

Operating Systems

Timo Hönig

Bochum Operating Systems and System Software (BOSS)

Ruhr University Bochum (RUB)

IX. Input/Output

June 14, 2023 (Summer Term 2023)



RUHR
UNIVERSITÄT
BOCHUM

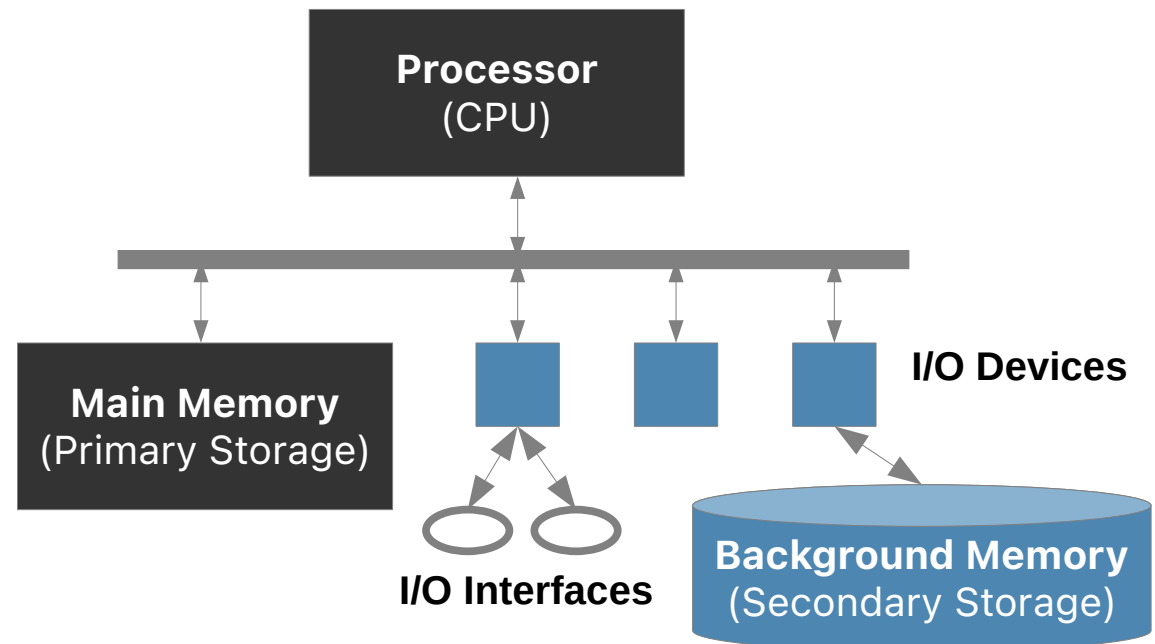
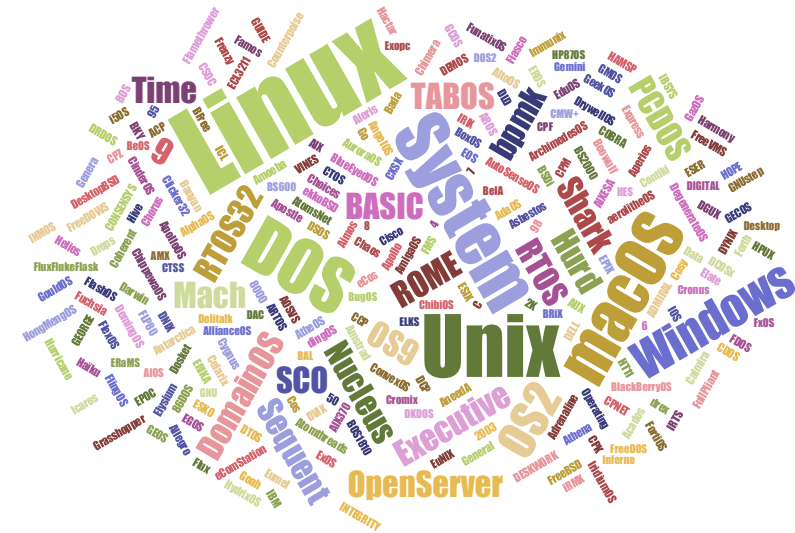
RUB

www.informatik.rub.de

Chair of Operating Systems and System Software

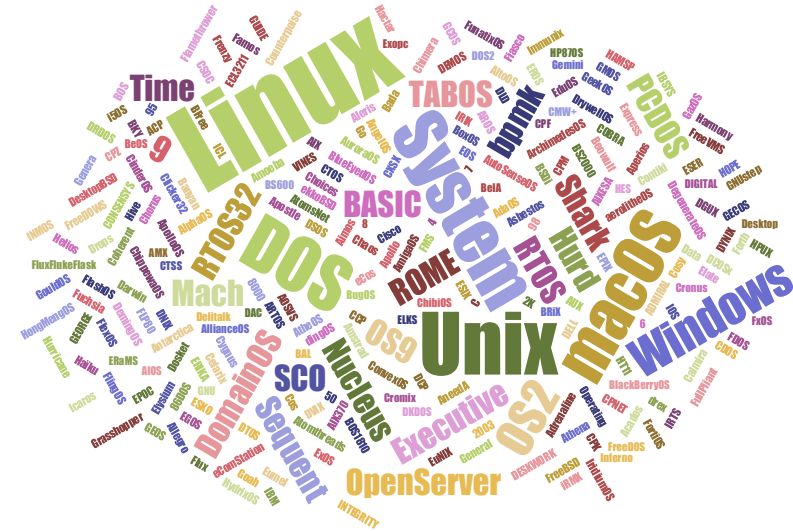
Recap

- file systems: operating system abstraction to organise persistent data on storage devices
- mapping of logical view (files, directories) to physical view (blocks)
- consider hardware properties for efficient data management
 - head positioning
 - wear leveling
- reliability by redundancy, better performance with copy-on-write



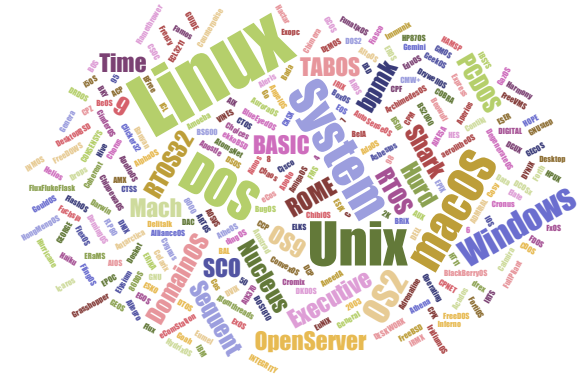
Agenda

- ▶ Recap
- ▶ Organizational Matters
- ▶ Input/Output Hardware
 - ▶ Interconnects
 - ▶ Device Classes
 - ▶ Interrupts, Direct Memory Access
- ▶ Device Programming
 - ▶ Address Space
 - ▶ Operating Modes
- ▶ Input/Output in Operating Systems
 - ▶ Input/Output Abstractions in UNIX
 - ▶ Buffered Input/Output
- ▶ Summary and Outlook



Organizational Matters

- [illegible]

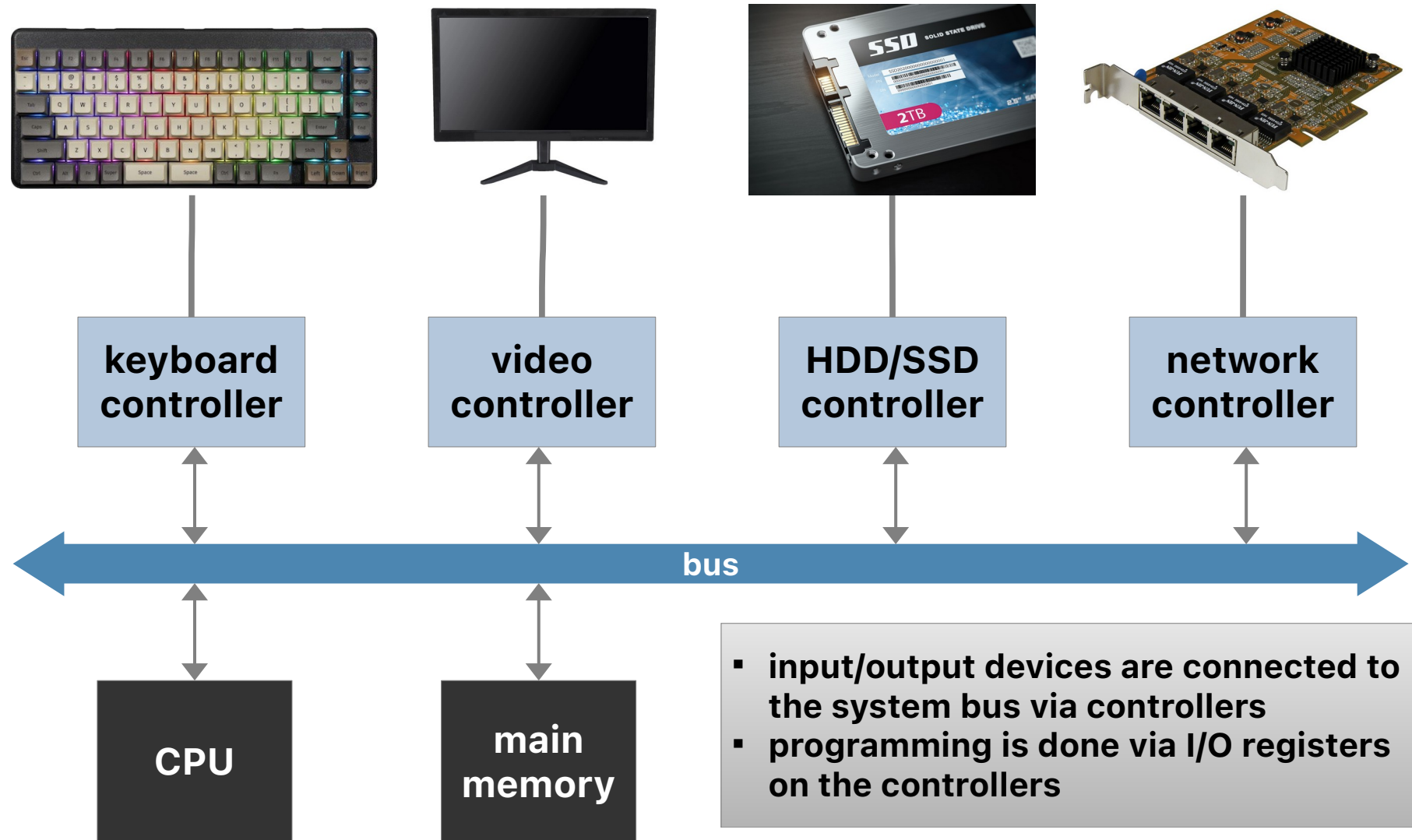


Agenda

- ▶ **Recap**
- ▶ **Organizational Matters**
- ▶ **Input/Output Hardware**
 - ▶ **Interconnects**
 - ▶ **Device Classes**
 - ▶ **Interrupts, Direct Memory Access**
- ▶ **Device Programming**
 - ▶ **Address Space**
 - ▶ **Operating Modes**
- ▶ **Input/Output in Operating Systems**
 - ▶ **Input/Output Abstractions in UNIX**
 - ▶ **Buffered Input/Output**
- ▶ **Summary and Outlook**

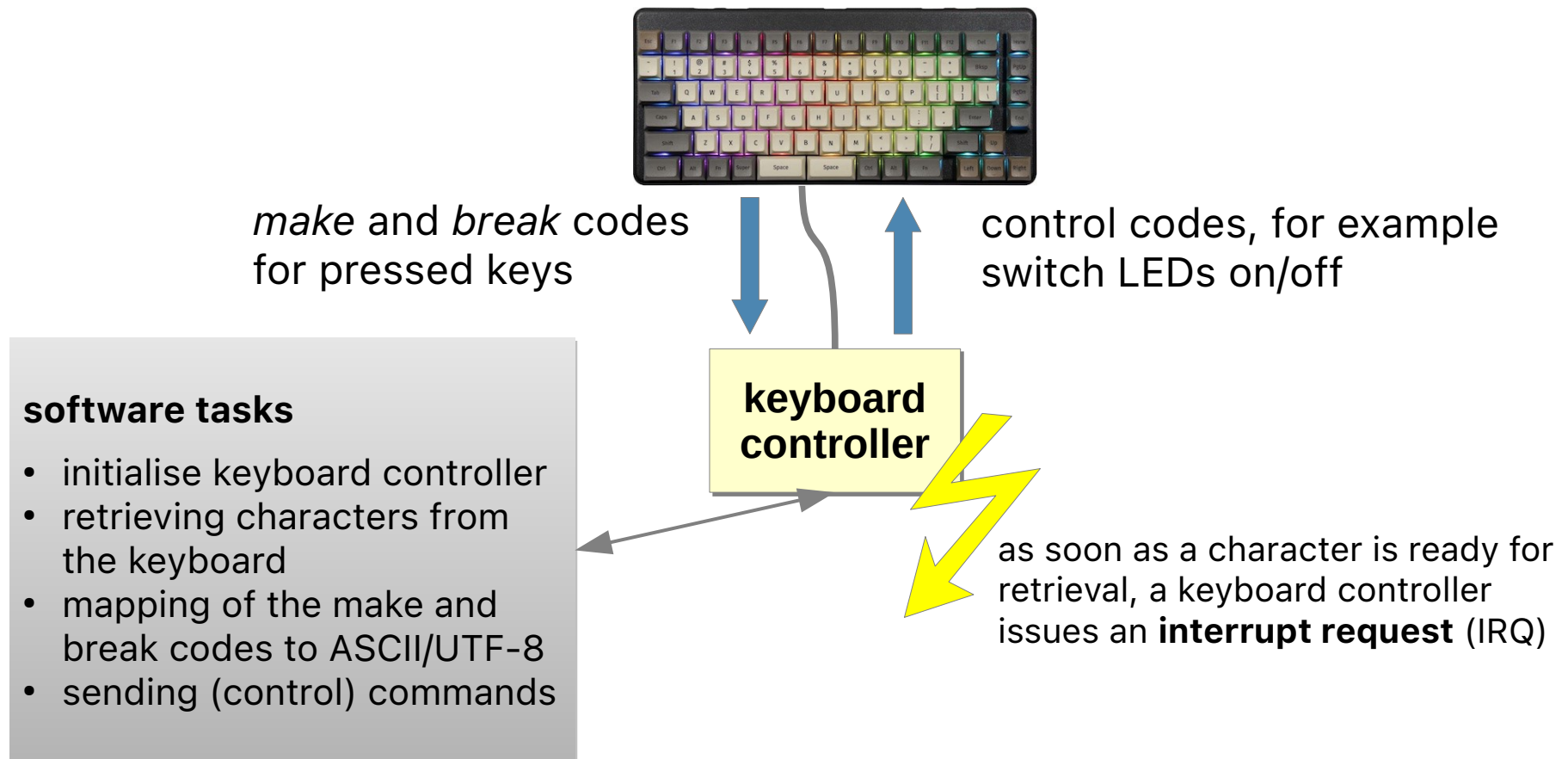


Interconnects of I/O Devices



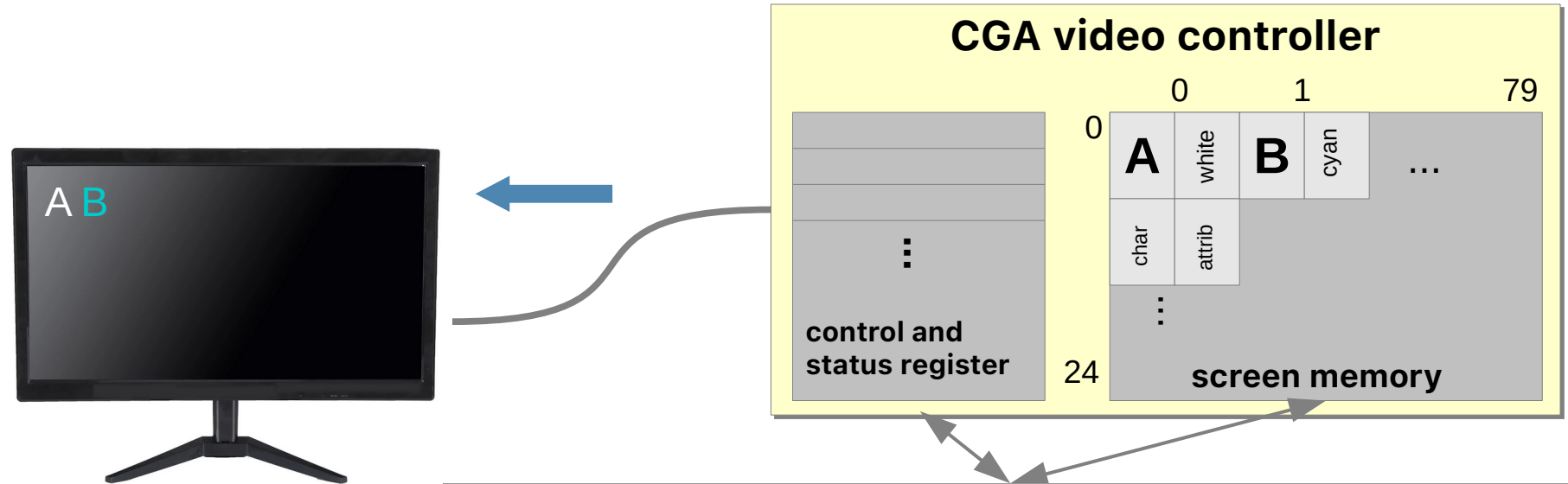
Example: PC Keyboard

- serial communication, character-based
 - keyboards are complex and have own (embedded) processors



Example: CGA Video Controller

- communication via video signal
 - converting the screen memory content into an image (80x25 chars)



software tasks

- initialise video controller
- fill screen memory with the desired character codes
- control the position of the cursor
- switch cursor (blinking) on and off

Example: IDE Controller (HDD/SSD)

- communication via AT commands
 - block-based random access to data blocks



AT commands

- calibrate drive
- read/write/verify block
- format track (HDD)
- head positioning (HDD)
- TRIM (SSD)
- diagnostic/health parameters



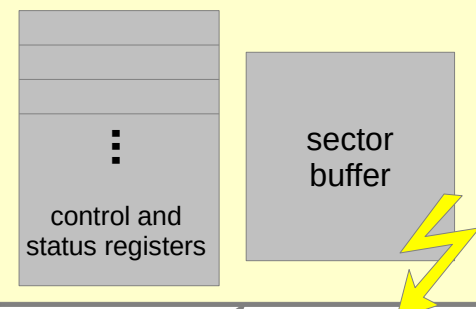
data blocks
(512 ... 4096 Bytes)

software tasks

- write AT commands to registers
- fill / empty sector buffer
- react on interrupt requests
- exception handling



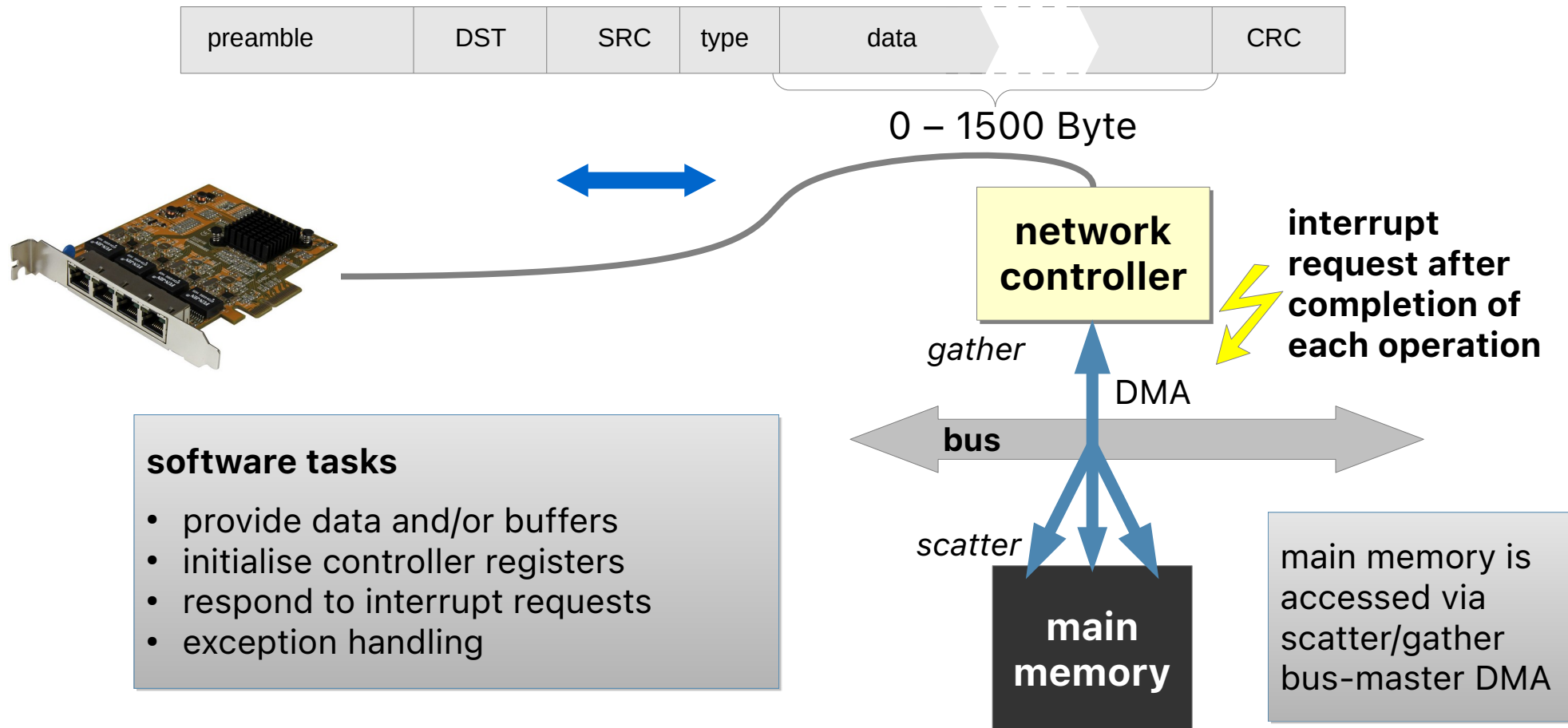
HDD/SSD controller



as soon as the sector buffer has been read or written full, the controller triggers an interrupt request

Example: Network Controller

- serial packet-based bus communication
 - packets have a variable size and contain addresses



Input/Output - Interrupts

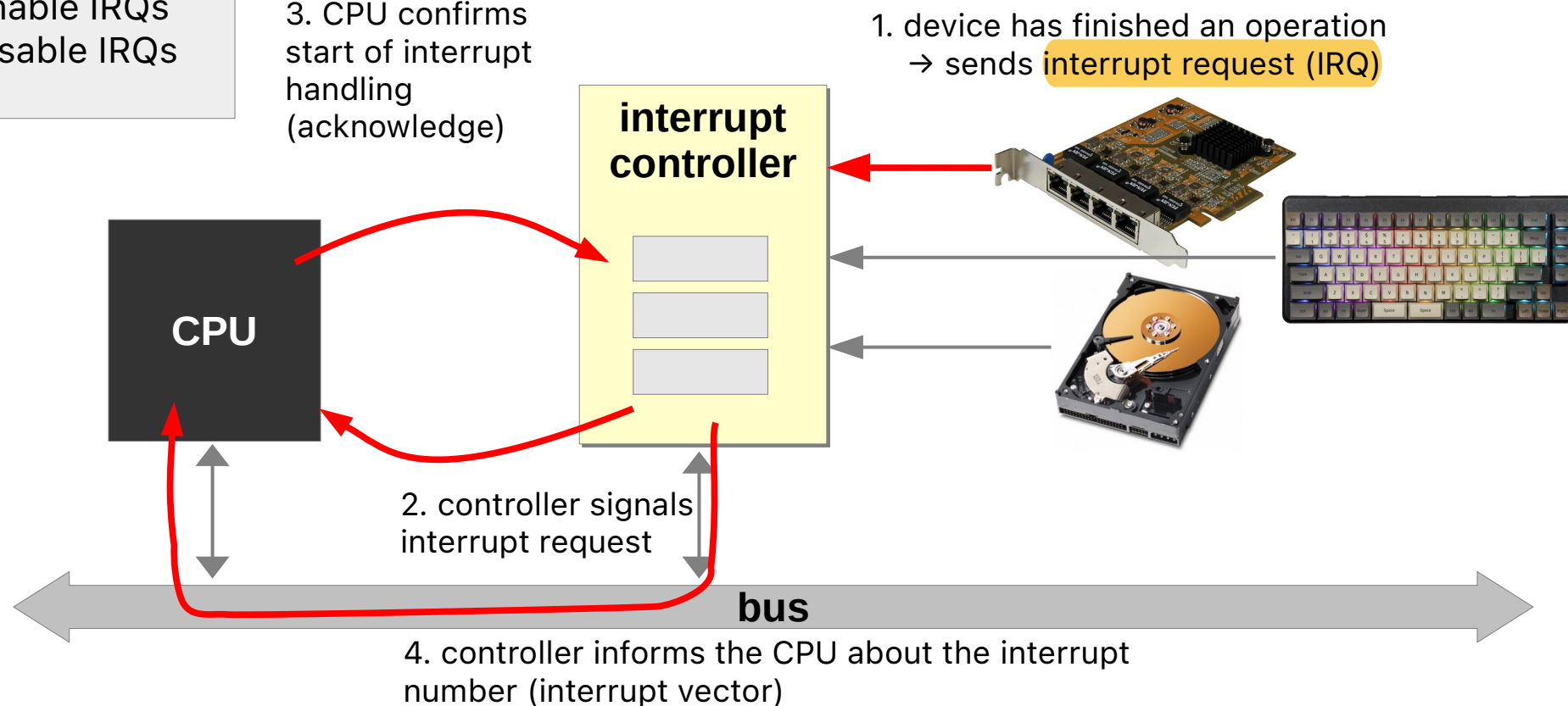
- hardware interrupt requests signal that the software must become active

software can
suppress **IRQ**
handling:

sti → enable IRQs

cli → disable IRQs

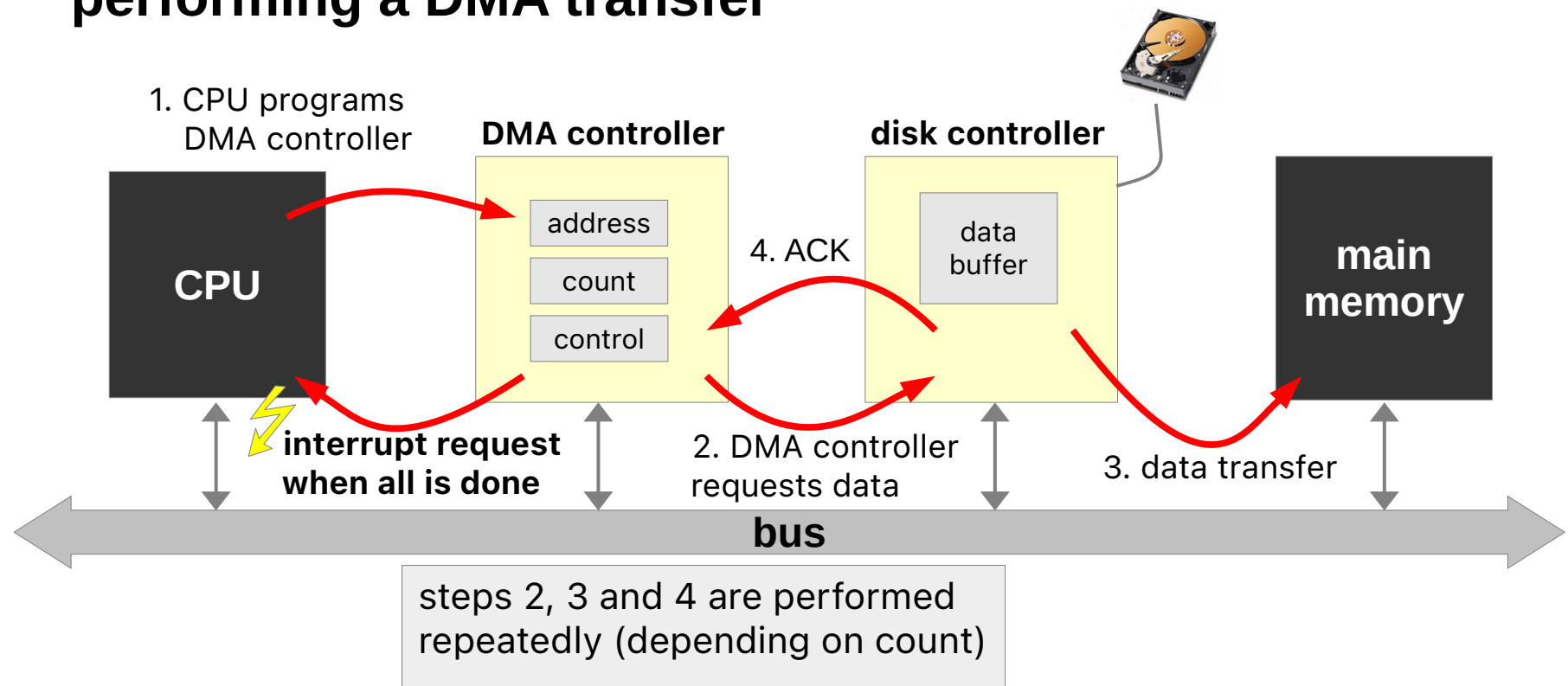
sequence of an interrupt handling on the hardware level



Input/Output - Direct Memory Access (DMA)

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

performing a DMA transfer



Device Classes

- **character-oriented devices**
 - keyboard, touch screen, printer, modem, mouse, etc.
 - mostly: purely sequential access, rarely random positioning
- **block-oriented devices**
 - hard disk (e.g., HDD), flash memory (e.g., SSD), optical disk drives
 - mostly: random block access (random access)
- other devices do not easily fit into the above scheme
 - graphics cards (especially 3D acceleration)
 - network cards (protocols, addressing, broadcast/multicast, message filtering, ...)
 - timer modules (one-time or periodic interrupts)

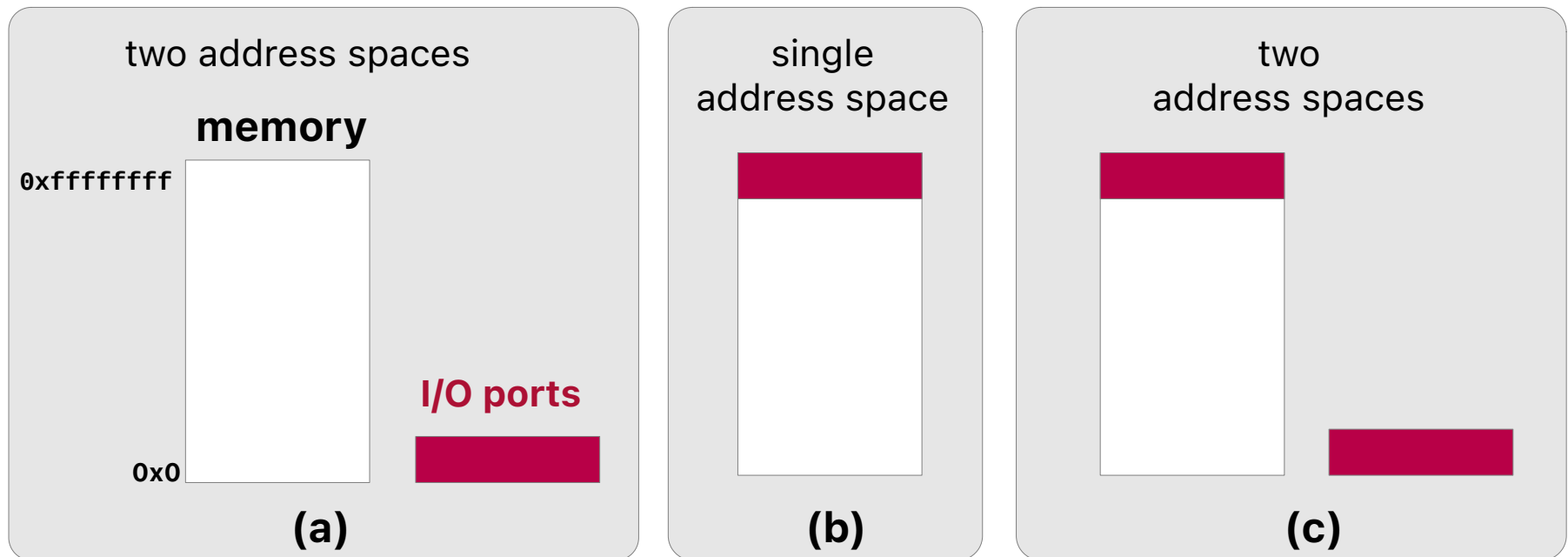
Agenda

- ▶ Recap
- ▶ Organizational Matters
- ▶ Input/Output Hardware
 - ▶ Interconnects
 - ▶ Device Classes
 - ▶ Interrupts, Direct Memory Access
- ▶ **Device Programming**
 - ▶ Address Space
 - ▶ Operating Modes
- ▶ Input/Output in Operating Systems
 - ▶ Input/Output Abstractions in UNIX
 - ▶ Buffered Input/Output
- ▶ Summary and Outlook



Address Space Models

- register access to I/O device controllers and controller memory is performed depending on the system architecture



(a) separate I/O address space

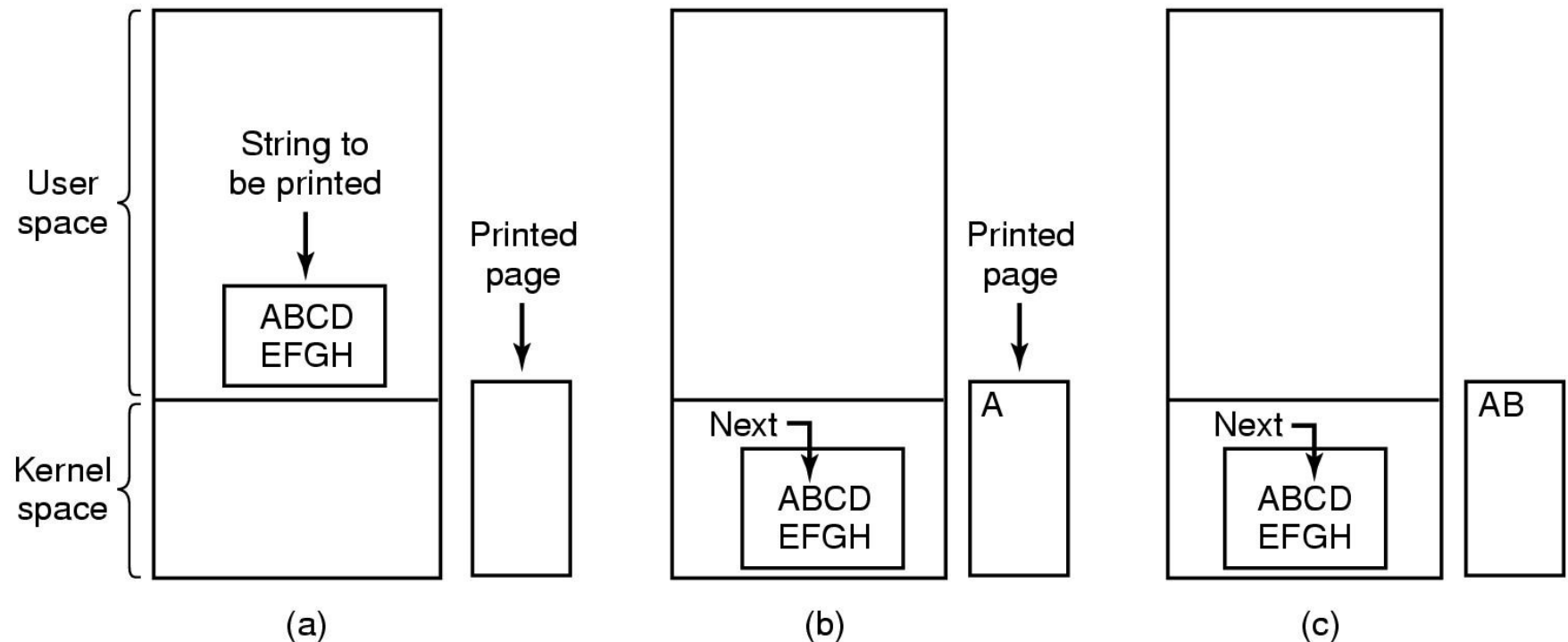
→ addressing using special machine instructions

(b) shared address space (memory-mapped I/O)

(c) hybrid architecture

Operating Modes of Device Drivers

- depending on the capabilities of the device, I/O is performed by means of:
 - **programmed input/output** (polling)
 - **interrupts**
 - **direct memory access**
- example: printing a line of text



Source: Tanenbaum, „Modern Operating Systems“

Programmed Input/Output

- **actively** waiting for an input/output device (polling)

```
/* copy character to kernel buffer p */  
copy_from_user (buffer, p, count);  
  
/* loop over all characters */  
for (i=0; i < count; i++) {  
  
    /* actively wait for printer */  
    while (*printer_status_reg != READY);  
  
    /* output one character */  
    *printer_data_reg = p[i];  
}  
  
return_to_user ();
```

pseudo code of an
operating system function
for printing text in polling
mode

Interrupt-Driven Input/Output

- the CPU can be assigned to another process during the waiting time

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

/* allow printer interrupts */
enable_interrupts ();

/* wait for printer */
while (*printer_status_reg != READY);

/* output one character */
*printer_data_reg = p[i++];

scheduler ();
return_to_user ();
```

code that initiates the I/O operation

```
if (count > 0) {
    *printer_data_reg = p[i];
    count--;
    i++;
}
else
    unblock_user ();
    acknowledge_interrupt ();
    return_from_interrupt ();
```

interrupt handling routine

Interrupt-Driven Input/Output – Discussion

- context saving
 - is partly done by the CPU itself, but only the necessary minimum (i.e., status register and return address)
 - all modified registers must be saved and restored at the end of the interrupt handling
- making interrupt handling as short as possible
 - during interrupt handling, further interrupts are usually suppressed
 - loss of interrupts is imminent
 - (advanced) interrupt controllers
 - only wake up the process that is waiting for I/O completion, if possible

Interrupt-Driven Input/Output – Discussion

- interrupts are *the* source of asynchronicity
 - root cause of race conditions in the operating system kernel
- interrupt synchronisation
 - simplest possibility: temporarily strictly disable interrupt handling by the CPU while critical sections are being executed
 - x86: `sti, cli`
 - again: risk of losing interrupts
 - in practice: multi-stage interrupt handling with minimising the time during which interrupts are strictly disabled
 - UNIX: top half, bottom half
 - Linux: tasklets
 - Windows: deferred procedures

DMA-Driven Input/Output

- the software (i.e., the OS) is no longer responsible for the data transfer between controller and main memory
 - the CPU load is reduced further

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

code that initiates the I/O operation

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

interrupt handling routine

DMA-Driven Input/Output - Discussion

■ caches

- today's processors operate with data caches; DMA bypasses the cache
- before setting up a DMA operation, the cache contents must be written back to the main memory and invalidated or the cache must not be used for the corresponding memory region

■ memory protection

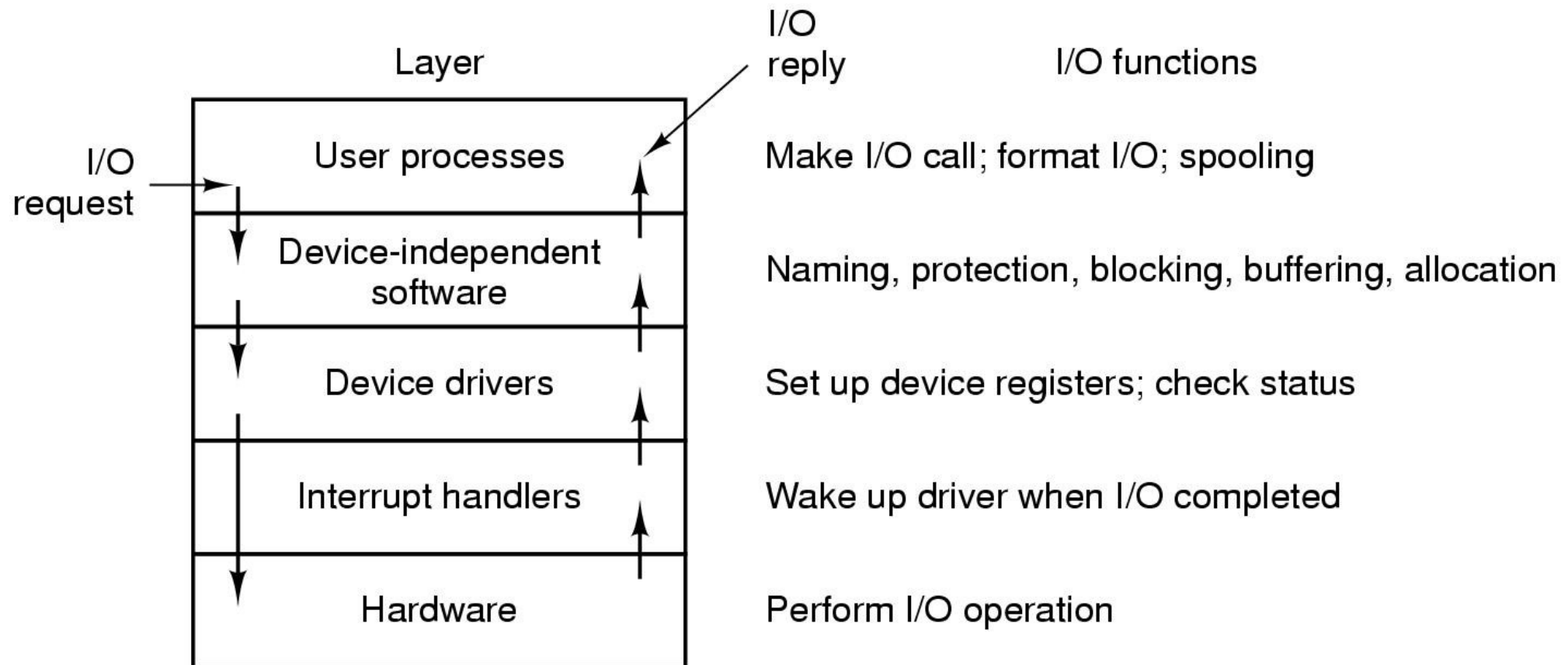
- today's processors use an MMU to isolate processes and protect the operating system; DMA bypasses memory protection
- errors in setting up DMA operations are critical
- application processes must never program DMA controllers directly → use system call(s) instead!

Agenda

- ▶ Recap
- ▶ Organizational Matters
- ▶ Input/Output Hardware
 - ▶ Interconnects
 - ▶ Device Classes
 - ▶ Interrupts, Direct Memory Access
- ▶ Device Programming
 - ▶ Address Space
 - ▶ Operating Modes
- ▶ **Input/Output in Operating Systems**
 - ▶ **Input/Output Abstractions in UNIX**
 - ▶ **Buffered Input/Output**
- ▶ Summary and Outlook



Input/Output Subsystem in Operating Systems



Source: Tanenbaum, „Modern Operating Systems“

Input/Output Device Abstractions in UNIX

- peripheral devices are represented as **special files**
 - devices are used like files using `read(2)` and `write(2)` operations
 - opening the special files creates a connection to the device, which is established by a driver
 - direct access from user space process to the driver
- block-oriented special files (block devices)
- character-oriented special files (character devices)







Input/Output Device Abstractions in UNIX

- unique **description** of I/O devices by a 3-tuple:
`<device type, major number, minor number>`
- **device type**: block device, character device
- **major number**: driver selection number
- **minor number**: selection of a device within a driver

Input/Output Device Abstractions in UNIX

- extract from the listing of the `/dev` directory

```
brw-rw---- user disk 3, 0 2023-06-12 14:14 /dev/hda
brw-rw---- user disk 3, 64 2023-06-12 14:14 /dev/hdb
brw-r----- root disk 8, 0 2023-06-12 14:13 /dev/sda
brw-r----- root disk 8, 1 2023-06-12 14:13 /dev/sda1
crw-rw---- root uucp 4, 64 2002-05-02 08:45 /dev/ttyS0
crw-rw---- root lp 6, 0 2023-06-12 14:13 /dev/lp0
crw-rw-rw- root root 1, 3 2006-05-02 08:45 /dev/null
lrwxrwxrwx root root 3 2023-06-12 14:14 /dev/cdrecorder -> hdb
lrwxrwxrwx root root 3 2023-06-12 14:14 /dev/cdrom -> hda
```

access rights owner major, minor number creation time stamp of the special file name of the special file representing the device

c: character device
 b: block device
 l: link

Input/Output System Calls in UNIX

- `int open(const char *devname, int flags)`
 - open a device - returns file descriptor as return value
- `off_t lseek(int fd, off_t offset, int whence)`
 - positions the read/write pointer - only for devices with random access
- `ssize_t read(int fd, void *buf, size_t count)`
 - read at max count bytes into buffer buf from descriptor fd
- `ssize_t write(int fd, const void *buf, size_t count)`
 - write count bytes from buffer buf to descriptor fd
- `int close(int fd)`
 - close a device - file descriptor fd cannot be used after that

Efficient Waiting for Multiple Devices in UNIX

- until now: blocking read or write calls
- **now**: asynchronous read or write calls
- what to do when reading from multiple sources?

- **solution 1**: non-blocking input/output
 - use `O_NDELAY` parameter with `open(2)`
 - polling operation: process has to call `read(2)` repeatedly until data is available
 - works, but is inadequate → CPU time is wasted

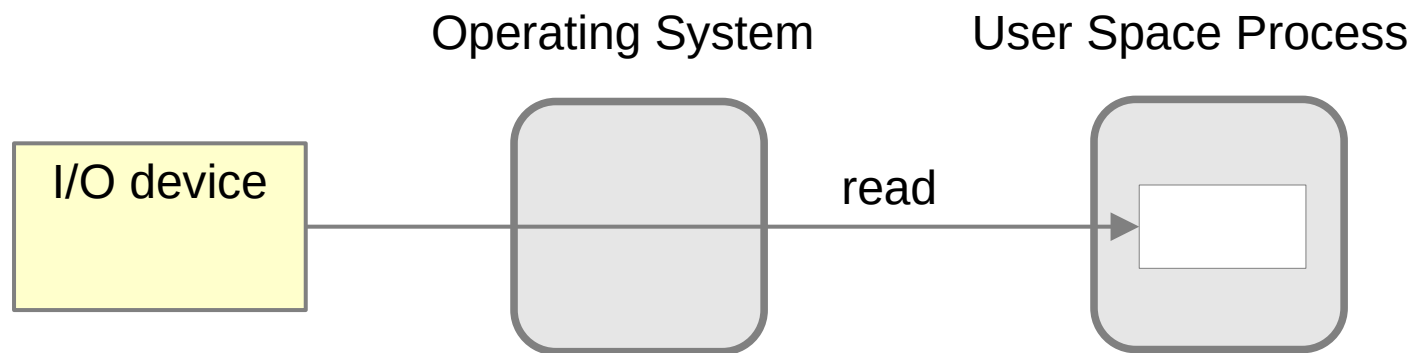
Efficient Waiting for Multiple Devices in UNIX

- **solution 2:** simultaneous blocking at multiple file descriptors
 - `select` system call:

```
int select (int nfd, fd_set *readfds, fd_set *writefds,  
            fd_set *errorfds, struct timeval *timeout);
```
 - `nfd` specifies up to which file descriptor `select` should operate
 - `read-`, `write-`, `errorfds` are file descriptors to wait for:
 - ➔ `readfds` — until there is data to read
 - ➔ `writefds` — until data can be written
 - ➔ `errorfds` — until an error occurs
 - `timeout` determines when the `select` call deblocks at the latest

Input/Output Operations without Buffering

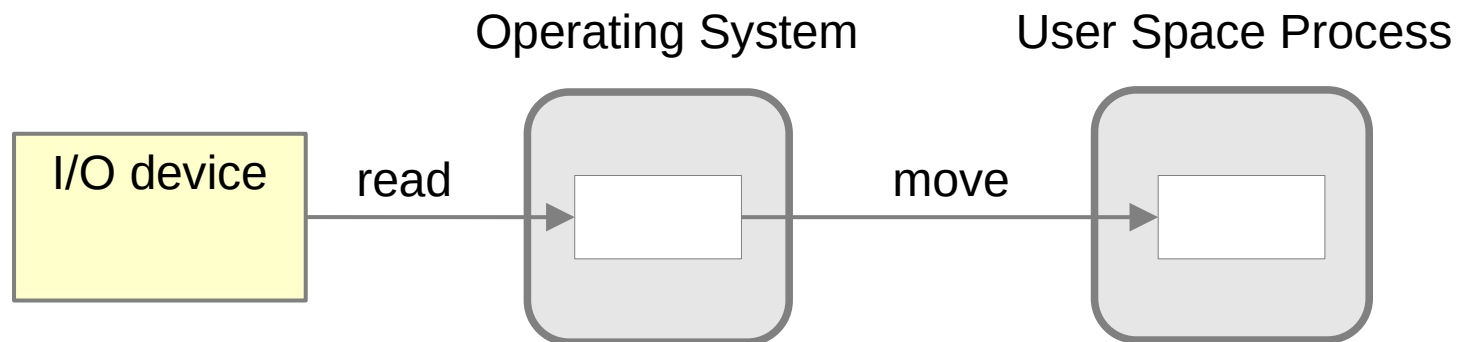
- problems without data buffer in the operating system:
 - data arriving before **read** is executed (e.g., from keyboard) would be lost
 - if an output device is busy, **write** would have to fail (or block the process) until the device is ready again
 - a process that performs an I/O operation cannot be swapped out



(a) read operation with no buffer

Buffered Input/Output - Single I/O Buffering

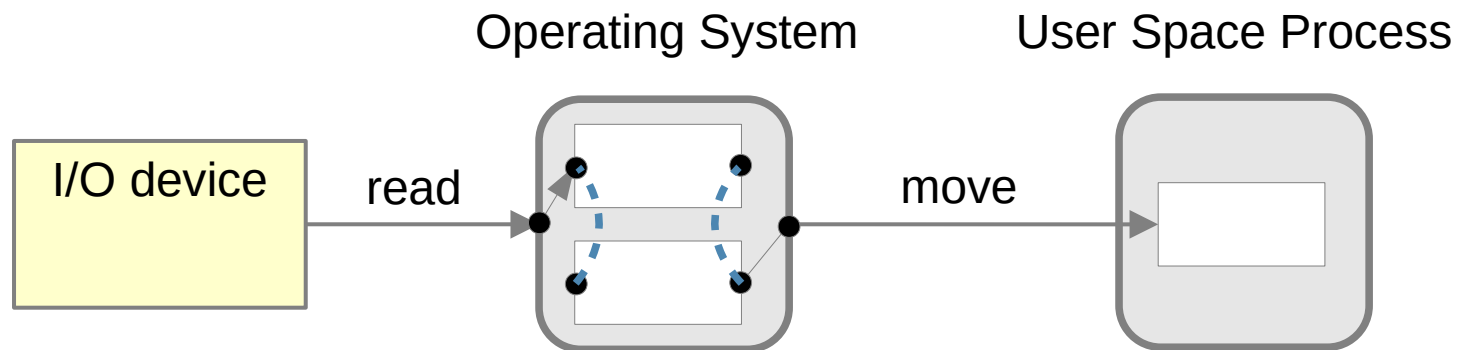
- read
 - system receives data even if the reading process has not yet called read
 - for block devices, the next block can be read ahead while the previous one is being processed.
 - process can be swapped-out, DMA takes place in buffer
- write
 - data is copied, caller is not blocked, data buffer in writer's address space can be reused immediately



(b) read operation with single buffer

Buffered Input/Output - Double I/O Buffering

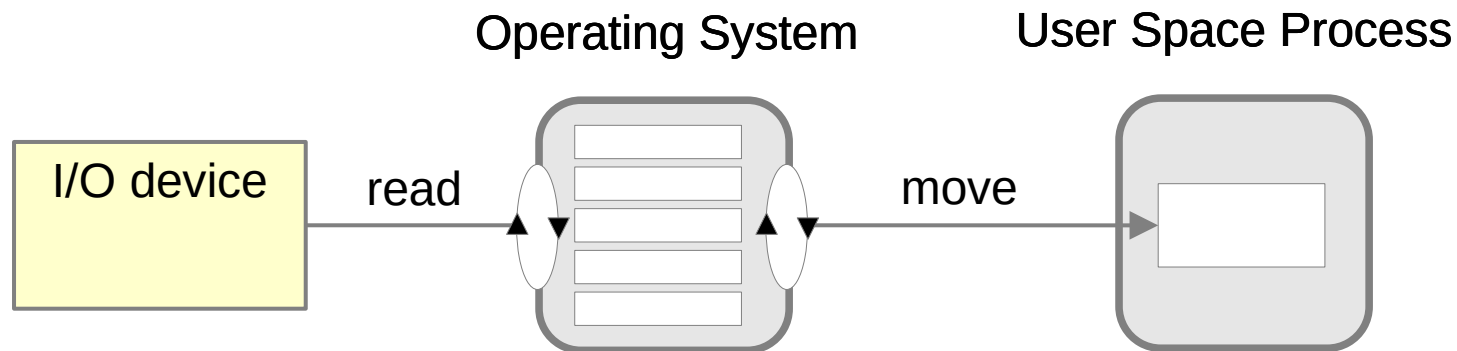
- read
 - while data is being transferred from the I/O device to one buffer, the other buffer's content can be copied to the reader process address space
- write
 - while data is being transferred from one buffer to the I/O device, the other buffer can already be filled with new data from the writer process address space



(c) read operation with double buffer

Buffered Input/Output – Circular I/O Buffering

- read
 - data can be buffered even if the reader process does not make read calls fast enough
- write
 - writer process can make multiple write calls without being blocked



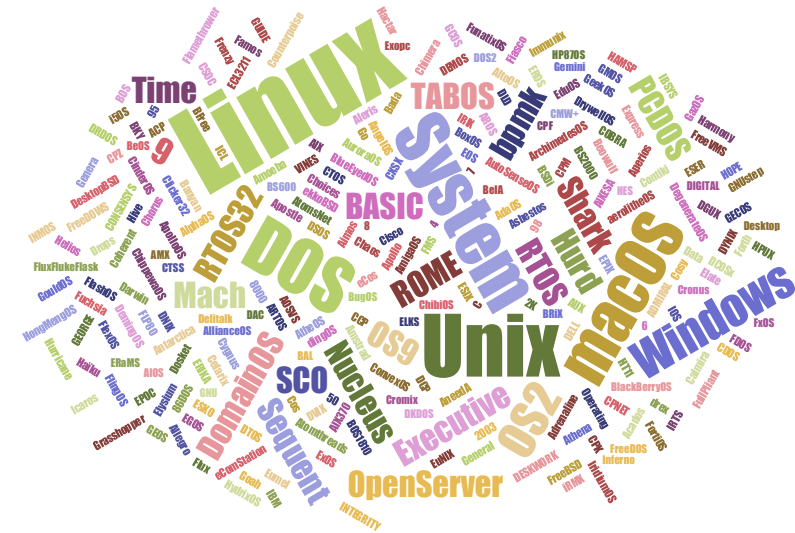
(d) read operation with circular buffer

Buffered Input/Output - Discussion

- buffers decouple I/O operations of processes (user space) from device drivers in the operating system (kernel space)
 - short term: an increased arrival rate of I/O jobs can be managed
 - long term: even with as many buffers as possible, a blocking of processes (or loss of data) cannot be ruled out
- buffers have their price (i.e., resource demand)
 - buffer structure management
 - storage space
 - time overhead for copying data
- in complex systems, buffering spreads across several layers
 - example: layers of network protocols

Agenda

- ▶ **Recap**
- ▶ Organizational Matters
- ▶ Input/Output Hardware
 - ▶ Interconnects
 - ▶ Device Classes
 - ▶ Interrupts, Direct Memory Access
- ▶ Device Programming
 - ▶ Address Space
 - ▶ Operating Modes
- ▶ Input/Output in Operating Systems
 - ▶ Input/Output Abstractions in UNIX
 - ▶ Buffered Input/Output
- ▶ **Summary and Outlook**



▶▶ Summary and Outlook

■ summary

- input/output hardware devices are very different and often difficult to handle
- the art of designing operating systems for efficient input/output is:
 - to define uniform and simple interfaces
 - to handle the heterogeneous hardware devices efficiently
 - to maximize CPU **and** I/O device utilisation
- device driver diversity is extremely important for wide adaptation of general purpose operating systems

■ outlook: scheduling

- methods and strategies to assign CPU time to processes and threads
- basic and advanced strategies for CPU scheduling

References and Acknowledgments

Lecture

- ▶ Systemnahe Programmierung in C (SPiC), Betriebssysteme (Jürgen Kleinöder, Wolfgang Schröder-Preikschat)
- ▶ Betriebssysteme und Rechnernetze (Olaf Spinczyk, Embedded Software Systems Group, Universität Osnabrück)

Teaching Books and Reference Book

- [1] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*, John Wiley & Sons, 2018.
- [2] Andrew Tanenbaum, Herbert Bos: *Modern Operating Systems*, Pearson, 2015.
- [3] Wolfgang Schröder-Preikschat: *Grundlage von Betriebssystemen – Sachwortverzeichnis*, 2023.
<https://www4.cs.fau.de/~wosch/glossar.pdf>