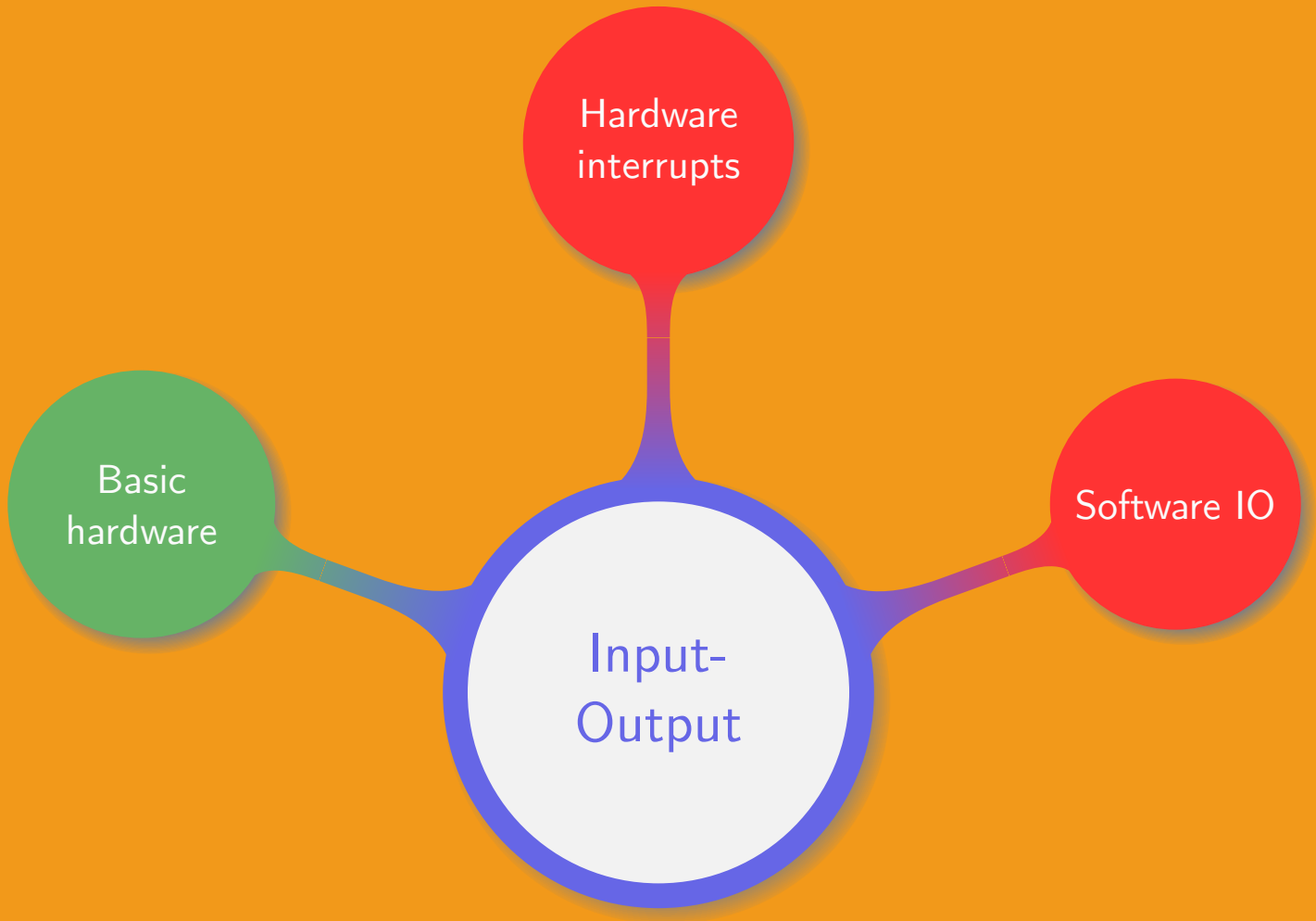
Introduction to Operating Systems

7. Input-Output

Manuel – Fall 2020

Chapter organisation



OS and Input-Output

The OS controls all the IO:

- Send commands to the device
- Handle errors
- Catch interrupts

Two main device categories:

- Block devices:
  - Stores information in blocks of fixed size
  - Can directly access any block independently of other ones
- Character devices:
  - Delivers or accepts a stream of characters
  - Not addressable, no block structure
- Clock: cause interrupts at some given interval

The OS provides a simple way to interface with hardware

Device controller

Most devices have two parts:

- Mechanical: the device itself
- Electronic: the part allowing to communicate with the device

The electronic part is called the device controller:

- Allows to handle mechanical part in an easier way
- Performs error corrections for instance in the case of a disk
- Prepares and assemble blocks of bits in a buffer
- The blocks are then copied into the memory

Memory-mapped IO

The CPU communicates with the device using *control registers*:

- OS writes on registers to: send|accept data, switch device on|off
- OS reads from registers to: know device's state

Modern approach:

- Map the buffer to a memory address
- Map each register to a unique memory address or IO port

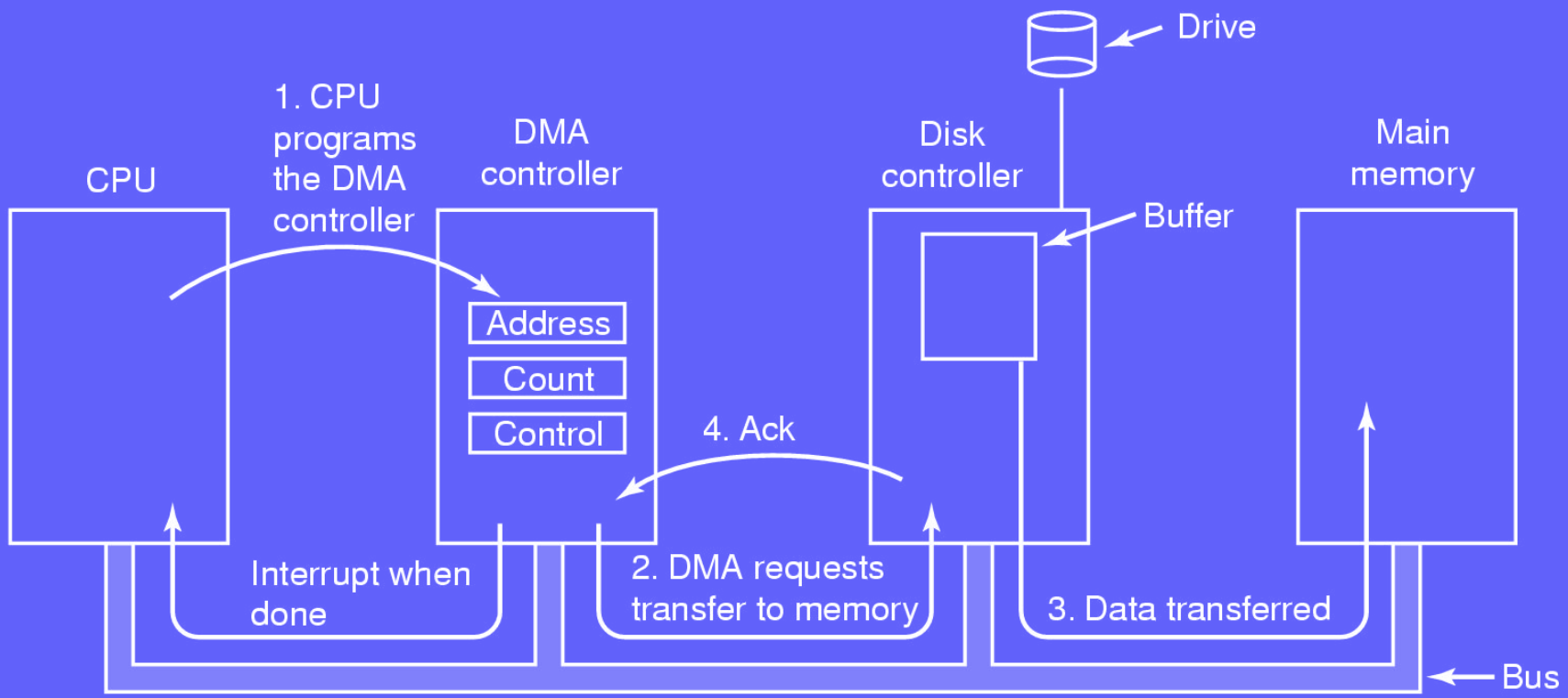
Memory-mapped IO

Strengths:

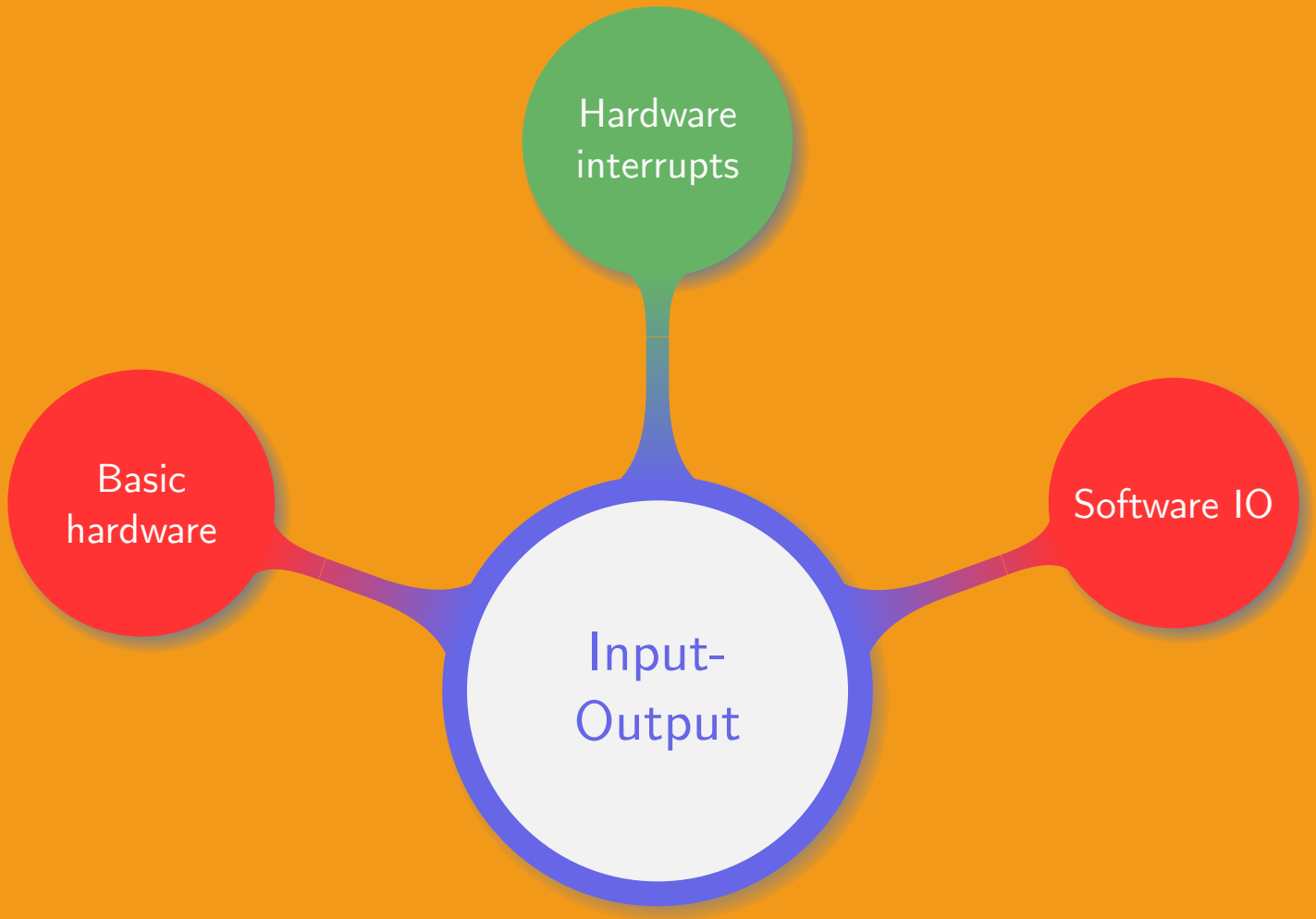
- Access memory not hardware: no need for assembly
- No special protection required: control register address space not included in the virtual address space
- Flexible: a specific user can be given access to a particular device
- Different drivers in different address spaces: reduces kernel size and does not incur any interference between drivers

Since memory words are cached what if the content of a control register is cached?

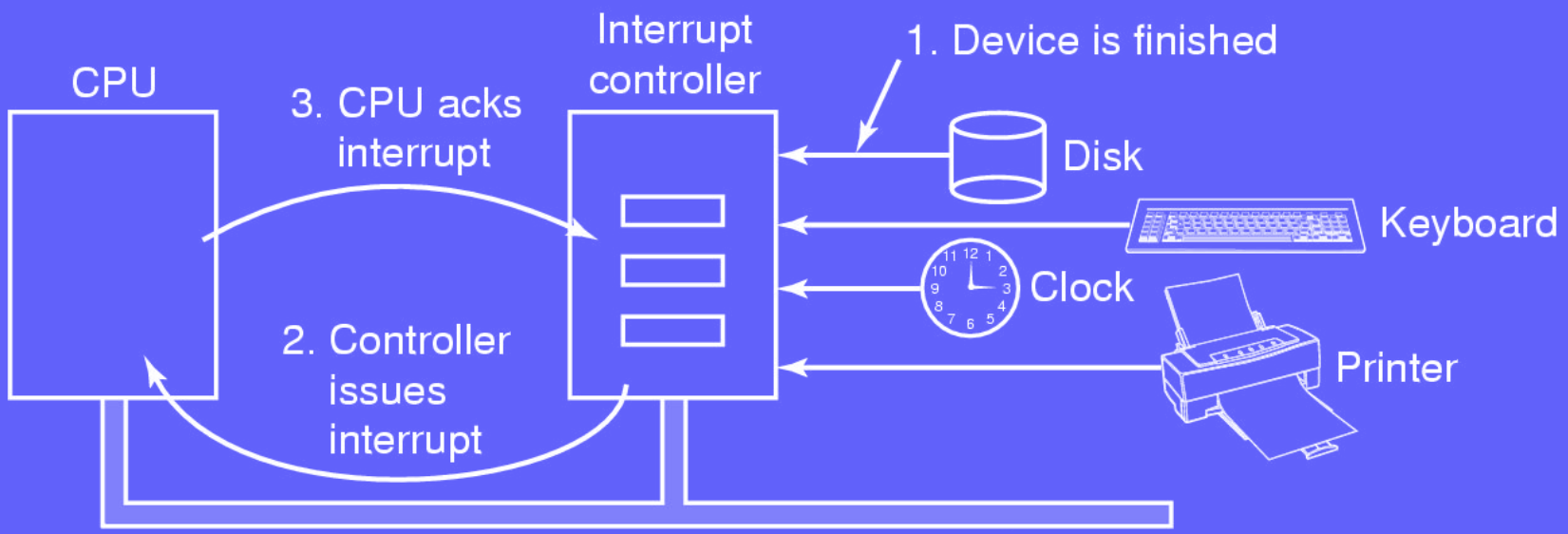
Direct Memory Access



Chapter organisation



Reminders


CPU
3. CPU acks interrupt
Interrupt controller
1. Device is finished
Disk
Keyboard
Clock
Printer
2. Controller issues interrupt

Precise interrupts

Interrupt handling on a basic setup:

- Push program counter (PC) and PSW on the stack
- Handle the interrupt
- Retrieve program counter and PSW from the stack
- Resume a process

Precise interrupt:

- Program counter is saved in a known place
- All instructions before PC have been completed
- No instruction after the one pointed by PC have be executed
- Execution state of the instruction pointed by PC is known

Precise interrupts

Interrupt handling on a basic setup:

- Push program counter (PC) and PSW on the stack
- Handle the interrupt
- Retrieve program counter and PSW from the stack
- Resume a process

Precise interrupt:

- Program counter is saved in a known place
- All instructions before PC have been completed
- No instruction after the one pointed by PC have be executed
- Execution state of the instruction pointed by PC is known

What are the consequences of more advanced architectures such as pipelined or superscalar CPU?

Imprecise interrupts

An interrupt which is not precise is called imprecise interrupt

Difficult to figure out what happened and what has to happen:

- Instructions near PC are in different stages of completion
- General state can be recovered if given many details
- Code to resume process is complex
- The more details the more memory used

Imprecise interrupts

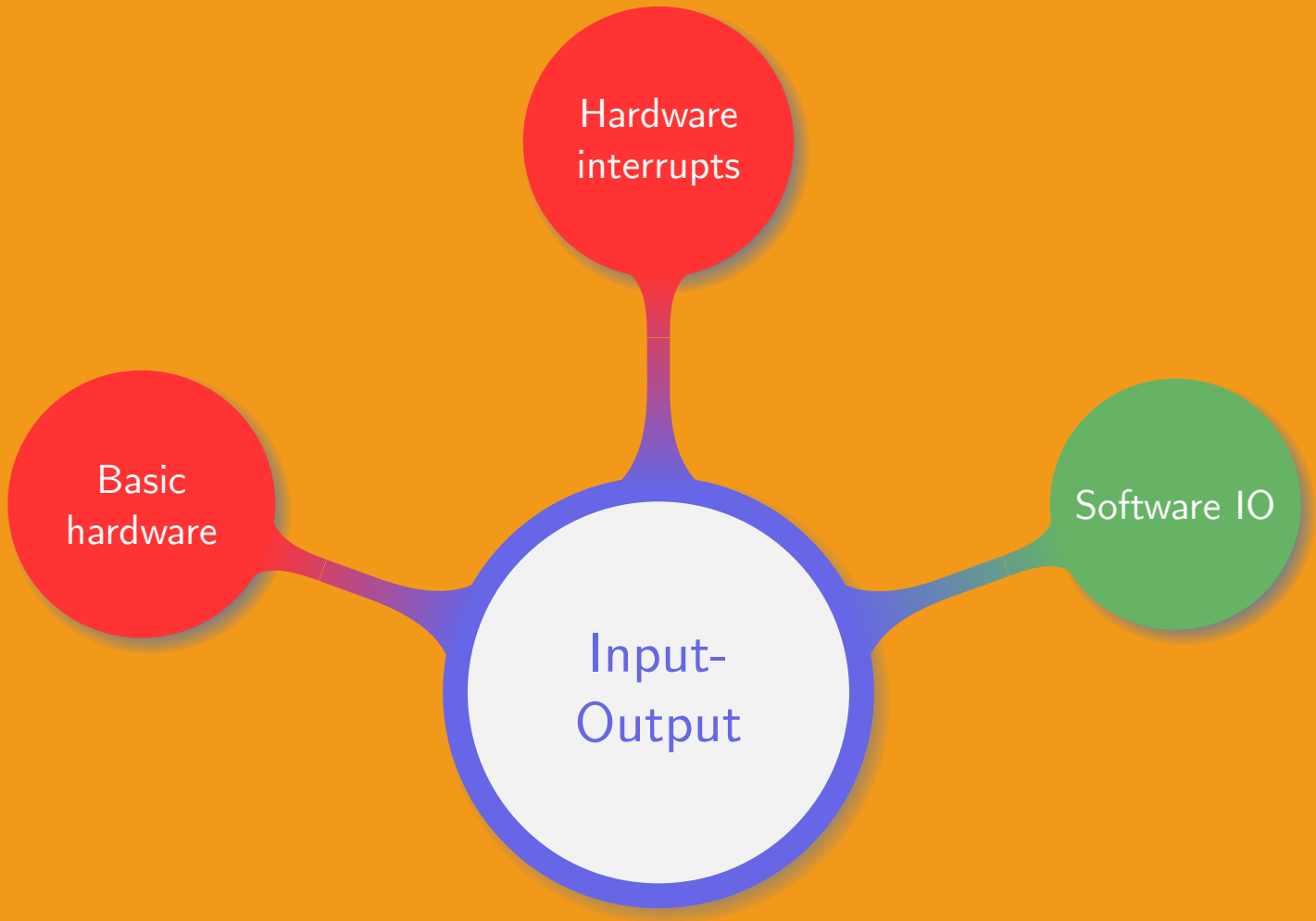
An interrupt which is not precise is called imprecise interrupt

Difficult to figure out what happened and what has to happen:

- Instructions near PC are in different stages of completion
- General state can be recovered if given many details
- Code to resume process is complex
- The more details the more memory used

Imprecise interrupts are slow to process, what is an optimal strategy?

Chapter organisation



Communications strategies

Three communications strategies:

Communications strategies

Three communications strategies:

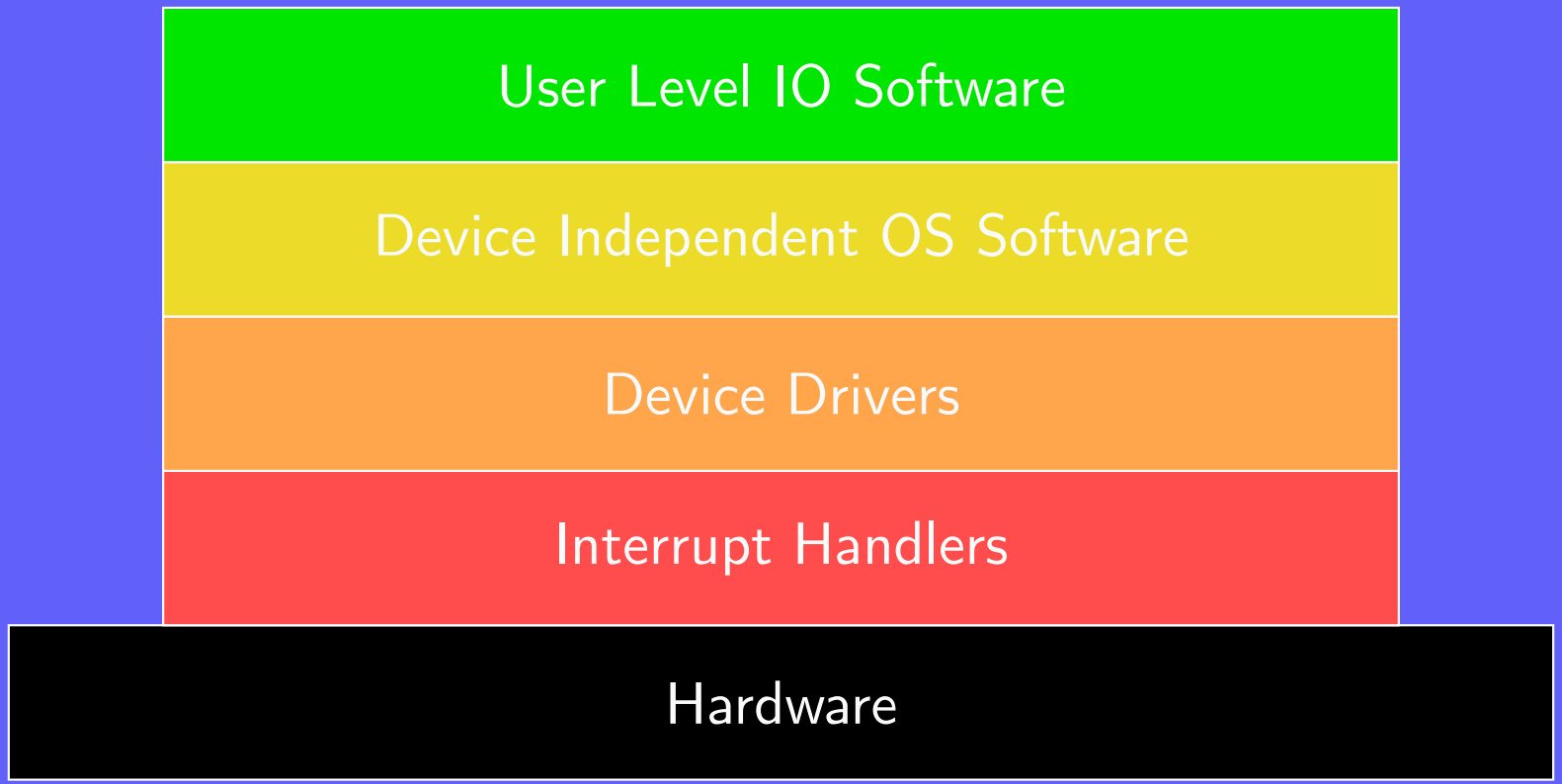
- Programmed IO:
  - Copy data into kernel space
  - Fill up device register and wait in tight loop until register is empty
  - Fill up, wait, fill up, wait, etc.
- Interrupt IO:
  - Copy data into kernel space, then fill up device register
  - The current process is blocked, the scheduler is called to let another process run
  - When the register is empty an interrupt is sent, the new current process is stopped
  - Filled up, block, resume, fill up, block, etc.
- DMA: similar to programmed IO, but DMA does all the work.

Key concepts

Main goals on the design of IO software:

- Device independence: whatever the support, files are handled the same way
- Uniform naming: devices organised by type with a name composed of string and number
- Error handling: fix error at lowest level possible
- Synchronous vs. asynchronous: OS decides if interrupt driven operations look blocking to user programs
- Buffer: need some temporary space to store data

Basic layers in an OS

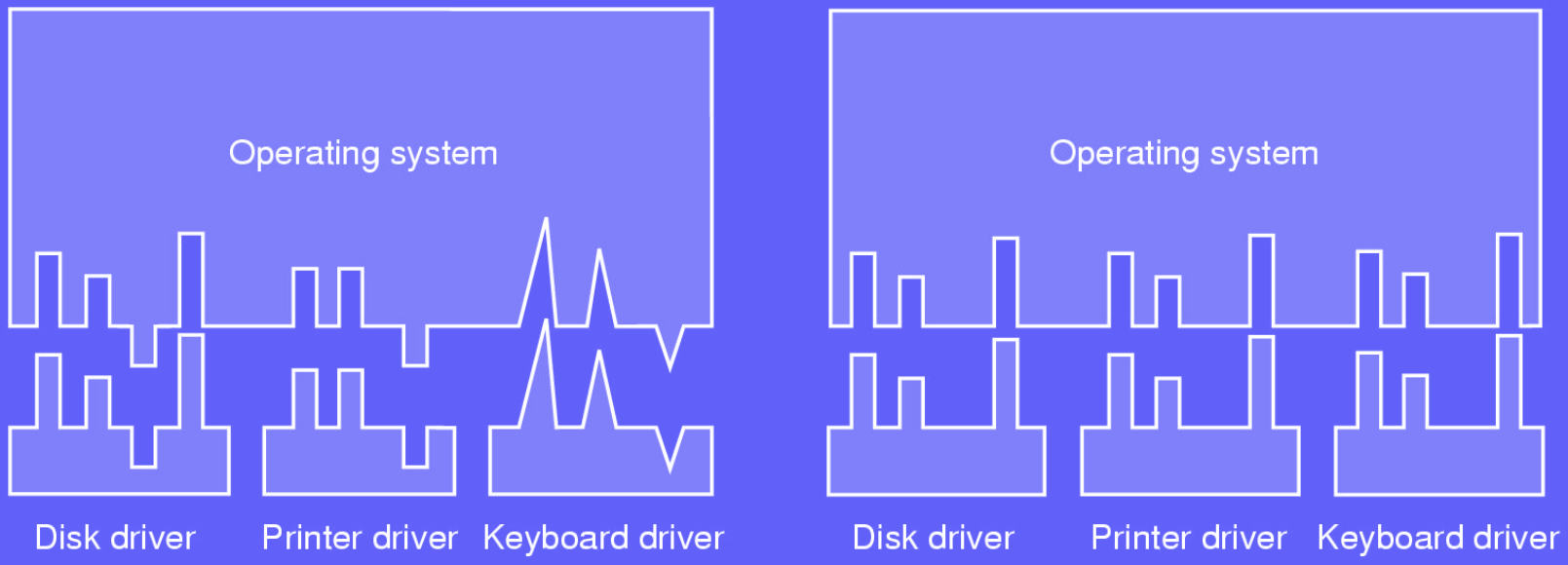


Interrupt handlers

Actions to performs on an interrupt:

1. Save registers
2. Setup a context for handling the interrupt
3. Setup a stack
4. Acknowledge interrupt controller + re-enable interrupts
5. Load registers
6. Extract information from interrupting device's controller
7. Choose a process to run next
8. Setup MMU and TLB for next process
9. Load new process registers
10. Run new process

Device drivers interface



Basic plugin design strategy:

- Similar devices have a common basic set of functionalities
- OS defines which functionalities should be implemented
- Use a table of function pointers to interface device driver with the rest of the OS
- Uniform naming at user level

Job of device drivers

Basic functions of a driver:

- Initialization
- Accept generic requests, e.g. read, write
- Log events
- Retrieve device status
- Handle device specific errors
- Specific actions depending on the device

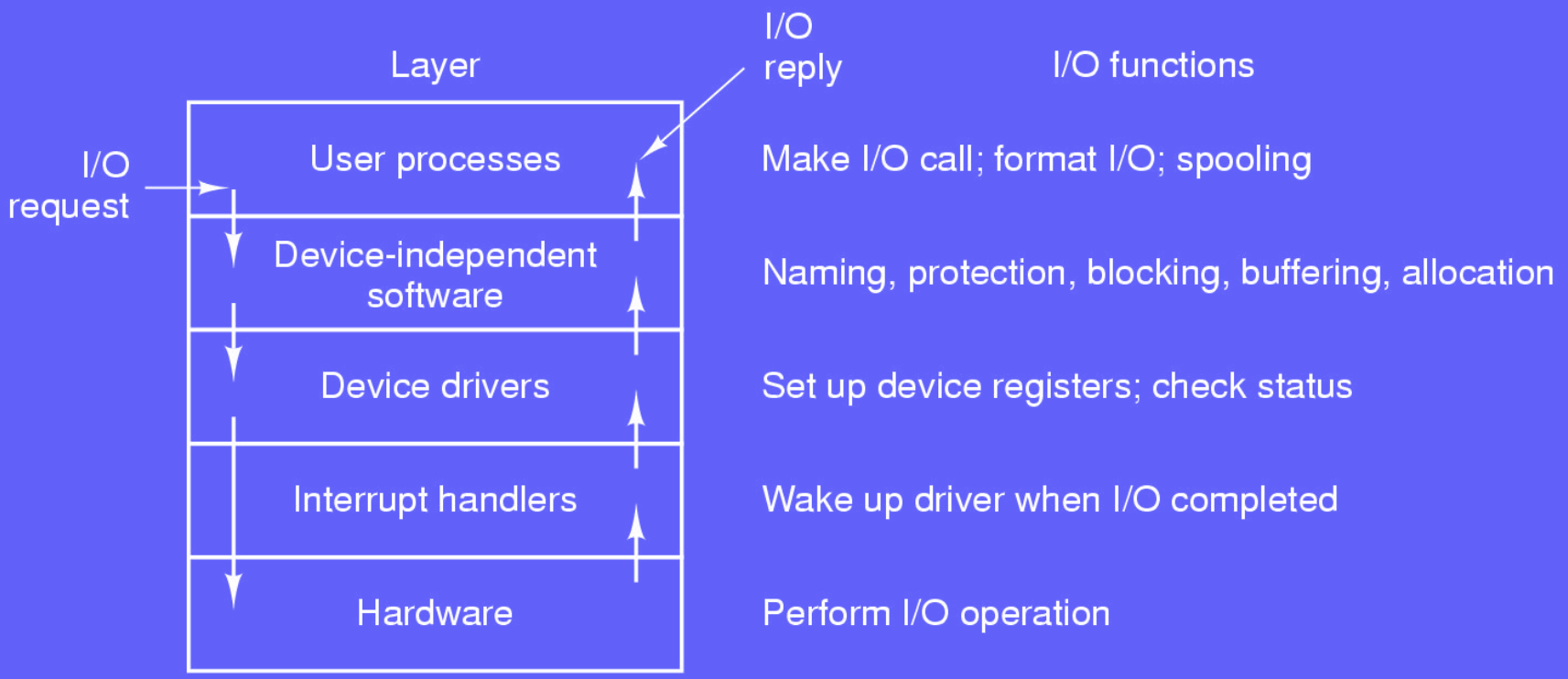
Device driver should react nicely even under special circumstances

Drivers in the OS

General remarks on drivers:

- Location: user or kernel space
- Drivers can be compiled in kernel
- Drivers can be dynamically loaded at runtime
- Drivers can call certain kernel procedures, e.g. to manage MMU, set timers
- IO errors framework is device independent
- Clean and generic API such that it is easy to write new drivers

Software IO and OS



Key points

- What are the two main categories of devices?
- Explain the difference between precise and imprecise interrupts
- List three communication strategies
- Why should implementation be done using layers?

Thank you!