NTNU | Norwegian University of Science and Technology

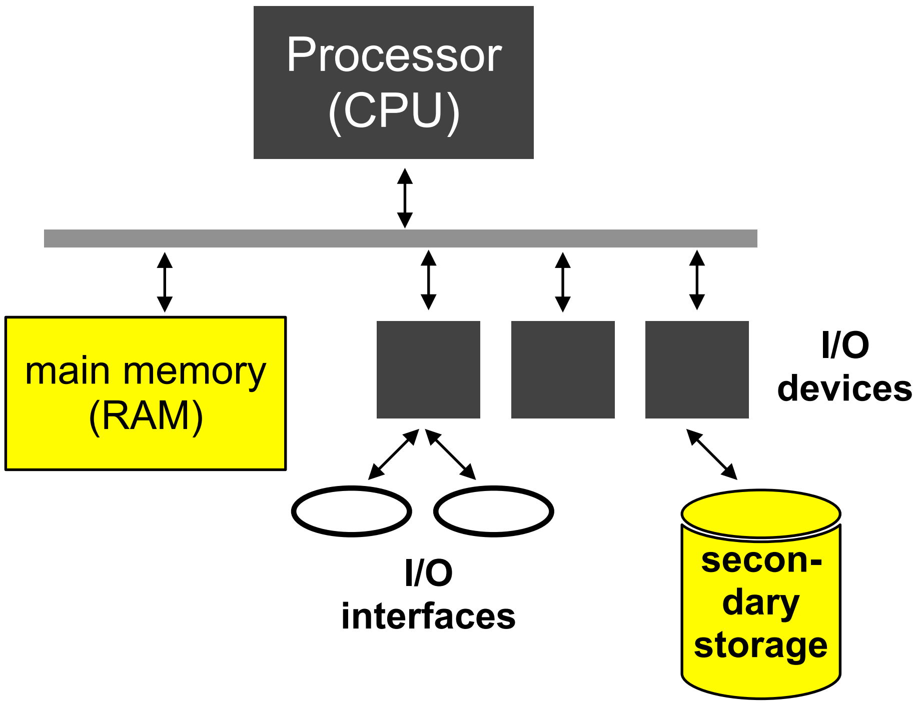
Operating Systems

Lecture 14: Input and output

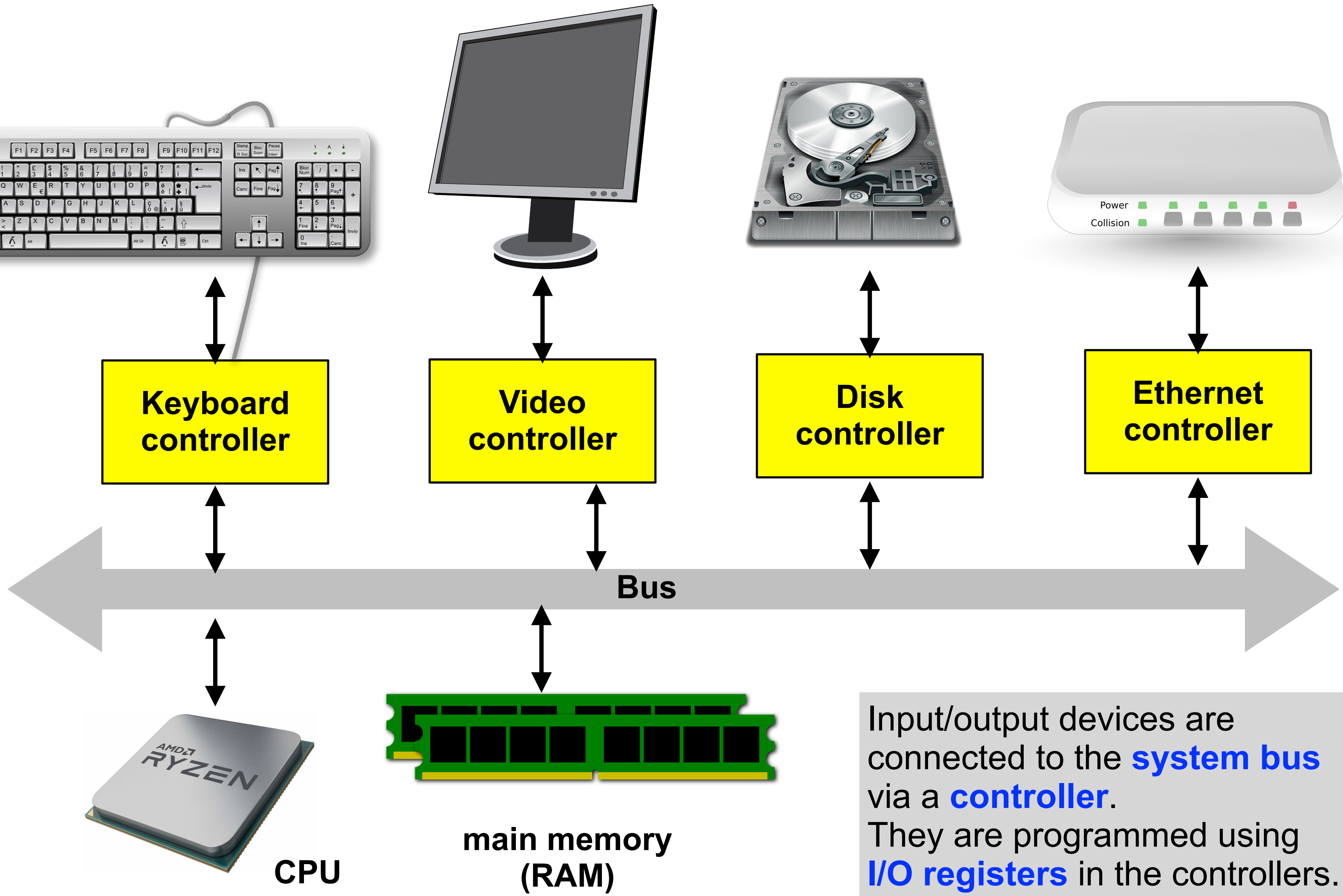
Michael Engel

Resources

- So far we considered:
  - CPU
  - main memory
- In the next lecture
  - background storage
- **Today: I/O devices**



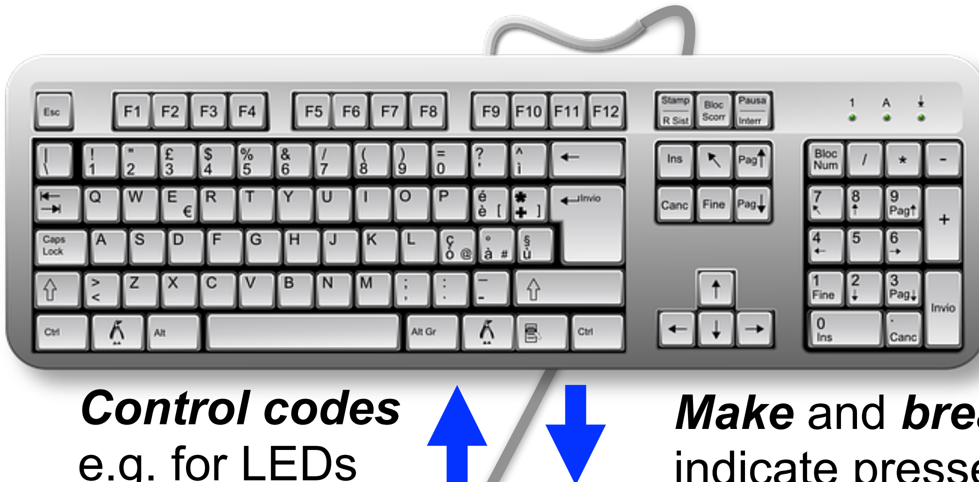
I/O device interfacing



Input/output devices are connected to the **system bus** via a **controller**.
They are programmed using **I/O registers** in the controllers.

Example: PC keyboard

- Serial communication, character oriented
  - Keyboards are "intelligent" (have their own processor)



Control codes
e.g. for LEDs

Make and ***break codes*** indicate pressed/released keys

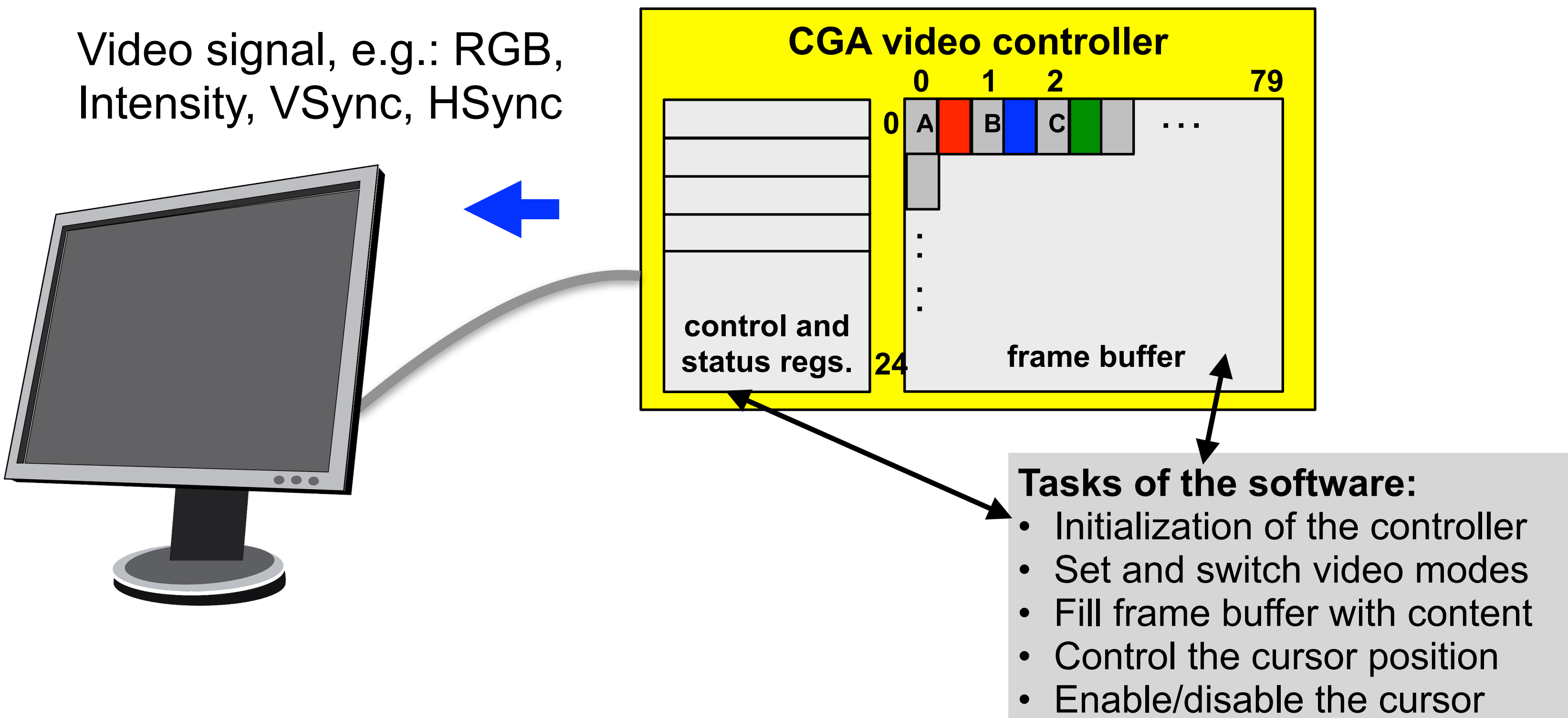
keyboard controller

Tasks of the software:
- Initialization of the controller
- Fetch characters from the keyboard
- Map the *make* and *break* codes to ASCII
- Send command (e.g. for switching LEDs)

The controller signals an ***interrupt*** as soon as a character is available

Example: CGA video controller

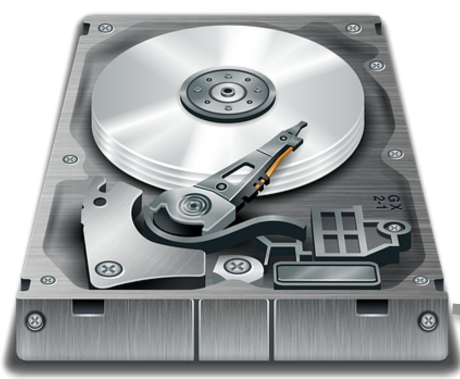
- Communication via video signal
  - analog: VGA, digital: DVI, HDMI, DisplayPort
- Transformation of the contents of the ***frame buffer*** (screen memory) into a picture (e.g. 80x25 character matrix or bitmap)

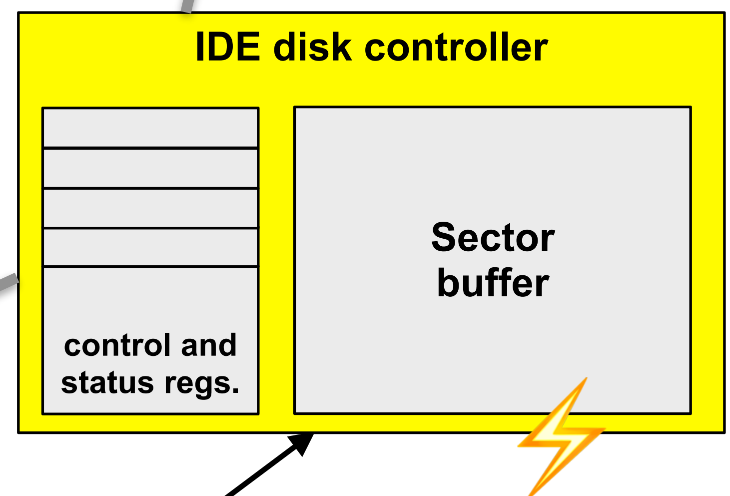


Example: IDE disk controller

- Communication via AT commands
  - **Blockwise random access** to data blocks

AT commands
- Calibrate disk drive
- Read/write/verify block
- Format track
- Position disk read/write head
- Diagnosis
- Configure disk parameters





Data blocks
(512 bytes each)

Tasks of the software:
- Write AT commands to registers
- Fill/empty sector buffer
- React to interrupts
- Error handling

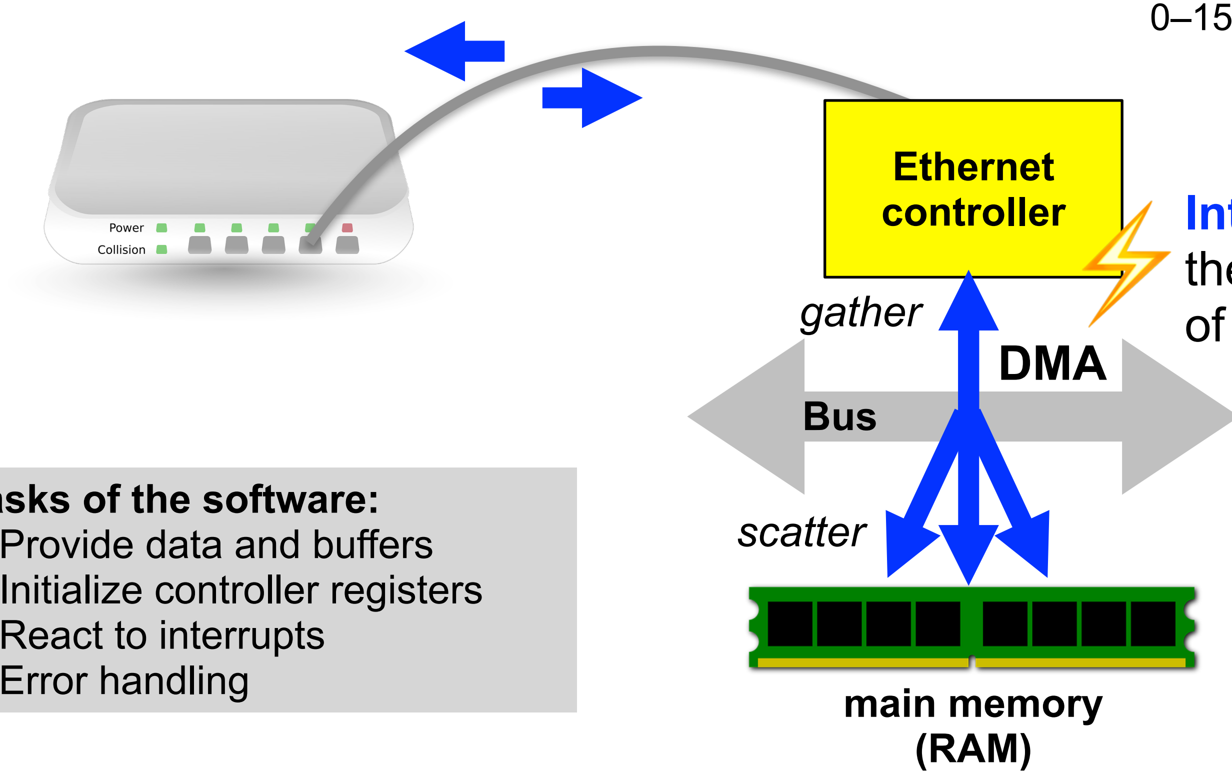
The disk controller signals an interrupt as soon as the sector buffer has been read or written

Example: Ethernet controller

- Serial packet-based bus communication
  - Packets have variable size and contain addresses:

preamble+SFD	destination	source	type	data ………………	CRC

0–1500 bytes (data)



Interrupt after the completion of each operation

Tasks of the software:
- Provide data and buffers
- Initialize controller registers
- React to interrupts
- Error handling

Memory access via ***scatter/gather*** bus master **DMA**

Device classes

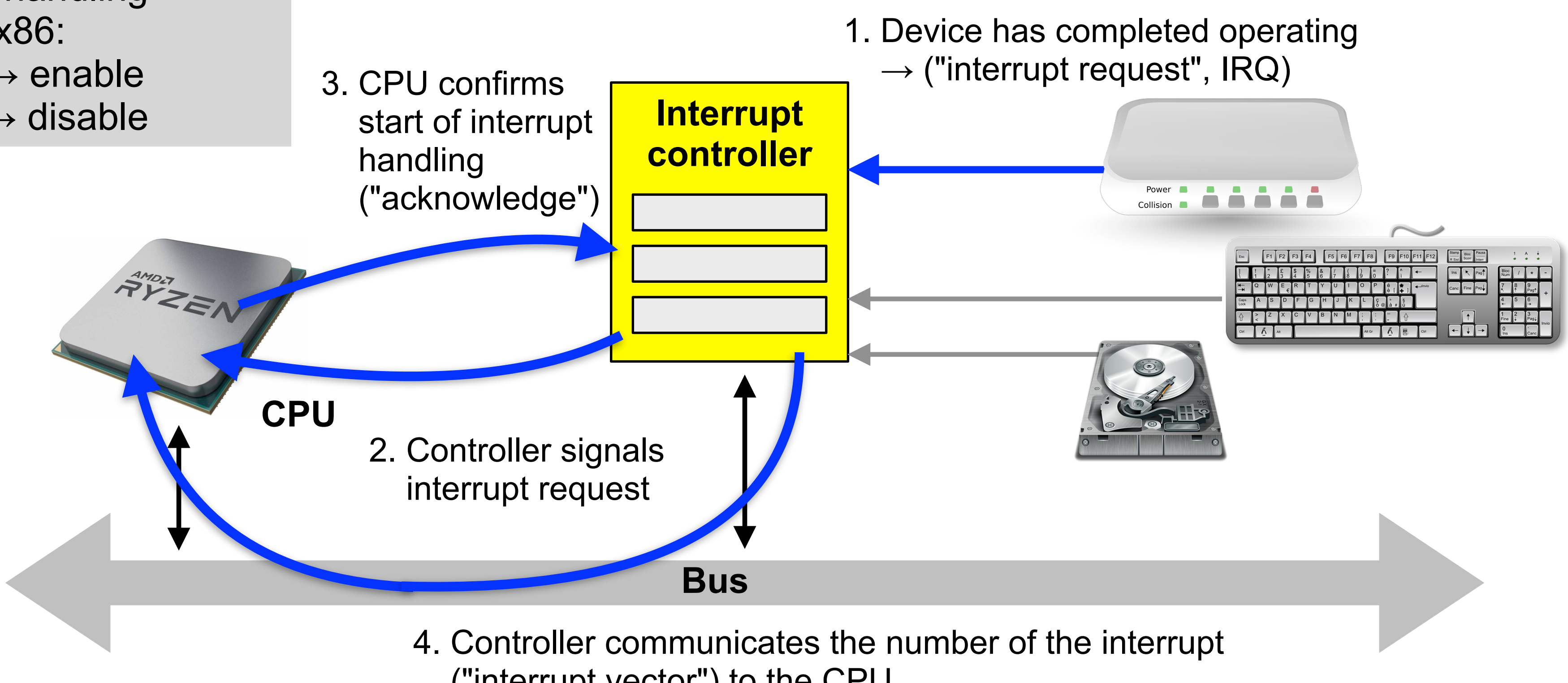
- **Character devices**
  - Keyboard, printer, modem, mouse, ...
  - Usually only **sequential access**, rarely random access
- **Block devices**
  - Hard disk, CD-ROM, DVD, tape drives, ...
  - Usually **blockwise random access**
- **Other devices** don’t fit this scheme easily, such as
  - (GP)GPUs (especially 3D acceleration)
  - Network cards (protocols, addressing, broadcast/multicast, packet filtering, ...)
  - Timer (sporadic or periodic interrupts)
  - ...

Interrupts

- …signal the software to become active

Interrupt processing sequence on hardware level

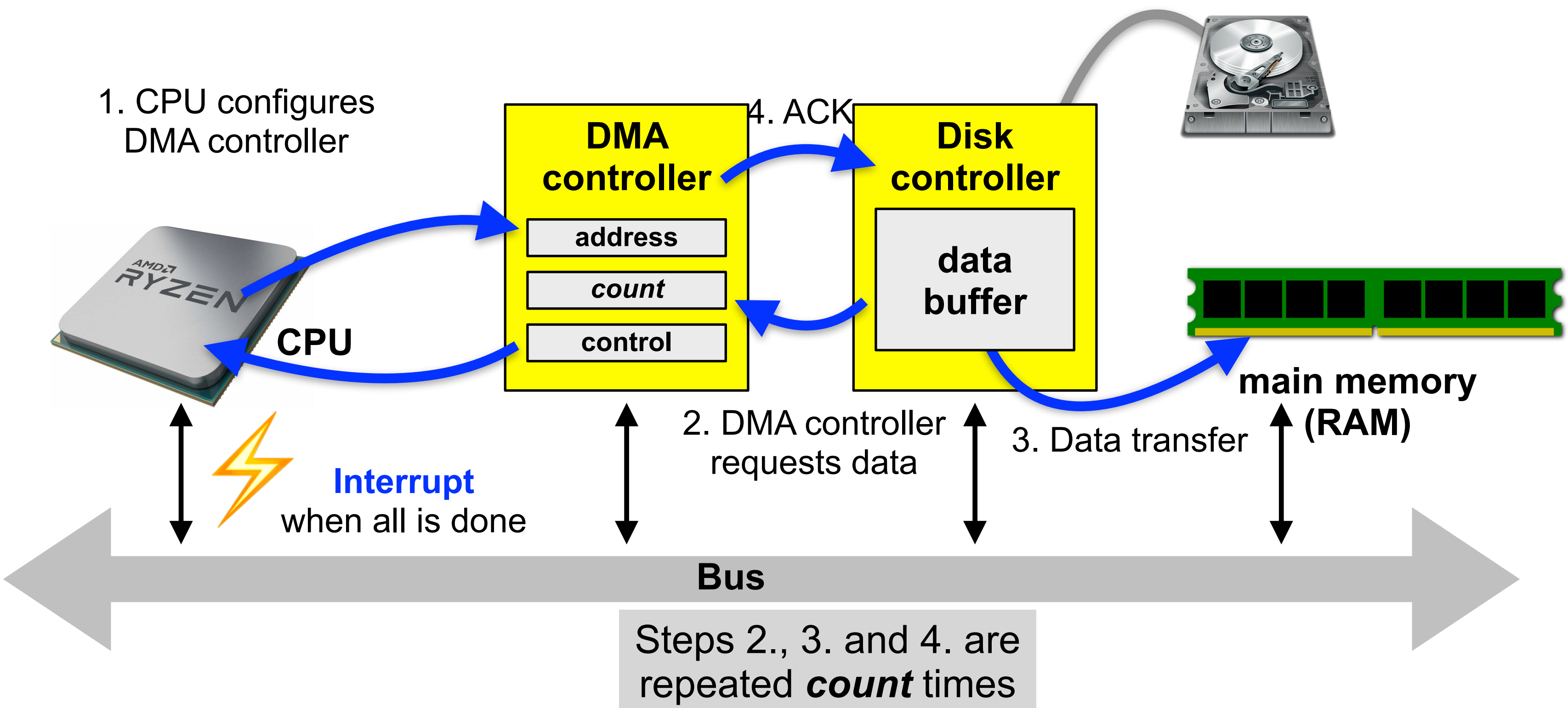
Software can disable IRQ handling
For x86:
sti → enable
cli → disable



Direct Memory Access (DMA) ...

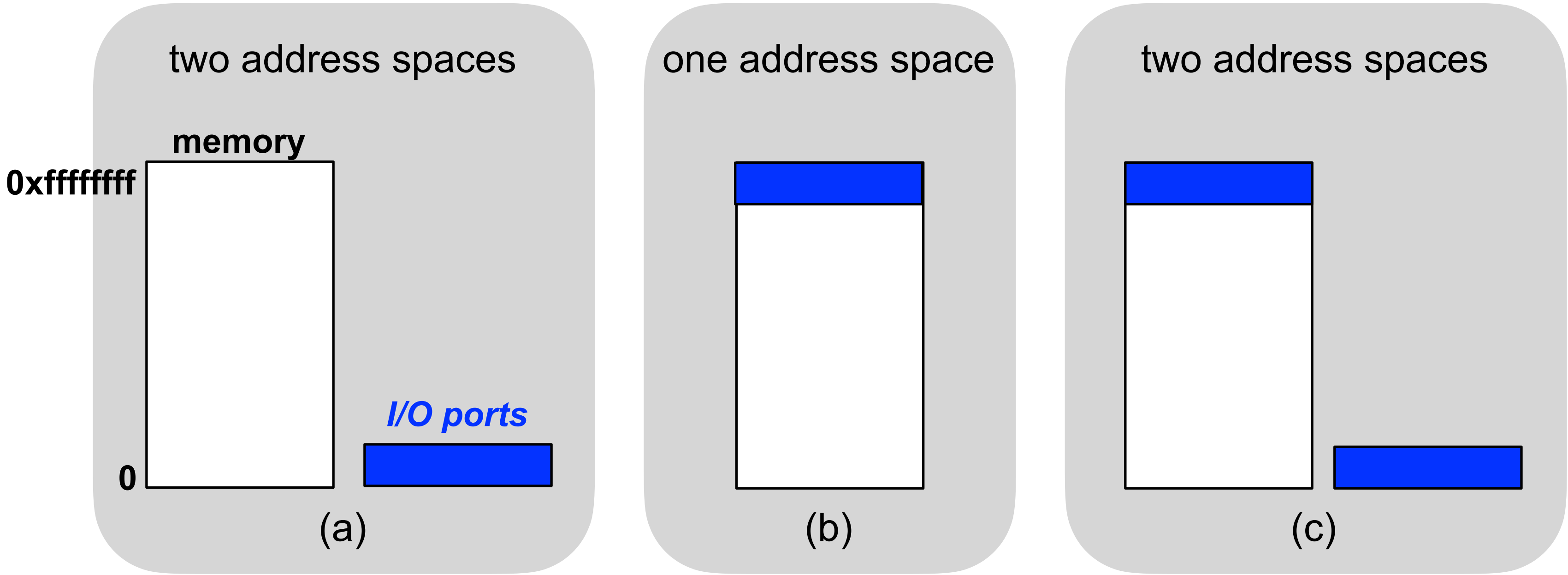
- is used by complex controllers to transfer data from and to main memory independent of the CPU

DMA transfer sequence



I/O address space

- Access to *controller registers* and *controller memory* depends on the system architecture



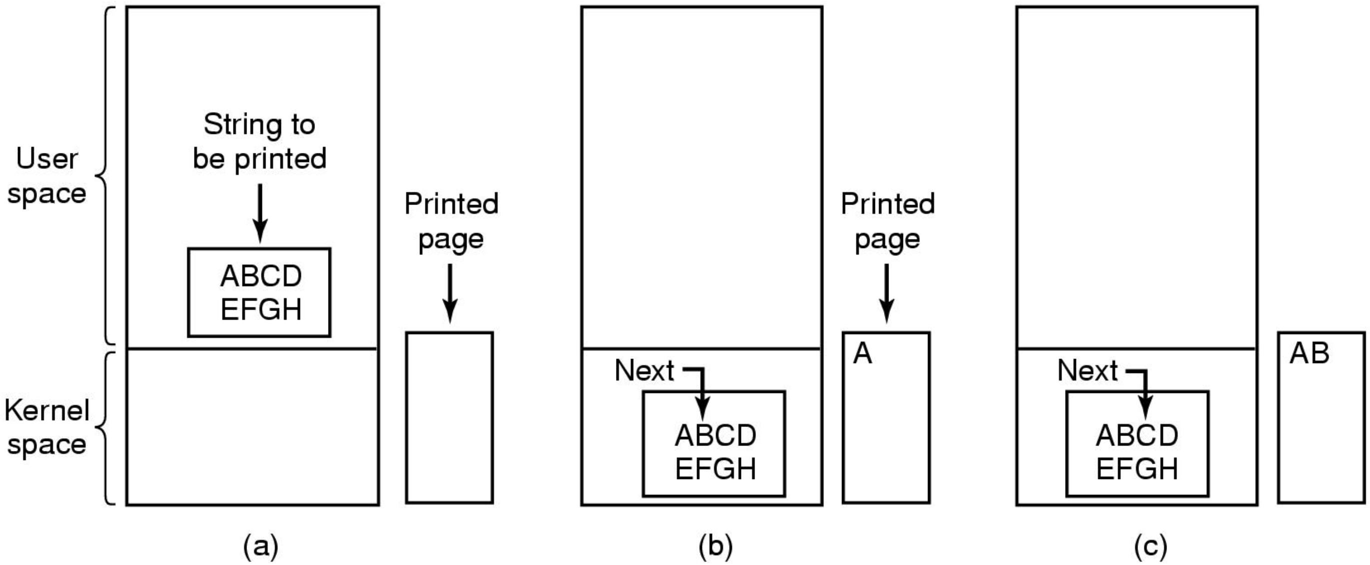
(a) Separate I/O address space
  - accessible using special machine instructions

(b) Shared address space (***memory mapped I/O***)

(c) Hybrid architecture

Device drivers

- Depending on the device, I/O can be performed via
  - ***Polling*** ("programmed I/O"),
  - ***Interrupts*** or
  - ***DMA***
- Example: Printing a page of text



Source: Tanenbaum, Modern Operating Systems

Polling ("programmed I/O")

... implies ***active waiting*** for an I/O device

```
/* Copy character into kernel buffer p */
copy_from_user (buffer, p, count);

/* Loop over all characters */
for (i=0; i<count; i++) {

  /* Wait "actively" until printer is ready */
  while (*printer_status_reg != READY);

  /* Print one character */
  *printer_data_reg = p[i];
}

return_to_user ();
```

Pseudo code of an operating system function to print text using polling

Interrupt-driven I/O

... implies that the CPU can be allocated to another process while waiting for a response from the device

```c
copy_from_user (buffer, p, count);

/* Enable printer interrupts */
enable_interrupts ();

/* Wait until printer is ready */
while (*printer_status_reg != READY);

/* Print first character */
*printer_data_reg = p[i++];

scheduler ();
return_to_user ();
```

Code to initiate the I/O operation

```c
if (count > 0) {

  *printer_data_reg = p[i];
  count--;
  i++;

} else {

  unblock_user ();

}
acknowledge_interrupt ();
return_from_interrupt ();
```

Interrupt handler

DMA-driven I/O

... the CPU is no longer responsible for transferring data between the I/O device and main memory

- further reduction of CPU load

```
copy_from_user (buffer, p, count);
set_up_DMA_controller (p, count);
scheduler ();
return_to_user ();
```

Code to initiate the I/O operation

```
acknowledge_interrupt ();
unblock_user ();
return_from_interrupt ();
```

Interrupt handler

Discussion: Interrupts

- Saving the process context
  - Partly performed directly by the CPU
    - e.g. saving status register and return address
    - minimal required functionality
  - All modified registers have to be saved before and restored after the end of interrupt processing

- Keep interrupt processing times short
  - Usually other interrupts are disabled while an interrupt handler is executed
    - Interrupts can get lost
  - If possible, the OS should only wake up the process that was waiting for the I/O operation to finish

Discussion: Interrupts (2)

- Interrupts are ***the*** source for asynchronous behavior
  - Can cause ***race conditions*** in the OS kernel
- Interrupt synchronization
  - Simple approach: disable interrupts "hard" while a critical section is executed
    - x86: instructions sti and cli
    - Again, interrupts could get lost
  - In modern systems, interrupts are realized using multiple stages. These minimize the amount of time spent with disabled interrupt
    - UNIX: top half, bottom half
    - Linux: Tasklets
    - Windows: Deferred Procedures

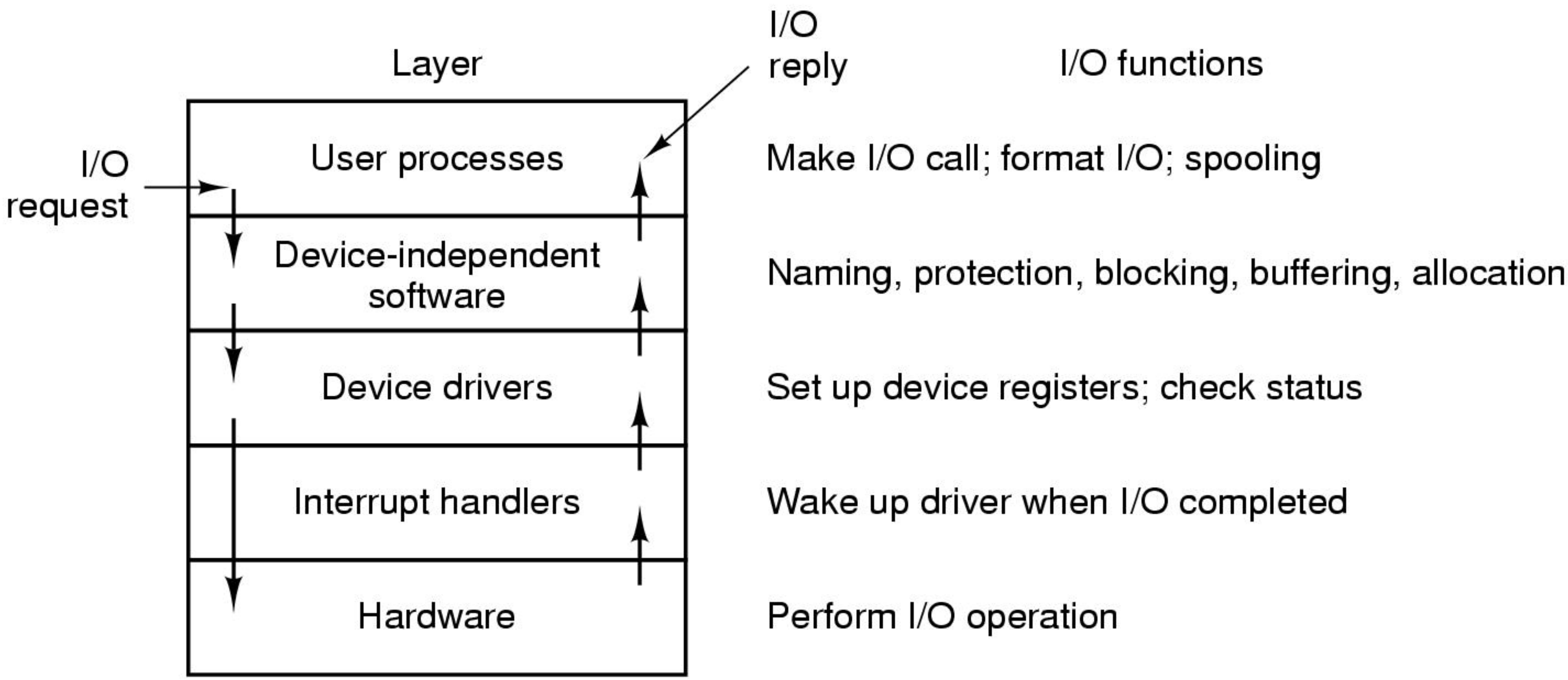
Discussion: Direct Memory Access

- **Caches**
  - Modern processors use *data caches*
    DMA bypasses the cache!
  - Before a DMA transfer is configured, cache contents must be written back to main memoy and the cache invalidated
    - Some processors support non-cachable address ranges for I/O operations
- **Memory protection**
  - Modern processors use a MMU to isolate processes from each other and to protect the OS itself
    DMA bypasses memory protection!
  - Mistakes setting up DMA transfers are very critical
  - Application processes can never have direct access to program the DMA controller!

Tasks of the OS

- Create **device abstrations**
  - Uniform, simple, but versatile
- Provide **I/O primitives**
  - Synchronous and/or asynchronous
- **Buffering**
  - If the device or the receiving process are not yet ready
- **Device control**
  - As efficient as possible considering mechanical device properties
- Handle **resource allocation**
  - For multiple access devices: which process may read/write where?
  - For single access devices: time-limited reservation
- Manage **power saving modes**
- Support **plug&play**
- ...

Layers of the I/O system



Source: Tanenbaum, "Modern Operating Systems"

Unix device abstractions

- Peripheral devices are realized as ***special files***
  - Devices can be accessed using read and write operations in the same way as regular files
  - Opening special files creates a connection to the respective device provided by the **device driver**
  - Direct access to the driver by the user

- **Block oriented special files** (block devices)
  - Disk drives, tape drives, floppy disks, CD-ROMs

- **Character oriented special files** (character devices)
  - Serial interfaces, printers, audio channels etc.

Unix device abstractions (2)

- Devices are uniquely identified by a tuple:
  - **device type**
    - block or character device
  - **major device number**
    - selects one specific *device driver*
  - **minor device number**
    - selects one of multiple devices controlled by the device driver identified by the major number

Unix device abstractions (3)

- Partial listing of the /dev directory that by convention holds the special files:

```
brw-rw---- me   disk 3,  0 2008-06-15 14:14 /dev/hda
brw-rw---- me   disk 3, 64 2008-06-15 14:14 /dev/hdb
brw-r----- root disk 8,  0 2008-06-15 14:13 /dev/sda
brw-r----- root disk 8,  1 2008-06-15 14:13 /dev/sda1
crw-rw---- root uucp 4, 64 2006-05-02 08:45 /dev/ttyS0
crw-rw---- root lp   6,  0 2008-06-15 14:13 /dev/lp0
crw-rw-rw- root root 1,  3 2006-05-02 08:45 /dev/null
lrwxrwxrwx root root     3 2008-06-15 14:14 /dev/cdrecorder -> hdb
lrwxrwxrwx root root     3 2008-06-15 14:14 /dev/cdrom -> hda
```

access permissions

owner and group

major and minor ID

modification date&time

name of the special file

`c`: *character device*

`b`: *block device*

`l`: *link*

Unix access primitives

A quick overview... (see the man pages for details...)

- `int open(const char *devname, int flags)`
  - "opens" a device and returns a *file descriptor*
- `off_t lseek(int fd, off_t offset, int whence)`
  - Positions the read/write pointer (relative to the start of the file) – only for random access files
- `ssize_t read(int fd, void *buf, size_t count)`
  - Reads at most `count` bytes from descriptor `fd` into buffer `buf`
- `ssize_t write(int fd, const void *buf, size_t count)`
  - Writes `count` bytes from buffer `buf` to file with descriptor `fd`
- `int close(int fd)`
  - "closes" a device. The file descriptor `fd` can no longer be used after `close`

Unix device specific functions

- Special properties of a devices are controlled via `ioctl`:

```
IOCTL(2)            Linux Programmer's Manual            IOCTL(2)

NAME
       ioctl - control device

SYNOPSIS
       #include <sys/ioctl.h>

       int ioctl(int d, int request, ...);
```

- Generic interface, but device-specific semantics:

```
CONFORMING TO
    No single standard. Arguments, returns, and semantics of
    ioctl(2) vary according to the device driver in question
    (the call is used as a catch-all for operations that
    don't cleanly fit the Unix stream I/O model). The ioctl
    function call appeared in Version 7 AT&T Unix.
```

Unix: waiting for multiple devices

- So far, we have encountered ***blocking*** read and write calls
  - What can we do if we need to read from several sources (devices, files) at the same time?

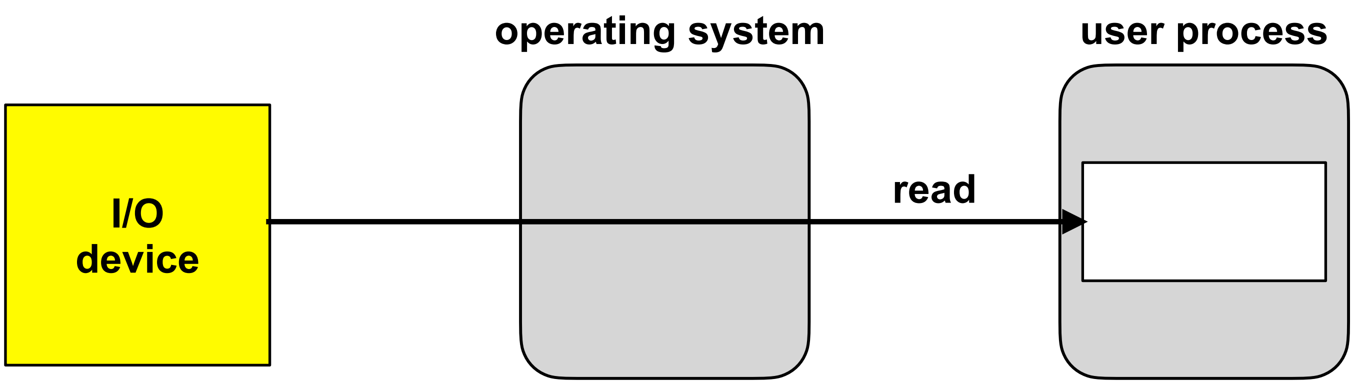
- **Alternative 1**: non-blocking input/output
  - Pass the `O_NDELAY` flag to the `open()` system call
  - *Polling* operation: the process has to call `read()` repeatedly until data arrives
  - Suboptimal solution that *wastes CPU time*

Unix: waiting for multiple devices (2)

- **Alternative 2**: blocking wait for *multiple file descriptors*
  - System call:
    ```
    int select (int nfds, fd_set *readfds, fd_set *writefds,
                          fd_set *errorfds, struct timeval *timeout);
    ```
  - `nfds` defines the *maximum file descriptor* which `select` should consider
  - …`fds` indicates the file descriptors to wait on:
    - `readfds` — wait on these until data is available
    - `writefds` — …until data can be written
    - `errorfds` — …until an error is signaled
  - `timeout` defines the time at which `select` unblocks if no other event occurred
  - Macros are provided to create the file descriptor sets
  - Result of `select`: the descriptor sets only contain those descriptors which resulted in the deblocking of the call

Buffering of I/O operations

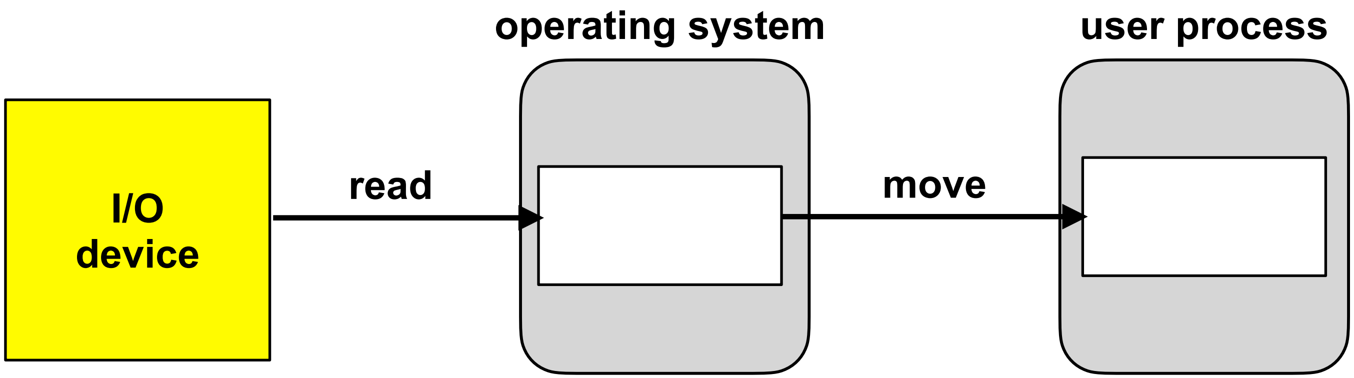
- **Problem** if an operating system does *not provide data buffers*:
  - Data which arrives before a corresponding `read` operation is executed (e.g. keyboard input) would get lost/discarded
  - If an output device is busy, `write` would either fail or block the process until the device is ready again
  - A process executing an I/O operation cannot be swapped



a) read operation without buffering

Single I/O buffers

- **Read**
  - The OS can accept data even if the reader process has not executed `read` yet
  - For block devices, a subsequent block can already be *prefetched*
  - The process can now be swapped, DMA writes to a buffer
- **Write**
  - Data is copied, the caller does not block. Data buffers in the user address space can immediately be reused



b) read operation with single buffering

Single I/O buffers

- **Read**

Performance estimation

A simple back-of-the envelope calculation gives an indication of the performance increase when repeatedly reading blockwise with subsequent processing:

ed

- **W**

er

T:	Duration of the read operation
C:	Compute time required for processing
M:	Duration of the copy process (system buffer→user process)
B:	Overall time required for reading and processing a block

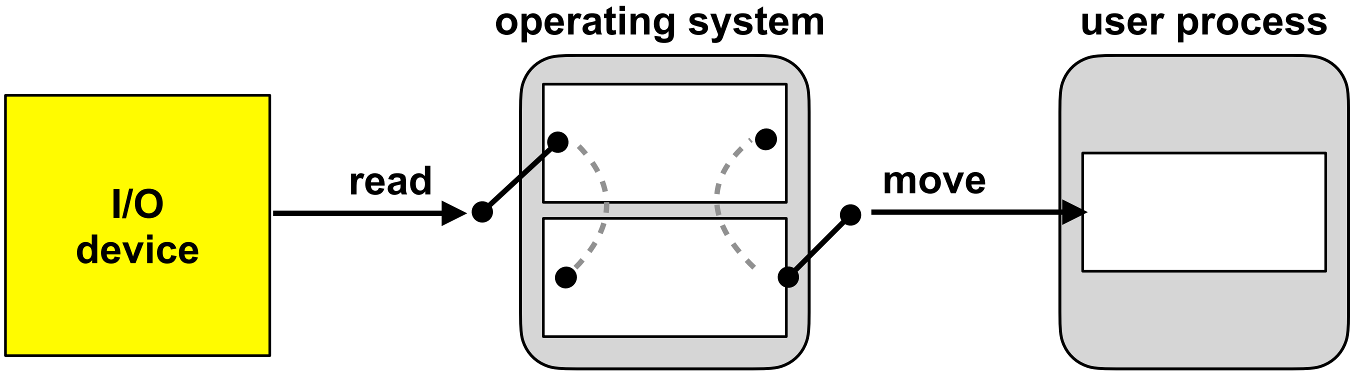
Without buffer: $B_0 = T + C$
With buffer: $B_E = \max(T, C) + M$

For $T \approx C$ und $M \approx 0$, $B_0 \approx 2 \cdot B_E$. Unfortunately, $M > 0$

b) read operation with single buffering

Double I/O buffering

- Read
  - While data is transferred from the I/O device to one of the buffers, the contents of the other buffer can be copied into the user address space
- Write
  - While data is transferred from one of the buffers to the I/O device, the contents of the other buffer can already be refilled with data from the process address space



c) read operation with double buffering

Double I/O buffering

- Read

Performance estimation

A double buffer enables to execute a read operation in parallel to a copy operation and processing

Without buffer: $B_0 = T + C$
With buffer: $B_E = \max(T, C) + M$
With double buffer: $B_E = \max(T, C + M)$

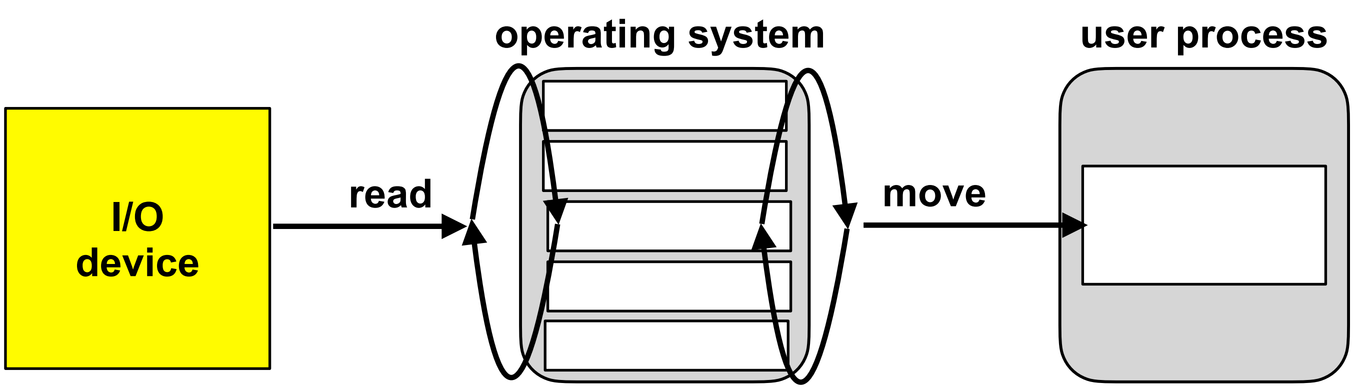
If C + M < T, the device could be utilized to 100%

- W

c) read operation with double buffering

I/O ring buffers

- **Read**
  - Multiple (many) data blocks can be buffered, even if the reading process does not call `read` fast enough
- **Write**
  - A writer process can execute multiple `write` calls without being blocked



d) read operation with ring buffer

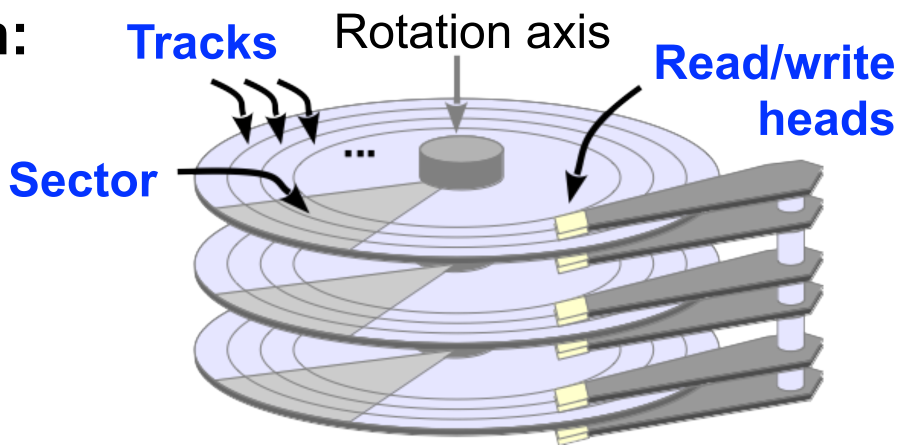
Discussion: I/O buffers

- I/O buffers *decouple* the I/O operations of user processes from the device driver
  - This enables to handle an increased rate of I/O requests for a **short duration**
  - In the **long run**, no amount of buffers can avoid a blocking of processes (or the loss of data)
- Buffers create *overhead*
  - Management of the buffer structure
  - Space in memory
  - Time required for copying
- In complex systems data can be buffered multiple times
  - Example: between layers of network protocols
  - Avoid if possible!

Device control example: disk

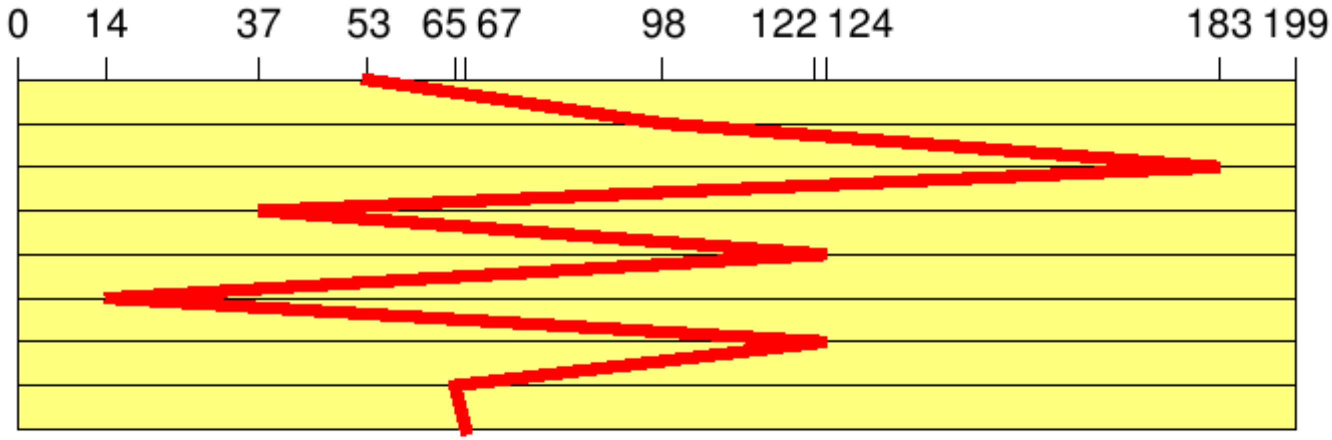
- Driver has to consider **mechanical properties**!
- Disk drivers usually queue multiple requests
  - The order of request execution can increase efficiency
  - The time required to process a request consists of:
    - Positioning time: depends on current position of the disk head arm
    - Rotational delay: time until the sector passes by the read/write head
    - Transfer time: time required to transfer the data

- **Optimization criterium:** positioning time



I/O scheduling: FIFO

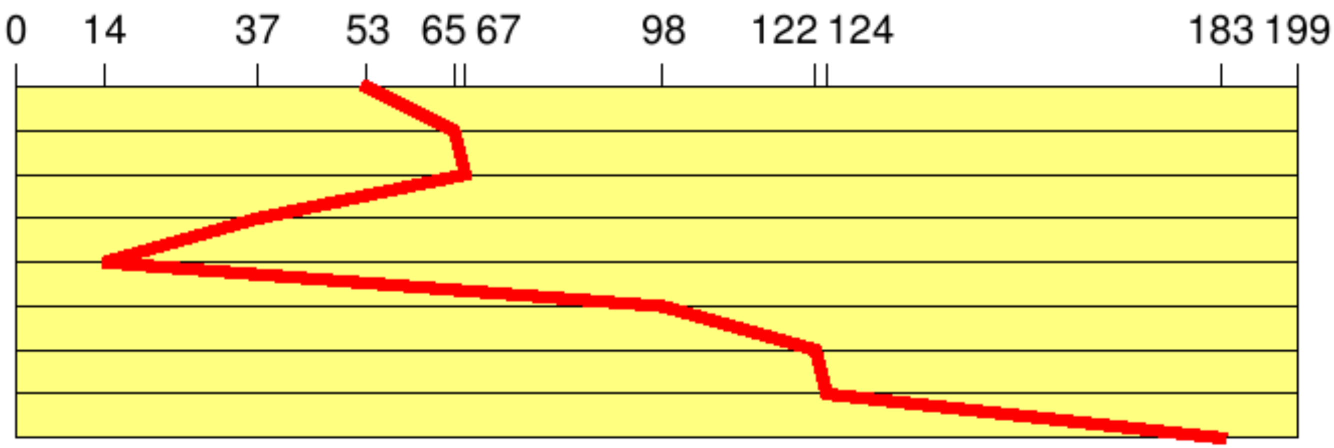
- Process requests in order of their arrival (**first in first out**)
  - Reference sequence (sequence of track numbers): 98, 183, 37, 122, 14, 124, 65, 67
  - Current track: 53



- Total number of **track changes: 640**
- Large movements of the disk arm: long average processing time!

I/O scheduling: SSTF

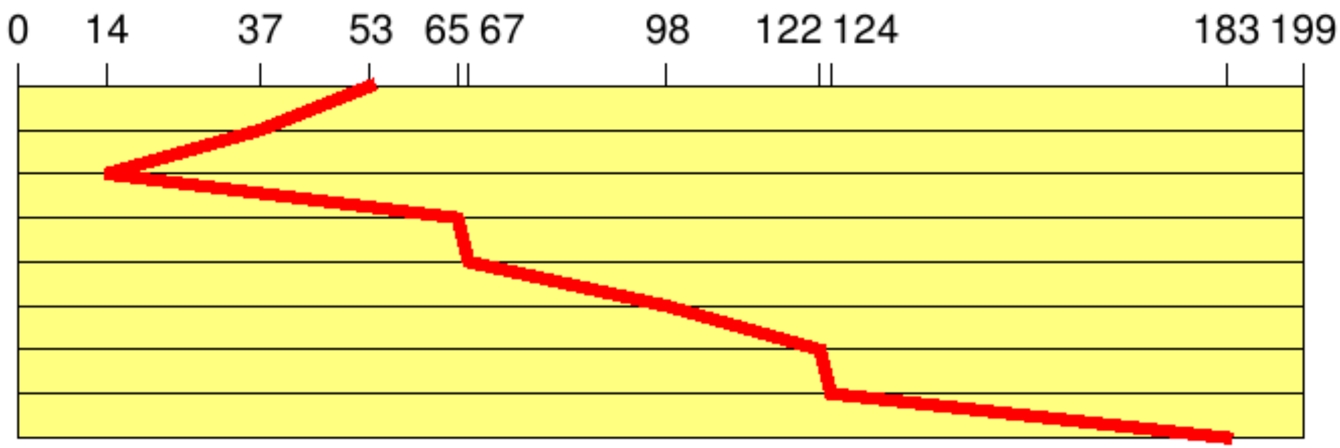
- The request with the shortest processing time is prioritized (**shortest seek time first**)
  - Same reference sequence
  - (Assumption: positioning time proportional to track distance)



- Total number of **track changes: 236**
- Similar to SJF scheduling, SSTF can also lead to *starvation*!
- still not optimal

I/O scheduling: Elevator

- Move the disk arm in one direction until no more requests are available (**elevator scheduling**)
  - Same reference sequence
    (assumption: head moves in direction 0)



- Total number of **track changes: 208**
- New requests executed without additional positioning time
- No starvation, but long waiting times are possible

Discussion: I/O scheduling today

- Disks are "*intelligent*" devices
  - Physical properties are hidden (logical blocks)
  - Disks have huge caches
  - *Solid State Disks* no longer contain mechanical parts
- I/O-scheduling slowly loses relevance
  - Success of a given strategy is more difficult to predict

- Nevertheless, I/O scheduling is still very important
- CPU speeds increase further, disk speeds do not
  - **Linux** currently implements two different variants of the **elevator algorithm** (+ FIFO for "disks" without positioning time):
    - DEADLINE: prioritizes read requests (shorter deadlines)
    - COMPLETE FAIR: all processes get an identical fraction of the I/O bandwidth

Conclusion

- I/O hardware comes in very many different variants
  - sometimes difficult to program
- The "art" of designing an OS consists of...
  - nevertheless defining uniform and simple interfaces
  - using the hardware efficiently
  - maximizing CPU and I/O device utilization

- The availability of a large number of device drivers is extremely important for the success of an operating system
  - Device drivers are by far the largest subsystem in Linux and Windows