficiency.

A paging system divides the physical address space into fixed-size regions called *frames* or *physical pages,* indexed by a **physical frame (or page) number (PFN),** the high bits of the address. The virtual address space is similarly divided into fixed-size regions called *virtual pages,* identified by a **virtual page number (VPN),** the high bits of the virtual address. The MMU translates from virtual addresses to physical addresses by looking up the VPN in a **page table** to obtain a PFN. Page table entries (PTEs) also hold protection metadata for the virtual page, including validity information. Access via invalid PTE causes a *page fault* processor exception. VPN-to-PFN translations are cached in a **Translation Lookaside Buffer (TLB).**

A **hierarchical page table** is organized in multiple layers, each one translating a different set of bits in the virtual address.

The process of translating a virtual address to physical address using a hierarchical page table is called a **page table walk.** Most MMUs have a *hardware table walker* which is used on TLB-misses to find and load the appropriate page table entry into TLB, but others require the OS to provide a *software page table walker.*

In practice, software in the operating system always has to map virtual to physical addresses:

- A software-loaded TLB simply has no hardware to walk the page table.
- In a case, when a page fault occurs, the OS needs to map the faulting (virtual) address to the relevant PTE, so that it can figure out what kind of fault occurred and also where to find a physical page to satisfy the fault. This requires a table walk.

To facilitate tracking the mappings between virtual and physical addresses, an OS typically divides an address space into a set of contiguous *virtual memory regions.*

## 8.3 Segment Paging

It is possible to combine segmentation and paging: A **page segmentation** memory management scheme is one where memory is addresses by a pair `(segment_id, offset)`, as in a segmentation scheme, but each segment is itself composed of fixed-size pages whose page numbers are then translated to physical page numbers by a paged MMU.

## 8.4 Page Mapping Operations

Each process has its own page table. At high level, all operating systems provide three basic operations on page mappings, which in turn manipulate the page table for a given process:

A page **map** operation on an address space $A$

$$A.map(v, p)$$

takes a virtual page number $v$ and a physical page number $p$, and creates a mapping $v \to p$ in the address space.

A page **unmap** operation on an address space $A$:

$$A.unmap(v)$$

takes a virtual page number $v$ and removes any mapping from $v$ in the address space.

A page **protect** operation on an address space $A$:

$$A.protect(v, rights)$$

takes a virtual page number $v$ and changes the page protection on the page.

An MMU typically allows different protection rights on pages:

- `READABLE`: the process can read from the virtual address
- `WRITABLE`: the process can write to the virtual address
- `EXECUTABLE`: the process can fetch machine code instructions from the virtual address

## 8.5 Copy-On-Write

Recall that the `fork()` operation in UNIX makes a complete copy of the address space of the parent process, and that this might be an expensive operation.

To avoid allocating all the physical pages needed for a process at startup time, **on-demand page allocation** is used to allocate physical pages lazily when they are first touched.

```
# Algorithm 8.16: On-demand page allocation
inputs:
    A {An address space}
    {(v_i), i = 1...n} {A set of virtual pages in A}
    # Setup the region
for i = 1...n do:
    A.unmap(v_i) {Ensure al mappings in region are invalid}
end for
    # Page fault
inputs:
    v' {Faulting virtual address}
    n <- VPN(v')
    p <- AllocateNewPhysicalPage()
    A.map(v' -> p)
return
```

**Copy-on-write** or COW is a technique which optimizes the copying of large regions of virtual memory when the subsequent changes to either copy are expected to be small.

```
# Algorithm 8.18: Copy-On-Write
inputs:
    A_p {Parent address space}
    A_c {Child address space}
    {(v_i, p_i), i = 1...n} {A set of virtual to physical mappings in A_p}
    # Setup
for i = 1...n do:
    A_p.protect(v_i, READONLY)
    A_c.map(v_i -> p_i)
    A_c.protect(v_i, READONLY)
end for
    #Page fault in child
inputs:
    V {VAulting virtal address}
n <- VPN(V)
p' <- AllocateNewPhyiscalPAge()
CopyPageContent(p' <- p_n)
A_c.map(v_n -> p')
A_p.protect(v_n, WRITABLE)
return
```

## 8.6 Managing Caches

Before looking at why and how the OS needs to manage the processor caches, let's go through the operations it can use from software on a cache:

- An **invalidate** operation on a cache (or cache line) marks the contents of the cache (or line) as invalid, effectively discarding the data.
- A **clean** operation on a cache writes any dirty data held in the cache to memory.
- A **flush** operation writes back any dirty data from the cache and then invalidates the line (or the whole cache).

### 8.6.1 Homonyms and Synonyms

**Synonyms** are different cache entries (virtual addresses) that refer to the same physical addresses. Synonyms can result in cache *aliasing,* where the same data appears in several copies in the cache at the same time. Synonyms case problems because an update to one copy in the cache will not necessarily update others, leaving the view of physical memory inconsistent.

In contrast, cache **homonyms** are multiple physical addresses referred to using the same virtual address (for example, in different address spaces). Homonyms are a problem since the cache tag may not uniquely identify cache data, leading to the cache accessing the wrong data.

### 8.6.2 Cache Types

These days, almost all processor caches are write-back, write-allocate, and set-associative. The main differences are to do with where the tag and index bits come from during a lookup.

- A *virtually-indexed, virtually-tagged* or *VIVT* cache is one where the virtual address of the access determines both the cache index and cache tag to lookup. VIVT caches are simple to implement, and fast. However, they suffer from homonyms. The problem can be alleviated by allowing cache entries to be annotated with "address space tags".

**Address-space tags** or *ASIDs* are small additional cache or TLB tags which match different processes or address spaces and therefore allow multiple contexts in a cache or TLB at the same time.

- A *physically-indexed, physically-tagged* or *PIPT* cache is one where the physical address after TLB translation determines the cache index and tag. PIPT caches are easier to manage, since nothing needs to change on a context switch, and they do not suffer from homonyms or synonyms. The downside is that they are slow: you can only start to access a PIPT cache after the TLB has translated the address.

- A *virtually-indexed, physically-tagged* or *VIPT* cache is one where the virtual address before TLB translation determines the cache index, but the physical address after translation gives the tag. VIPT caches are a great choice for L1 D-caches these days, if they can be made to work.
- A *physically-indexed, virtually-tagged* or *PIVT* cache is one where the physical address after TLB translation determines the cache index to look up, but the virtual address before translation supplies the tag to then look up in the set. It is hard to imagine anyone building such a cache, and even harder to figure out why, but they do exist.

## 8.7 Managing The TLB

The **TLB coverage** of a processor is the total number of bytes of virtual address space which can be translated by a TLB at a given point in time. Modern processors are quite complex, with multiple TLBs: primary and secondary TLBs (as with L1 and L2 caches), separate TLBs for instructions and data, and even different TLBs for different page sizes.

**TLB shootdown** on a multiprocessor is the process of ensuring that no out-of-date virtual-to-physical translations are held in any TLB in the system following a change to a page table. Since the TLB is a cache, it should be coherent with other TLBs in the system, and with each process' page tables. Shootdown is basically a way to invalidate certain mappings in every TLB, if they refer to the affected process.