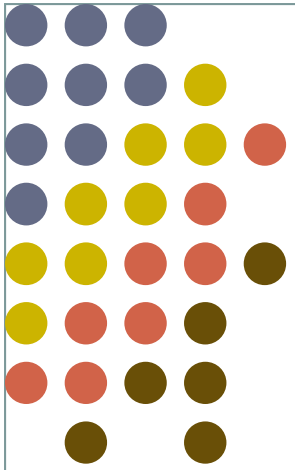


Input/Output

I/O

- One of the functions of the OS, controlling the I/O devices
- Wide range in type and speed
- The OS is concerned with how the interface between the hardware and the user is made
- The goal in designing the OS is to provide a uniform interface (e.g. if I replace my HD I'd like to see the same sort of filesystem)

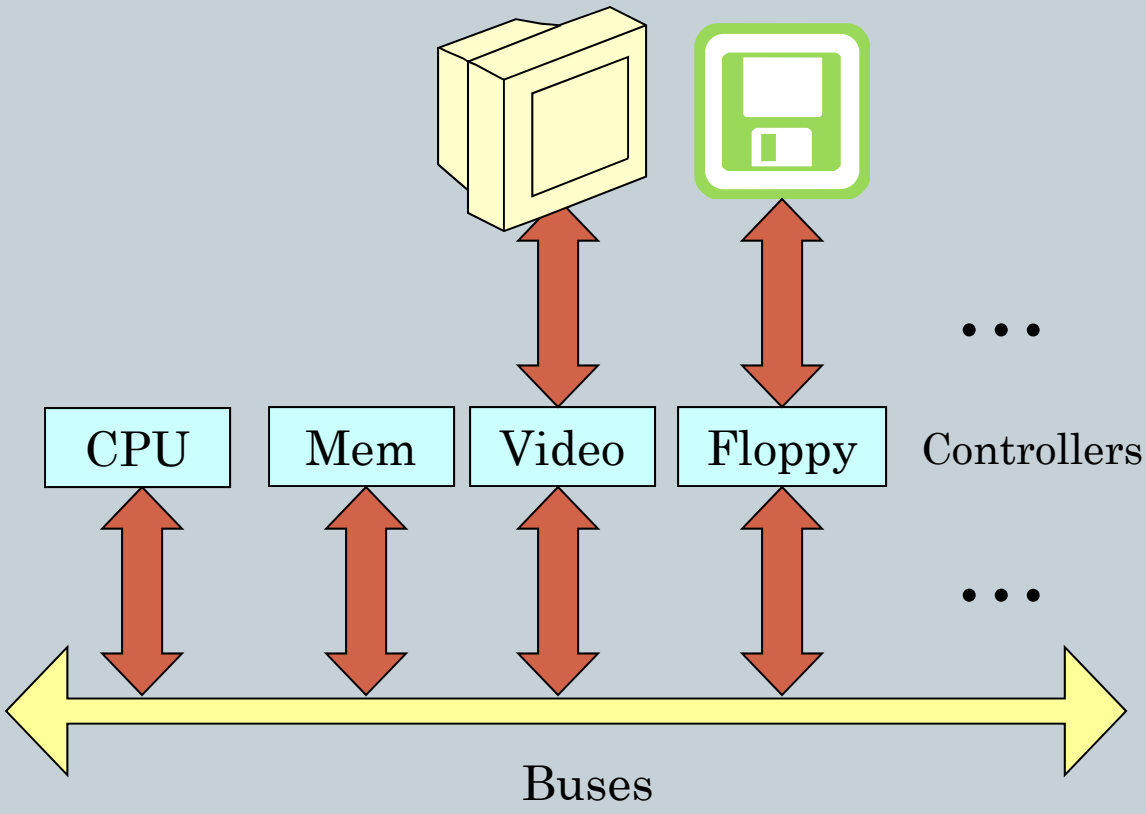
Types of devices

- Block devices: random addressable, blocks of fixed size
  - Example: disks
- Character: stream of date, they don't have a *seek* operation
  - Example: terminal, printer, mouse, etc.
- Others:
  - Timers and clocks

Device	Data rate [byte/s]
Keyboard	10 bytes/s
Mouse	100 bytes/s
56K modem	7Kb/s
Telephone channel	8Kb/s
ISDN line	16Kb/s
Scanner	400Kb/s
Ethernet (10Mbit)	1.25Mb/s
USB	1.5Mb/s
Digital camcorder	4Mb/s
IDE disk	5Mb/s
40x CD	5Mb/s
Ethernet (100Mbit)	12.5Mb/s
ISA bus	16.7Mb/s
Firewire (IEEE1394)	50Mb/s
XGA monitor	60Mb/s
SCSI ultra 2 disk	80Mb/s
Gigabit Ethernet	125Mb/s
PCI bus	528Mb/s

Computer hardware

- Processors
- Memory
- I/O devices
- Buses



Interface

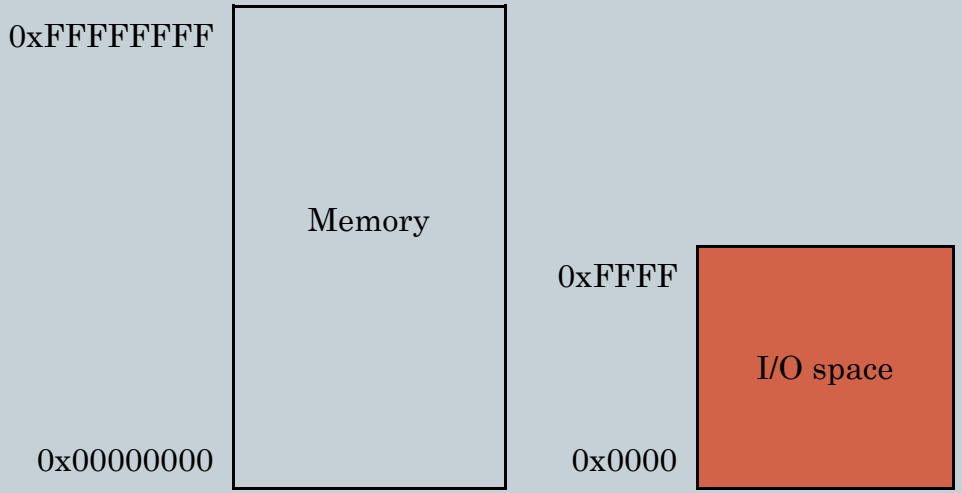
- Controller and device
  - Controller: a piece of electronics
  - Device: the mechanics
- Interface
  - E.g. for a disk, a bit-stream with a *preamble*, a certain amount of bits (4096) of a sector, and finally an *error-correcting code* (ECC).

Example: floppy drive

- Specific chip (NEC PD765)
- 16 different commands
- Load between 1 and 9 bytes into a device register
- Read/Write require 13 parameters packed into 9 bytes
- Reply from the device consists of 7 bytes (23 parameters)
- Control of the motor (on/off)

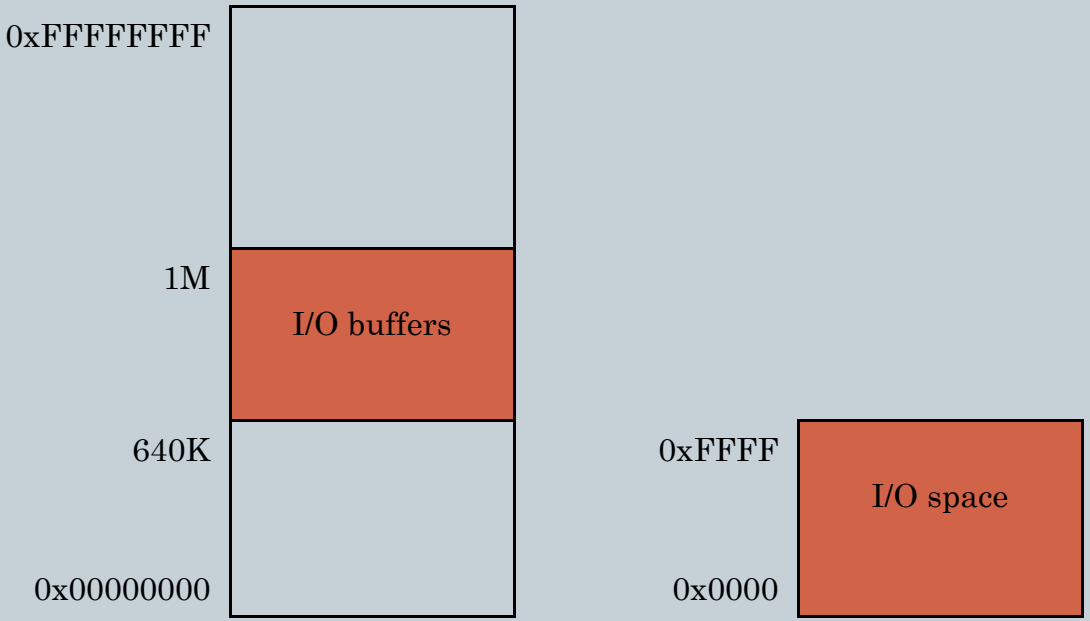
Memory mapped I/O

- Two solutions:
  - Memory mapped registers or buffers (e.g. the display buffer)
  - Special I/O instructions
    - IN/OUT instructions
  - A mix of the two



Pentium example

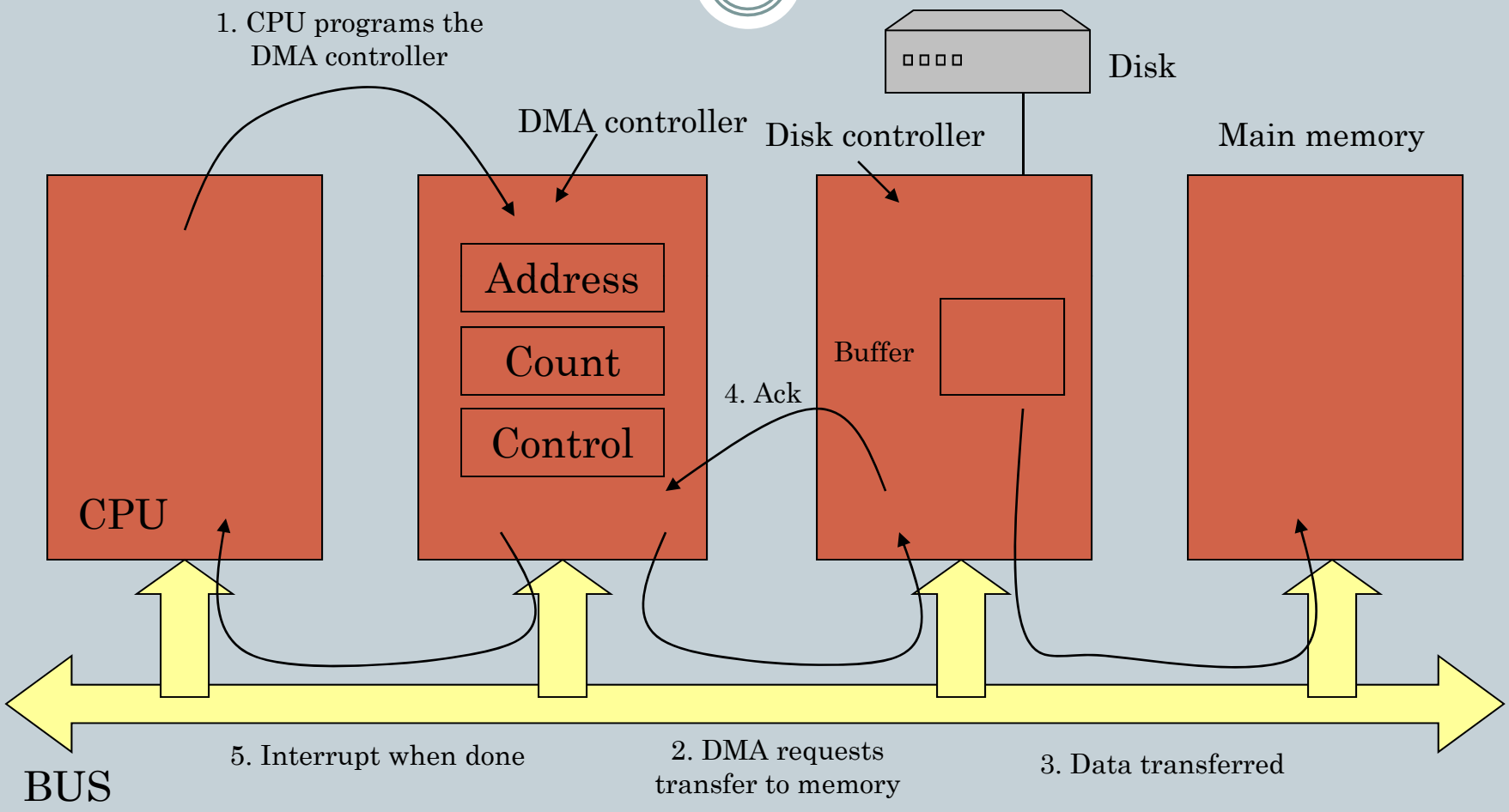
- I/O ports (using IN and OUT): 0 to 64K
- I/O buffers: from 640K to 1M



How do they fare in practice?

- Address is put in the bus.
  - Another line tells whether the address is I/O or memory.
- For memory mapped I/O:
  - C/C++ instructions can write into I/O registers
  - No special protection is needed (beside the usual one needed to protect memory pages)
  - Every instruction can reference memory (e.g. can do a TEST without loading into a register first)
- For special I/O instructions:
  - Don't need to disable caching selectively as for memory mapped
  - Multiple buses, need a way to send the memory address on all the buses where it might be required (e.g. memory bus, PCI, etc.)
- Pentium solution: the PCI bridge chip(s) does the filtering of addresses (e.g. the 640 to 1M area)

DMA


1. CPU programs the DMA controller
Disk
DMA controller
Disk controller
Main memory
Address
Count
Control
Buffer
4. Ack
CPU
BUS
5. Interrupt when done
2. DMA requests transfer to memory
3. Data transferred

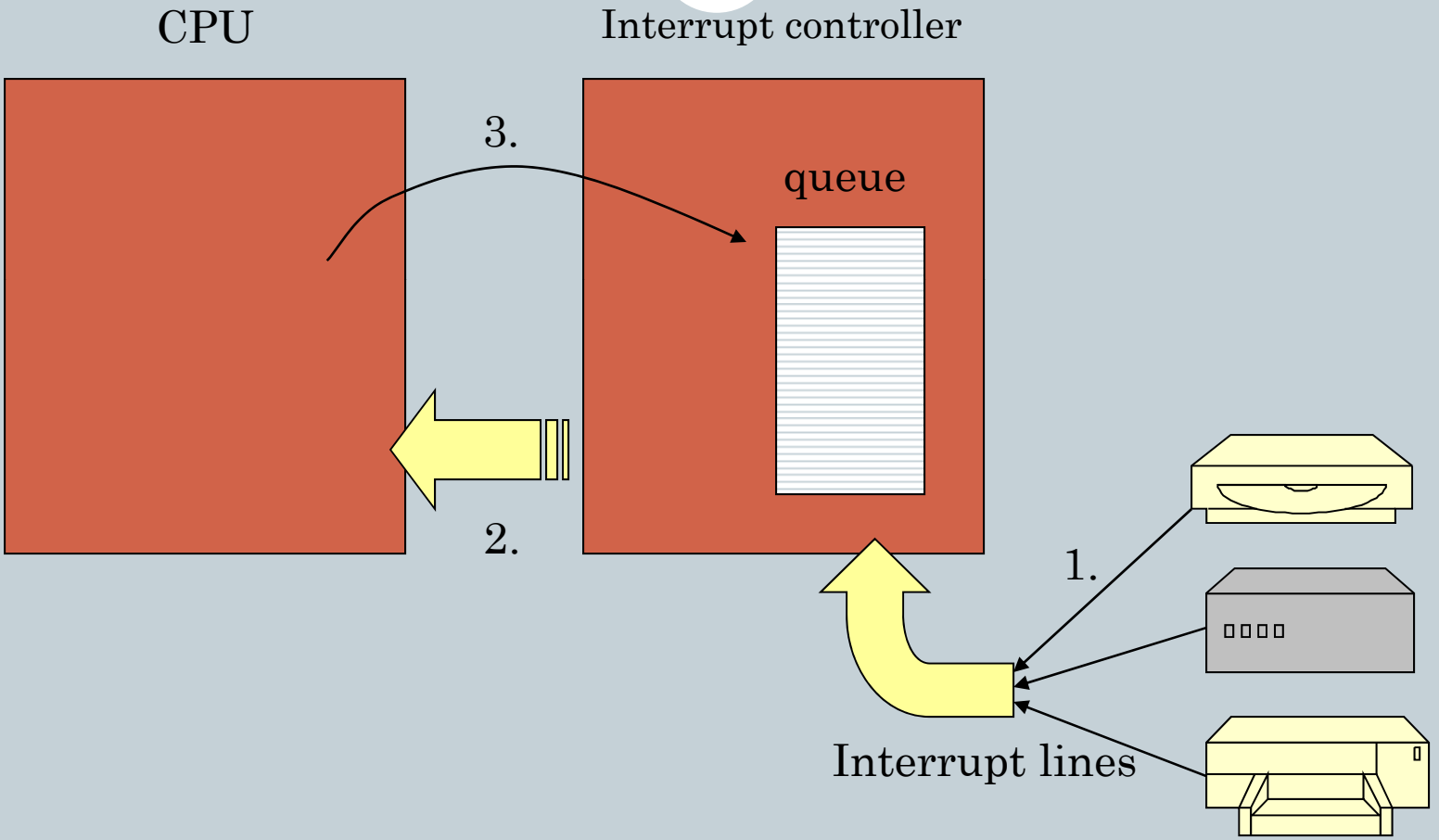
DMA's steps

1. CPU programs the DMA controller by setting its registers (where and what to transfer)
2. DMA controller issues the request to read from the disk. The memory address where to write to is passed along with the READ request
3. Data is read and then transferred to memory directly by the HD controller
4. ACK is sent to the DMA controller to tell the transfer is completed
5. The DMA controller interrupts the CPU to tell the data is in main memory now.

DMA sophistication

- One word at a time or block mode:
  - Word at a time: cycle stealing
  - Block or burst mode
- Many lines and multiple requests
  - Similar in principle but can accept many requests simultaneously
- DMA controllers use physical addresses
  - Not virtual addresses

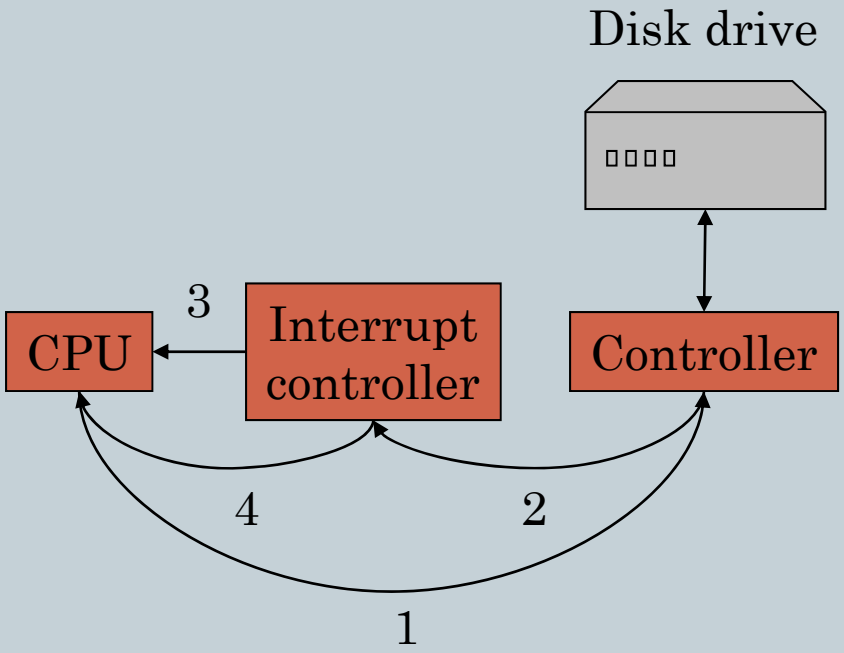
Interrupts



Interrupt (old slide)

- A piece of hardware called “interrupt controller”

1. CPU issues the I/O request via the device driver
2. On termination the device signals the CPU’s interrupt controller (if the interrupt controller is not busy servicing another higher priority interrupt)
3. If the interrupt can be handled then the controller asserts a pin on the CPU.
4. The interrupt controller puts the address of the device into the bus



On occurrence of interrupt

- Save information about the state of the CPU:
  - PC at least
- Where to save the information:
  - Stack (page faults?), user or kernel mode?
  - Internal registers, beware of a second interrupt, interrupting the copy, delay the ACK
- More troubles:
  - Pipeline, at the moment of interrupt, some instructions are only partly executed, the PC might not even point to the last fully executed instruction
  - On superscalar CPU things are even worse (instructions executed out of order)

Definition

- Precise interrupt, when:
  - The PC is saved in a known place
  - All instructions before the one pointed to by the PC have fully executed
  - No instruction beyond the one pointed to by the PC has been executed
  - The execution state of the instruction pointed to by the PC is known

Pentium

- Starting from the PPRO the Pentium has a superscalar architecture
- The price paid to have precise interrupts (and being compatible with 486's) is in chip complexity
- A part of the chip allows instructions to complete after the interrupt has been issued and empties the remainder of the pipeline

I/O Software

General goals

- Device independence:
  - E.g. use the floppy as you use the HD or CD
- Uniform naming:
  - Related to device independence
  - Use a common naming system (e.g. within the filesytem hierarchy)
- Error handling:
  - Errors should be handled as low-level as possible, e.g. an error on disk might be serviced by re-reading the sector already from the controller
- Make calls appear synchronous:
  - Blocking vs. non-blocking
- Buffering:
  - Keep data being transferred at a constant rate

Ways of doing I/O

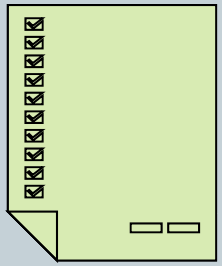
- Programmed I/O
  - The CPU does all the work
- Interrupt-driven
  - We've already said a lot about interrupts
- DMA-based
  - We've already said a lot about DMA

Programmed I/O


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

for (i = 0; i < count; i++)
{
    while (*printer_status_reg != READY);
    *printer_data_register = p[i];
}

return_to_user();
```



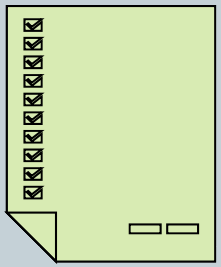
Interrupt-driven I/O


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

enable_interrupts();

while (*printer_status_reg!=READY);
*printer_data_register = p[0];

scheduler();
```



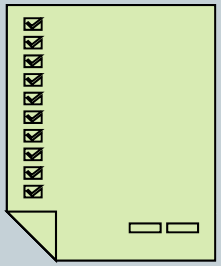
```
if (count == 0)
{
    unblock_user();
}
else
{
    *printer_data_register = p[i];
    count = count – 1;
    i = i + 1;
}

ack_interrupt();
return_from_interrupt();
```

DMA-based I/O


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

```
ack_interrupt();
unblock_user();
return_from_interrupt();
```



Software layers

User level I/O software
Device-independent OS software
Device drivers
Interrupt handlers
Hardware

Interrupt handlers

- How to hide interrupts:
  - Caller – does a Wait on a semaphore
  - Interrupt handler – does a Post on the semaphore
- The user process knows nothing of the interrupt (it just blocks for some time)

In reality the OS…

- Save any register that wasn't saved by the HW
- Set up the context for the handler to run.
  - E.g. setup TLB, MMU, page table, etc.
- Set up a stack for the handler
- ACK the interrupt controller
- Copy registers from where they were saved (stack)
  - E.g. the interrupted process not necessarily will continue next
- Run the interrupt handler
  - Maybe communicating again with the HW controller
- Choose which process to run next (scheduler)
- Setup the MMU, TLB, etc. for the process to run next
- Load the new process' registers
- Start running the new process

- A nightmare!

Device drivers

- Device-specific code talking to the device controller
  - E.g. a SCSI driver could control a set of SCSI disks or CD
- In order to access the HW, the device driver needs to be part of the kernel
  - Loadable at run time
  - To be compiled into the kernel
- Since the OS writer doesn't know in advance which device will be attached to the machine
  - Well-defined model of how the DD interacts with the kernel

Functions

- Character/block interface
  - Accept abstract read/write requests and translates into actual commands to the HW
- Check status of the device when R/W are issued
  - E.g. start/stop motor if disk is not ready
- Parameters/addresses conversion
- Blocking vs. non-blocking
- Reentrant code
  - An interrupt might cause another request on the same device driver already running

By the way…

- Device drivers are a source of troubles
  - In fact, a buggy DD can interfere with the kernel leading to unpredictable results
  - Likely a system crash!
- A nice architecture would see the DD not being part of the kernel
  - Microkernel architecture we mentioned some time ago

Device independent I/O

- Uniform interfacing
- Buffering
- Error reporting
- Allocating and releasing dedicated devices
- Providing device-independent block size

Uniform interfacing

- Required, otherwise how could the OS call the DD?
- Not all devices are identical, but…
  - There is a finite number of classes though
- Protection
  - E.g. it is better not to leave anyone the control of the printer

Buffering

- Continuous flow of data
  - E.g. from a modem
  - Requires something like *double-buffering*
  - Time is required to move data around (kernel, user space)
- Buffering can affect performance
  - Delay

Other issues

- Error reporting
  - The OS should try to do its best before complaining
- Allocation of dedicated devices
  - Possible deadlocks
  - E.g. a CD burner
- Uniform block size
  - Show all disks as having the same block size

User space I/O

- Spooling:
  - Daemon + spooling directory
  - It solves the problem of not leaving control of devices completely to the user
  - Example: printer