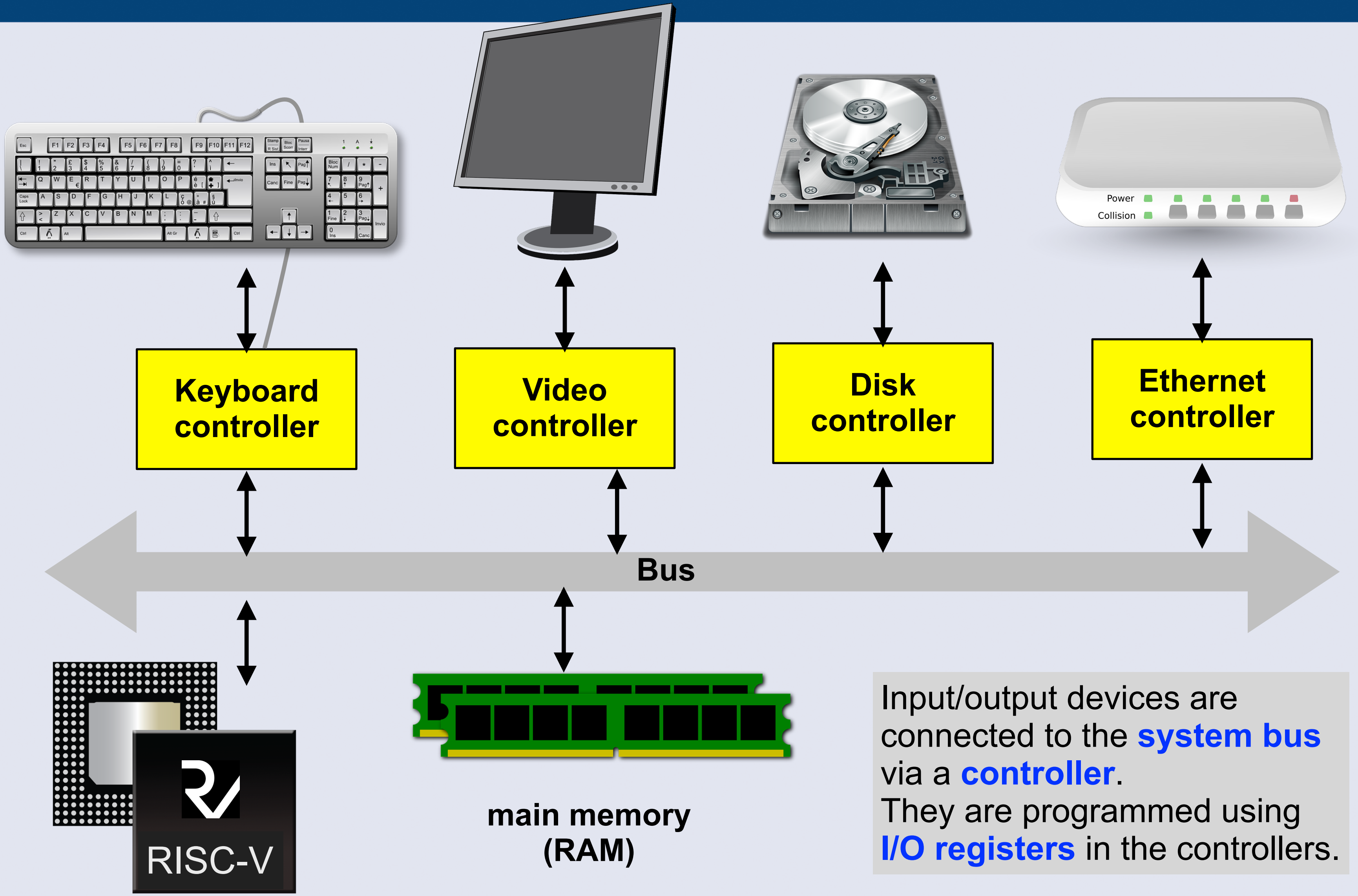
Universität Bamberg

Operating Systems Engineering

Lecture 7: Device drivers

Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insb. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

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

Universität Bamberg

- 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

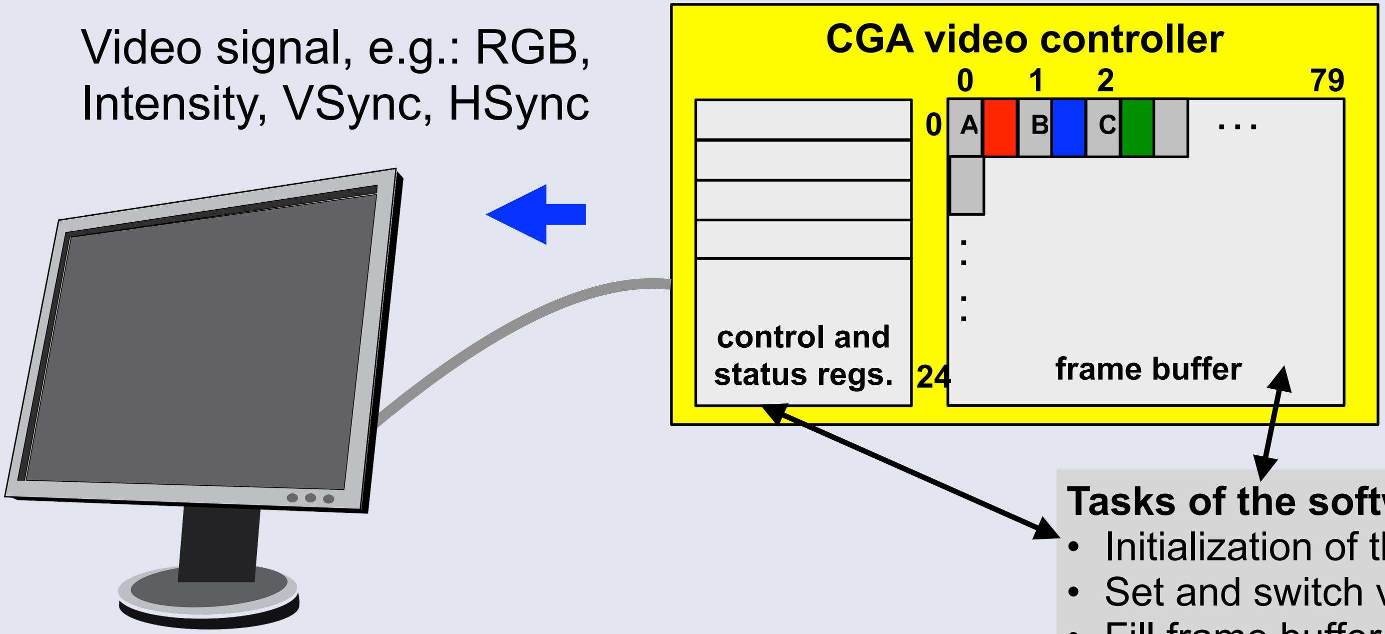
keyboard controller

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: 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)

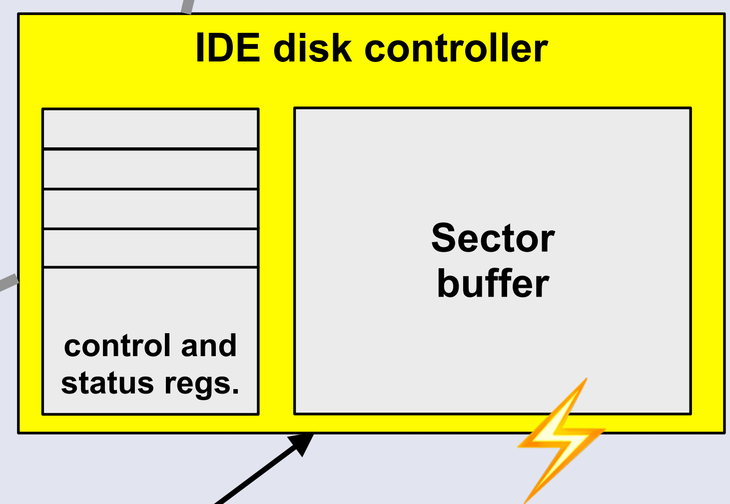


Tasks of the software:
- Initialization of the controller
- Set and switch video modes
- Fill frame buffer with content
- Control the cursor position
- Enable/disable the cursor

Example: IDE hard disk

- 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

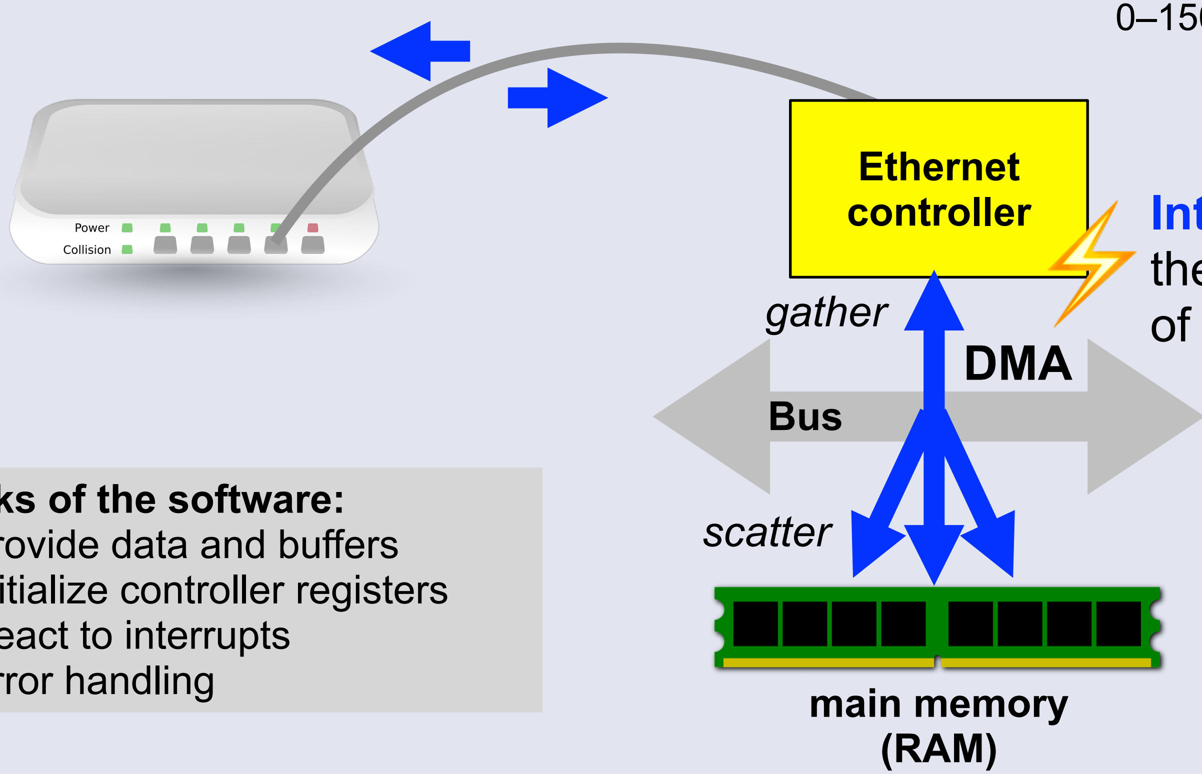
Example: Ethernet controller

Universität Bamberg

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

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

$\underbrace{\text{data ………………}}_{\text{0–1500 bytes}}$



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

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

Classes of devices

Universität Bamberg

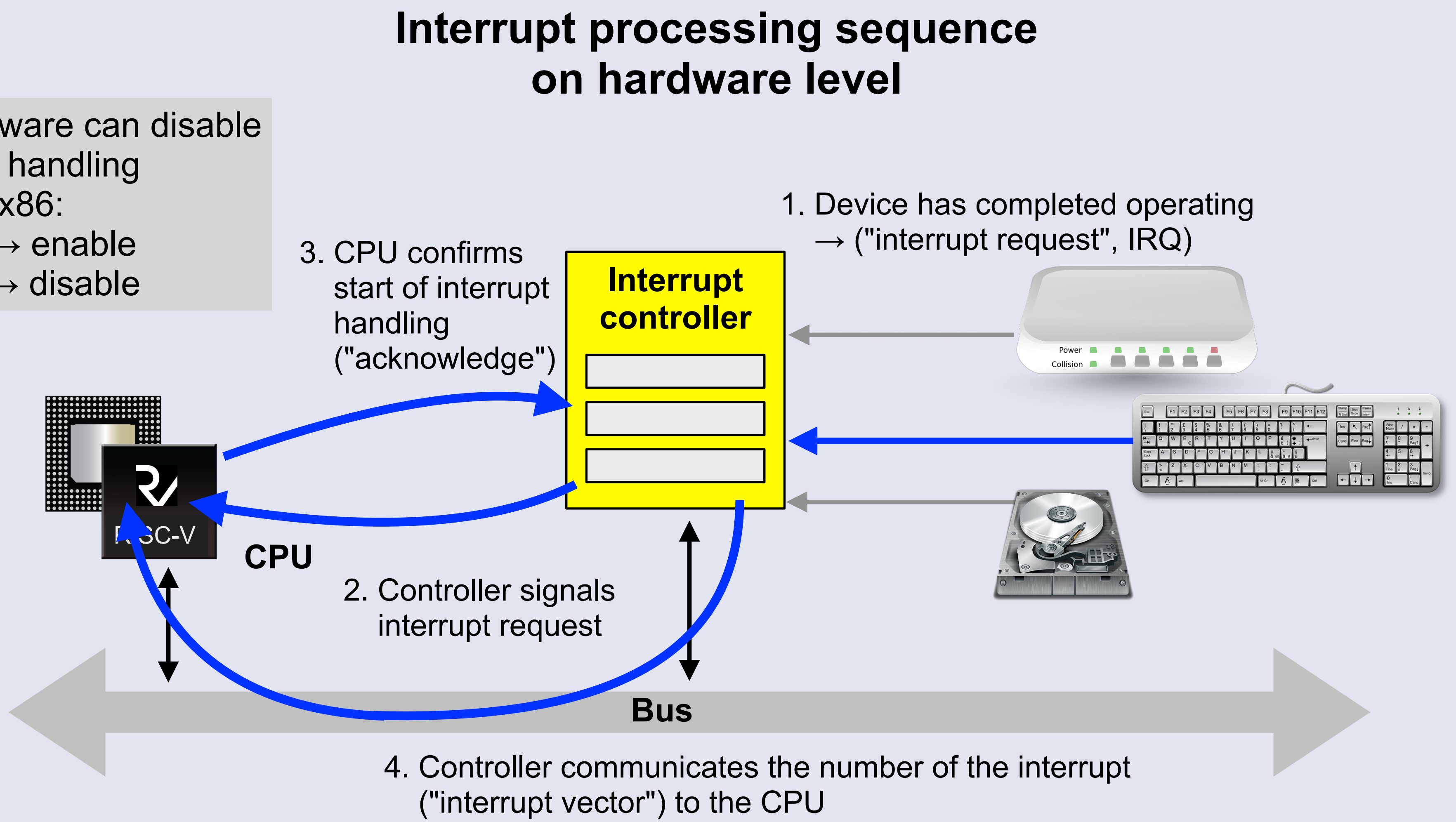
- **Character** devices
  - Keyboard, printer, modem, mouse, ...
  - Usually only sequential access, rarely random access
- **Block** devices
  - Hard disk, CD-ROM, DVD, tape drives, ...
  - Usually blockwise random access
- Other devices don’t fit this scheme easily, such as
  - (GP)GPUs (especially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

Interrupt handling

Universität Bamberg

- …signal the software to request handling a device

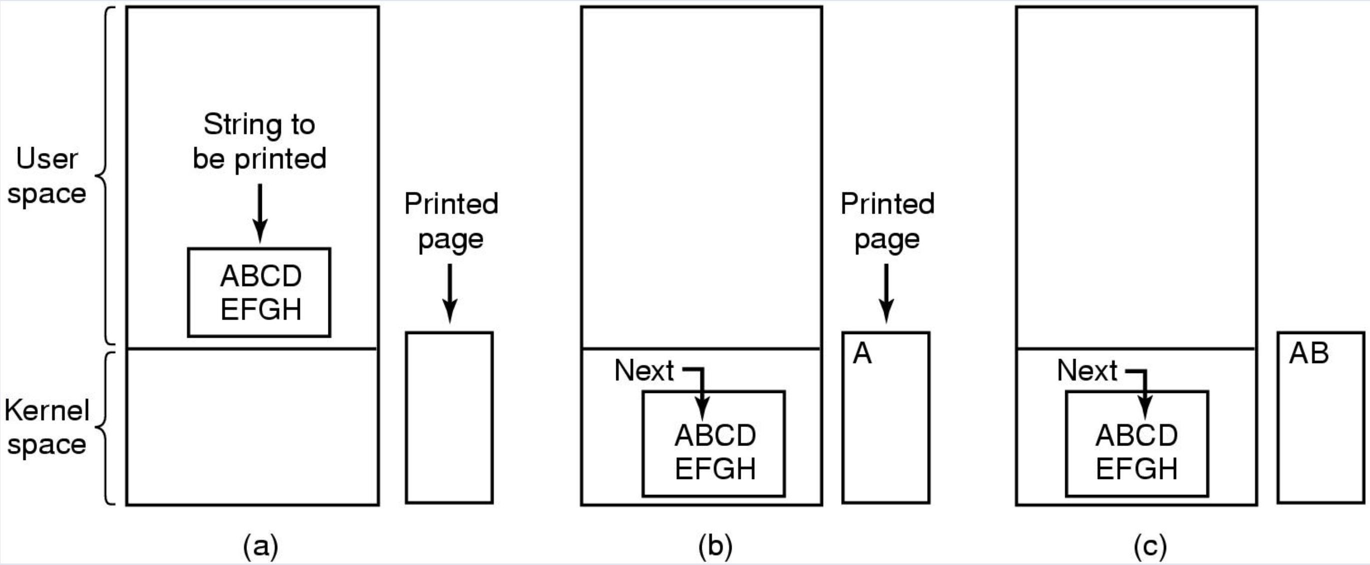
Software can disable IRQ handling
For x86:
sti → enable
cli → disable



Device drivers

Universität Bamberg

- 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")

Universität Bamberg

- ... 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

Universität Bamberg

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

```
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

```
if (count > 0) {

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

} else {

  unblock_user ();

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

Interrupt handler

DMA-driven I/O

Universität Bamberg

- ... 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 be 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
      - RISC-V: `mstatus` register (MIE/SIE bits), `mie` register
      - 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

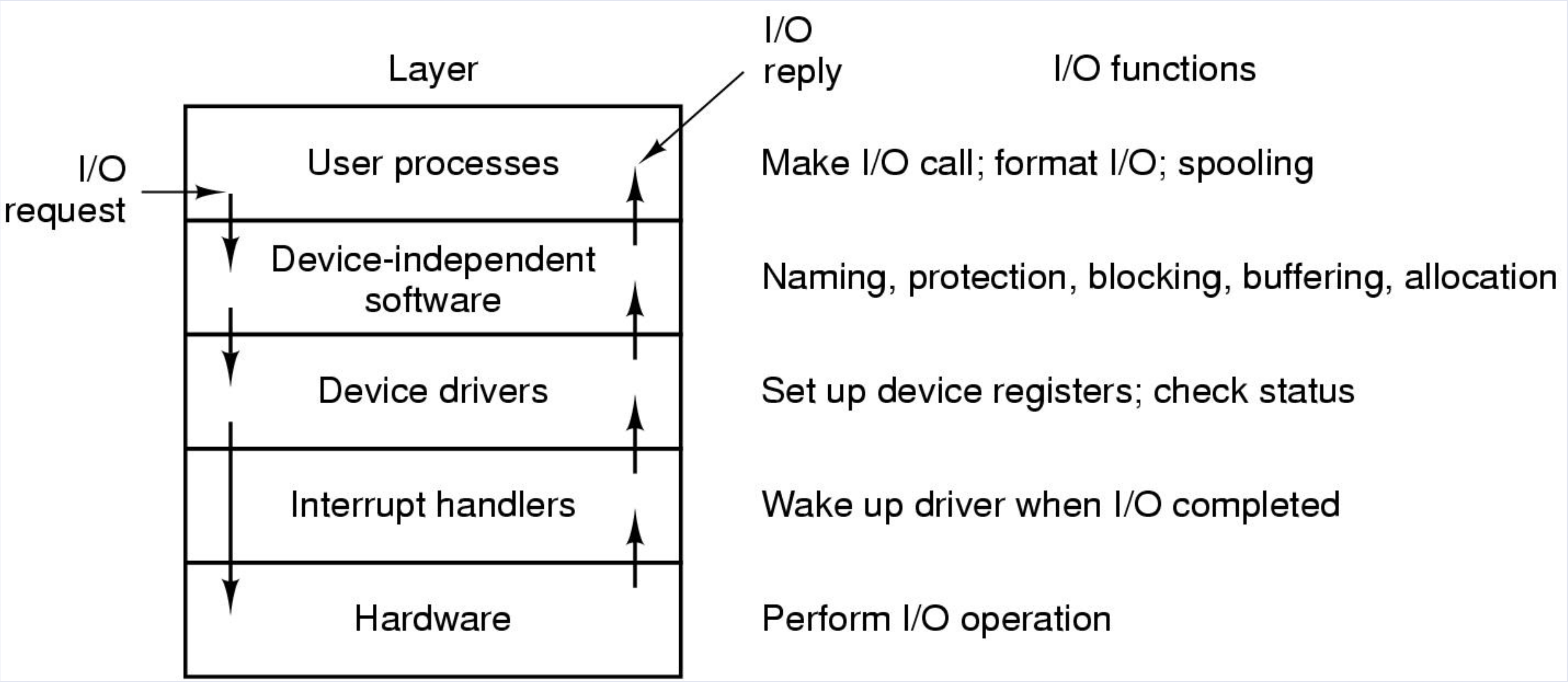
- **Caches**
  - Modern processors use data caches
    DMA bypasses the cache!
  - Before a DMA transfer is configured, cache contents must be written back to main memory and the cache invalidated
    - Some processors support non-cacheable address ranges for I/O operations
- **Memory protection**
  - 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 operating system

Universität Bamberg

- Create **device abstractions**
  - 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**
  - Enable adding and removing devices at runtime

Layers of the I/O system



Source: Tanenbaum, "Modern Operating Systems"

Device access API in Unix

Universität Bamberg

- Unix philosophy: ***everything is a file***

- 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

- 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

Universität Bamberg

- 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 file (and device) 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`

Device specific functions

Universität Bamberg

- 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, ...);
```

Example: speed of Ethernet or UART interface

- 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 device driver handling

Universität Bamberg

- How does the kernel map access to a device via open/read/… to the specific device driver?
  - The **device switch table** contains **function pointers** to the specific functions of a device driver for each device [1]
  - Early Unix: **static device configuration**
    - kernel recompilation required to add/remove drivers
  - Today: **kernel modules**
    - drivers loadable and installable at runtime
    - possible **security problem** – modules run in kernel address space and with kernel privileges! [2]
      - OpenBSD removed loadable kernel module support in 2015 [3]

Device switch table

conf.h

```
/* Declaration of block device switch. Each entry (row) is the only link between the main
 * unix code and the driver. The initialization of the device switches is in the file conf.c. */
struct bdevsw
{
        int     (*d_open)();
        int     (*d_close)();
        int     (*d_strategy)();
        int     (*d_dump)();
        int     (*d_psize)();
        int     d_flags;
};
struct  bdevsw bdevsw[];

/* Character device switch */
struct cdevsw
{
        int     (*d_open)();
        int     (*d_close)();
        int     (*d_read)();
        int     (*d_write)();
        int     (*d_ioctl)();
        int     (*d_stop)();
        int     (*d_reset)();
        struct tty *d_ttys;
        int     (*d_select)();
        int     (*d_mmap)();
};
struct  cdevsw cdevsw[];
```

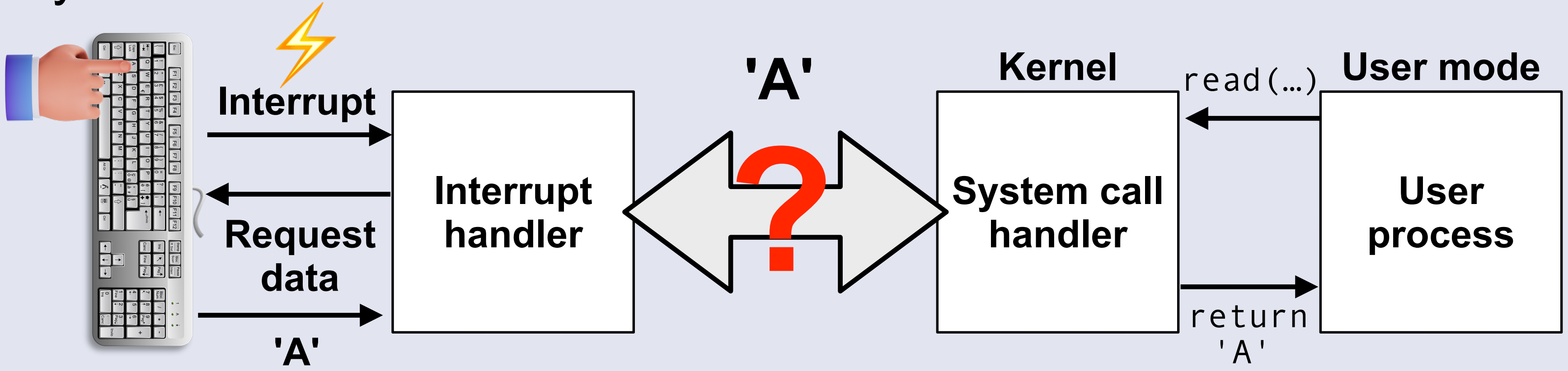
conf.c

```
struct bdevsw   bdevsw[] = {
{ hpopen, nulldev, hpstrategy, hpdump, /*0*/
  hpsize,       0 },
{ htopen, htclose, htstrategy, htdump, /*1*/
       0,  B_TAPE },
{ upopen, nulldev, upstrategy, updump, /*2*/
  upsize,       0 },
…
}
```

Major device ID:
index in `devsw` array!

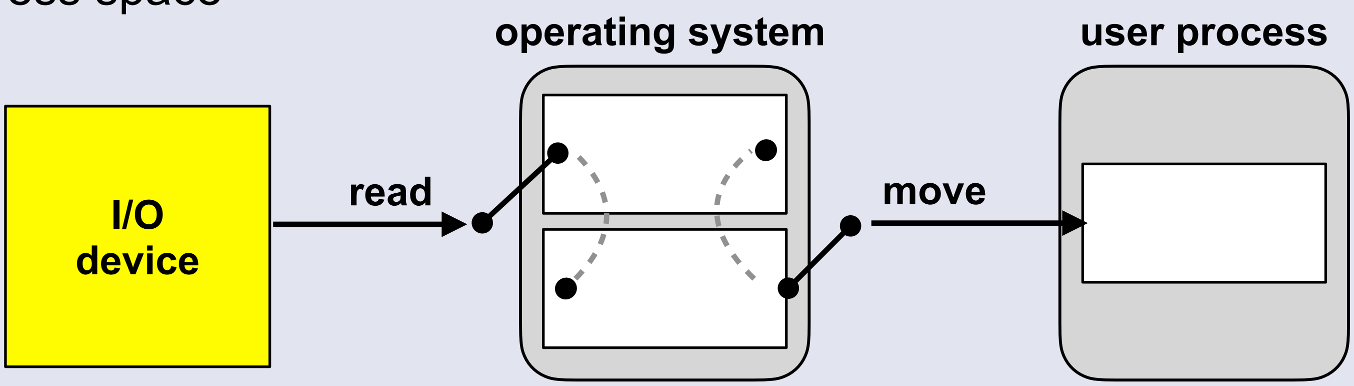
Kernel–interrupt handler communication

- **Problem:**
  - Device access functions (read, write,…) run **synchronously** as a system call, i.e. they cannot wait for the termination of I/O operations
    - Otherwise, the whole system would hang due to polling
  - The interrupt handlers run **asynchronously** whenever a device initiates a request
- How can we **exchange data** between interrupt handler and system call handler?



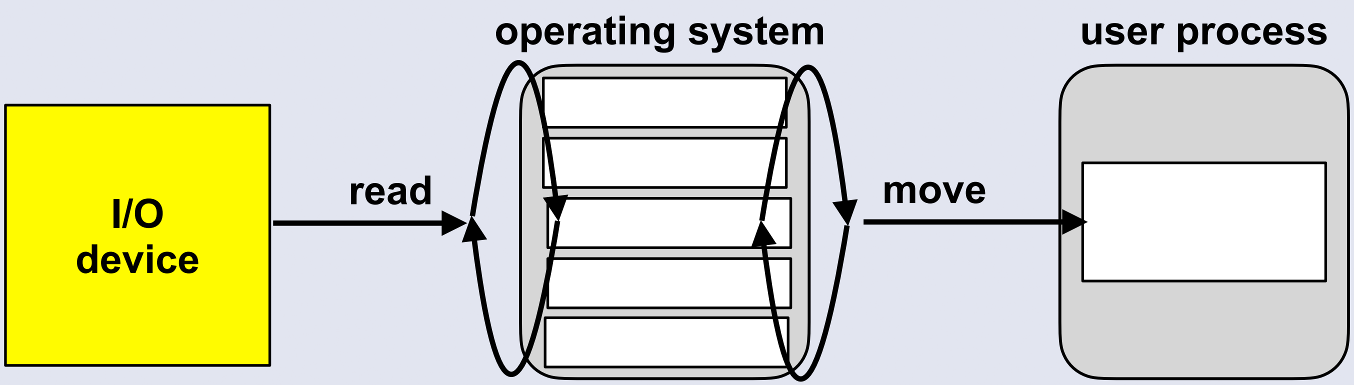
Buffering

- **Problem:**
  - Interrupt handler and syscalls run at different times and speeds
  - **Buffering** provides a way to exchange data without one part of the system having to wait for another
- **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



Ring buffers

- Idea: **decouple** kernel and interrupt handler
  - **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
- Ring buffers are a **circular data structure** implemented as an **array**
  - If read/write reaches the end, start from the beginning



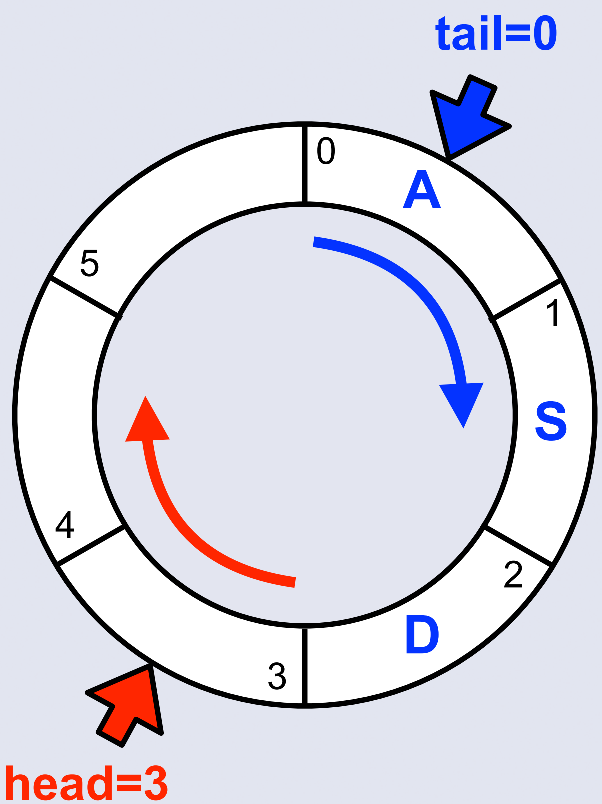
Ring buffer implementation

Universität Bamberg

```
#define BUFFER_SIZE 6
char ringbuffer[BUFFER_SIZE];
int  head, tail, full_flag;
```

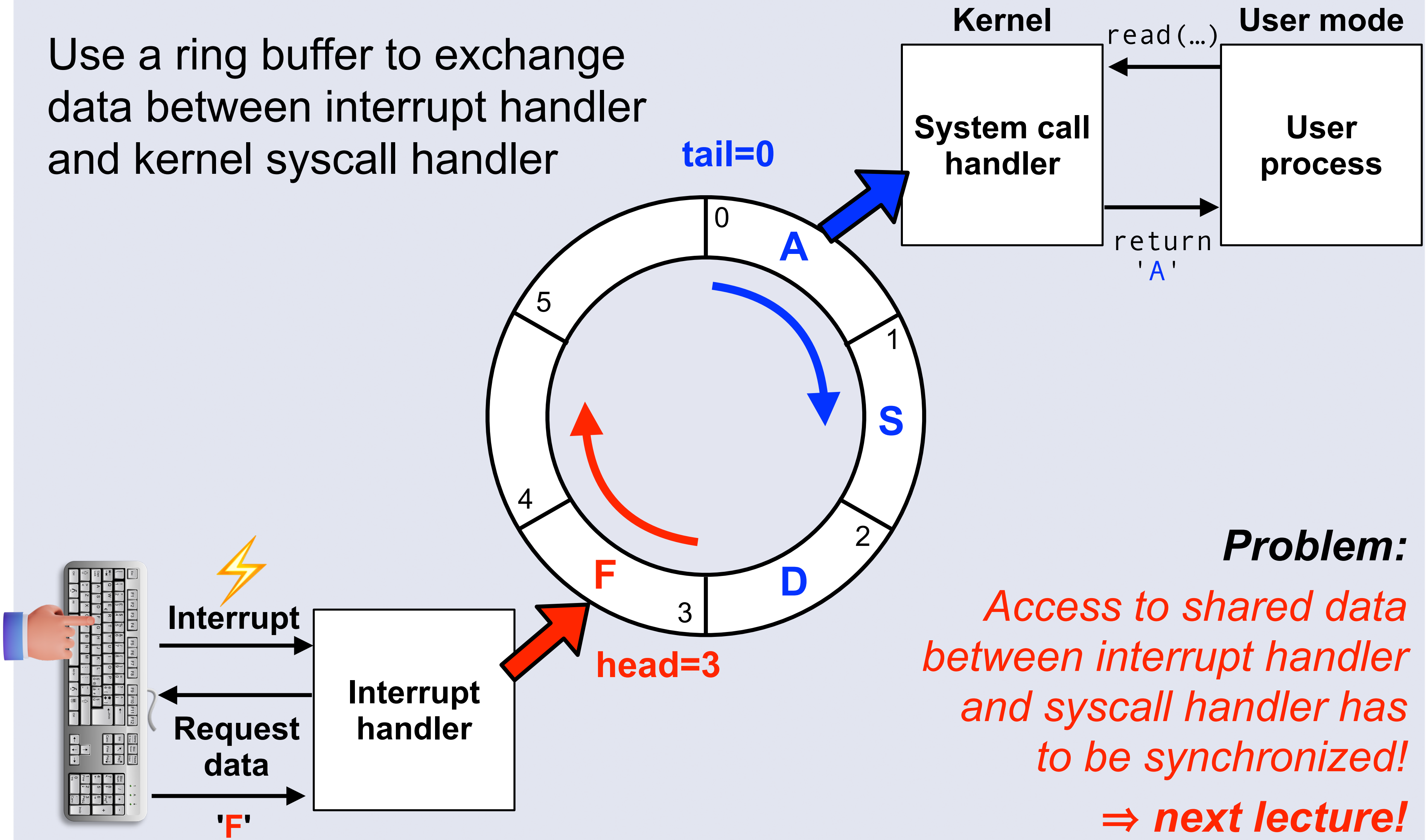
```
int rb_write(char c) {
  if (buffer_is_full()) {
    return -1;
  } else {
    ringbuffer[head] = c;
    head = (head+1) % BUFFER_SIZE;
    if (head == tail)
      full_flag = 1;
  }
}

int rb_read(char *c) {
  if (buffer_is_empty()) {
    return -1;
  } else {
    *c = ringbuffer[tail];
    tail = (tail+1) % BUFFER_SIZE;
    full_flag = 0;
  }
}
```



Kernel↔interrupt handler communication

Use a ring buffer to exchange data between interrupt handler and kernel syscall handler



Problem:

Access to shared data between interrupt handler and syscall handler has to be synchronized!

⇒ next lecture!

Conclusion

Universität Bamberg

- **Devices** can be assigned to different **classes**
  - Character, block, others (e.g. network)
- **Exchanging data** with devices
  - Polling, interrupts, DMA
- Device **drivers** and a common **API**
  - Unix file I/O API, device switch table
- **Buffering** to decouple interrupt handler and syscalls
  - Ring buffers are versatile
  - Requires **synchronization** ⇒ next lecture!

References

1. Chris Siebenmann,
   How major and minor device numbers worked in V7 Unix,
   https://utcc.utoronto.ca/~cks/space/blog/unix/V7DeviceNumbersHow
2. William C. Benton,
   Kernel Korner: Loadable Kernel Module Exploits,
   Linux Journal January 2001,
   https://dl.acm.org/doi/fullHtml/10.5555/509824.509831
3. Michael Larabel,
   OpenBSD Drops Support For Loadable Kernel Modules,
   https://www.phoronix.com/scan.php?page=news_item&px=MTgyNDI
4. UIUC CS241 SystemProgramming Wiki,
   https://csresources.github.io/SystemProgrammingWiki/SystemProgramming/Synchronization,-Part-8:-Ring-Buffer-Example/