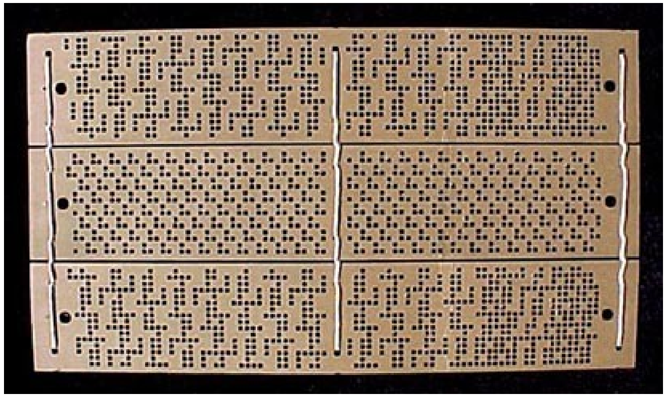
I/O Management
Intro

Chapter 5 - 5.3

Learning Outcomes

- A high-level understanding of the properties of a variety of I/O devices.
- An understanding of methods of interacting with I/O devices.



commodore
RECORD
PLAY
REWIND
F.FWD
STOP
EJECT

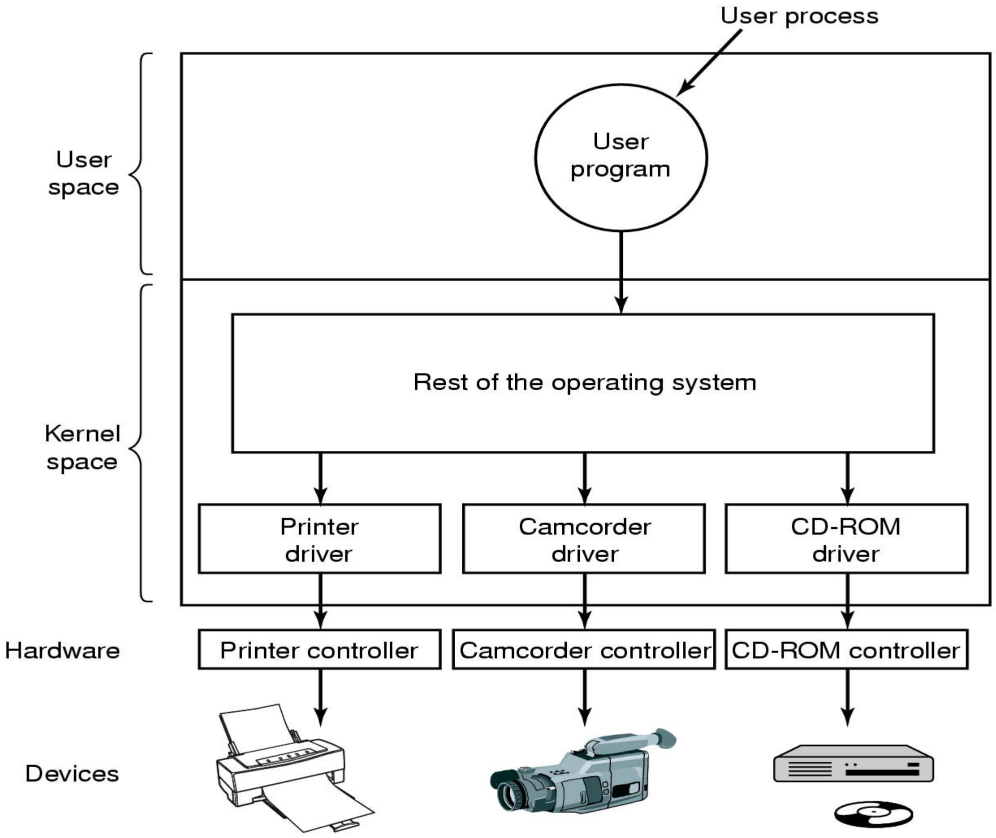
Seagate

I/O Devices

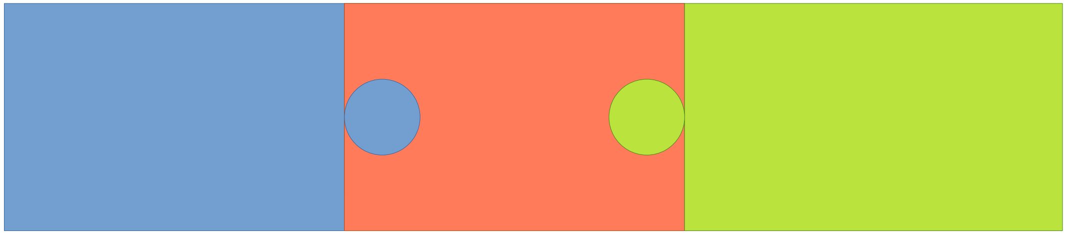
- There exists a huge variety of I/O devices
- Challenge:
  - Uniform and efficient approach to I/O

Device Drivers

- Logical position of device drivers is shown here
- Drivers (originally) compiled into the kernel
  - Including OS/161
  - Device installers were technicians
  - Number and types of devices rarely changed
- Nowadays they are dynamically loaded when needed
  - Linux modules
  - Typical users (device installers) can't build kernels
  - Number and types vary greatly
    - Even while OS is running (e.g hot-plug USB devices)



Device Drivers



- **Drivers classified into similar categories**
  - Block devices and character (stream of data) device
- **OS defines a standard (internal) interface to the different classes of devices**
  - Example: USB *Human Input Device* (HID) class specifications
    - human input devices follow a set of rules making it easier to design a standard interface.

USB Device Classes

Base Class	Descriptor Usage	Description
00h	Device	Use class information in the Interface Descriptors
01h	Interface	Audio
02h	Both	Communications and CDC Control
03h	Interface	HID (Human Interface Device)
05h	Interface	Physical
06h	Interface	Image
07h	Interface	Printer
08h	Interface	Mass Storage
09h	Device	Hub
0Ah	Interface	CDC-Data
0Bh	Interface	Smart Card
0Dh	Interface	Content Security
0Eh	Interface	Video
0Fh	Interface	Personal Healthcare
10h	Interface	Audio/Video Devices
DCh	Both	Diagnostic Device
E0h	Interface	Wireless Controller
EFh	Both	Miscellaneous
FEh	Interface	Application Specific
FFh	Both	Vendor Specific

I/O Device Handling

- Data rate
  - May be differences of several orders of magnitude between the data transfer rates

  - Example: Assume 1000 cycles/byte I/O
    - Keyboard needs 10 KHz processor to keep up
    - Gigabit Ethernet needs 100 GHz processor…..

Sample Data Rates

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

USB 3.0 625 MB/s (5 Gb/s)
Thunderbolt 2.5GB/sec (20 Gb/s)
PCIe v3.0 x16 16GB/s

Device Drivers

- **Device drivers job**
  - translate request through the device-independent standard interface (open, close, read, write) into appropriate sequence of commands (register manipulations) for the particular hardware
  - Initialise the hardware at boot time, and shut it down cleanly at shutdown

Device Driver

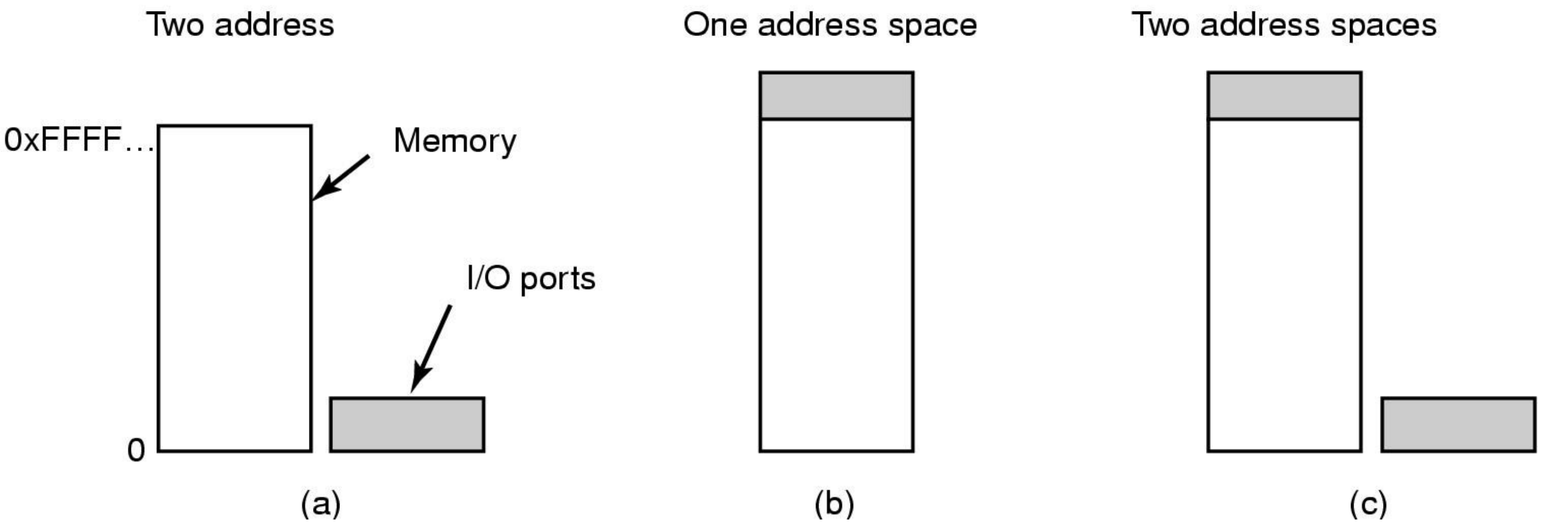
- **After issuing the command to the device, the device either**
  - Completes immediately and the driver simply returns to the caller
  - Or, device must process the request and the driver usually blocks waiting for an I/O complete interrupt.
- **Drivers are thread-safe** as they can be called by another process while a process is already blocked in the driver.
  - Thead-safe: Synchronised…

Device-Independent I/O Code

- OS Software can support multiple devices
- Divide I/O software into device-dependent and device-independent I/O software
- Device independent software includes
  - Buffer or Buffer-cache management
  - TCP/IP stack
  - Sound multiplexing
  - Error reporting

Accessing Devices

Accessing I/O Controllers

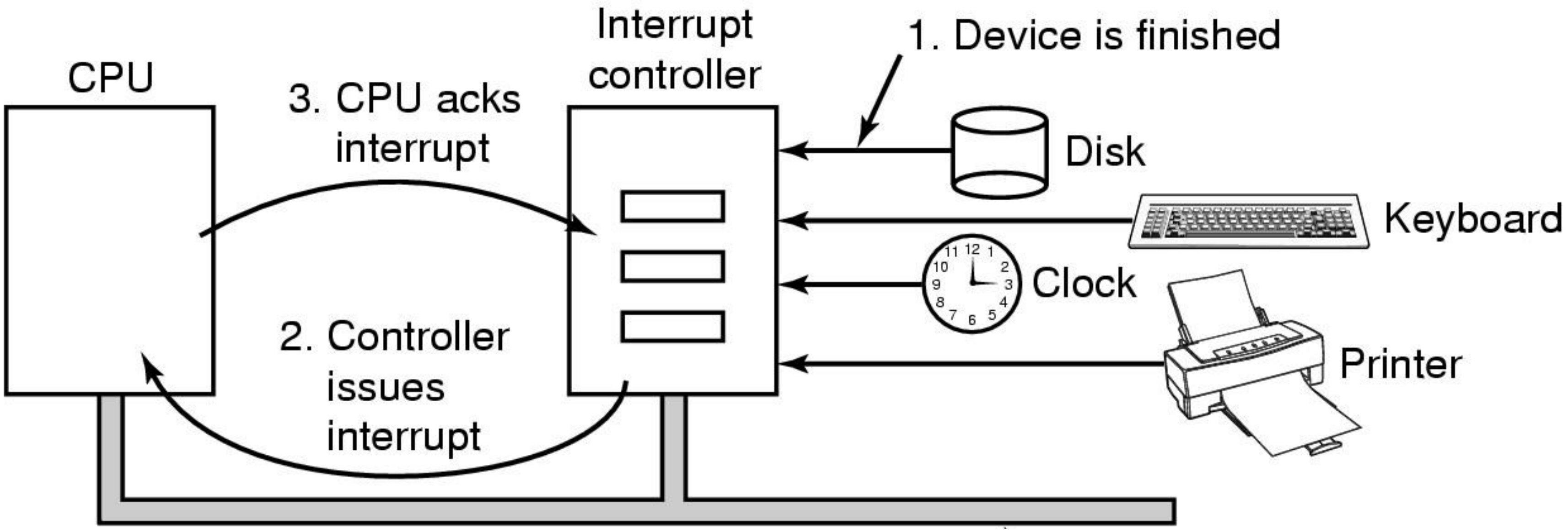


a) **Separate I/O and memory space**
   - I/O controller registers appear as I/O ports
   - Accessed with special I/O instructions

b) **Memory-mapped I/O**
   - Controller registers appear as memory
   - Use normal load/store instructions to access

c) **Hybrid**
   - x86 has both ports and memory mapped I/O

Interrupts

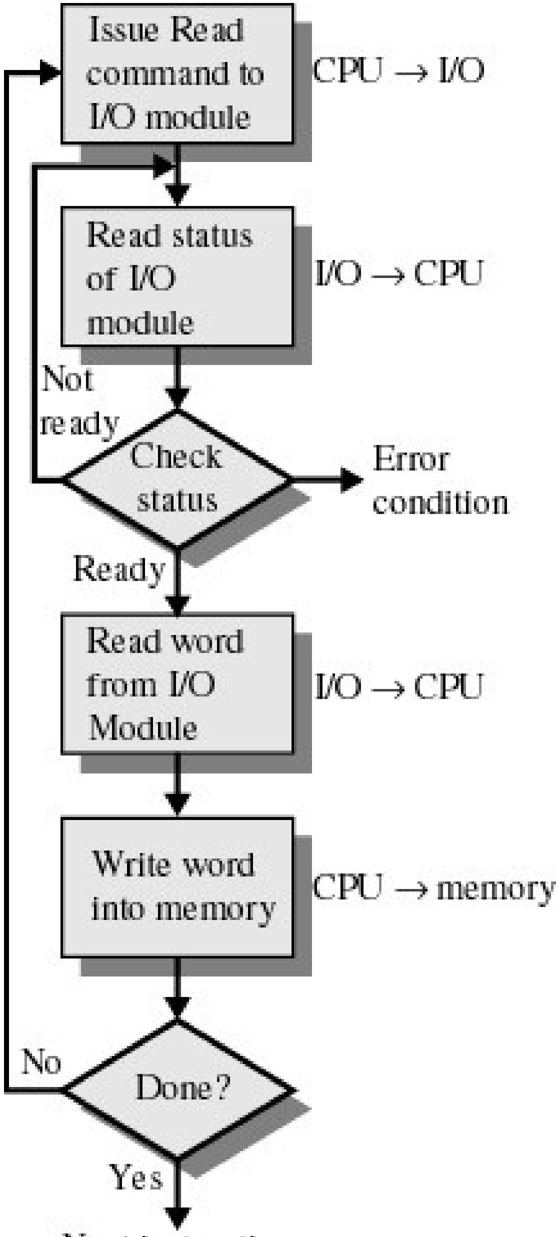


- Devices connected to an *Interrupt Controller* via lines on an I/O bus (e.g. PCI)
- Interrupt Controller signals interrupt to CPU and is eventually acknowledged.
- Exact details are architecture specific.

I/O Interaction

Programmed I/O

- Also called *polling*, or *busy waiting*
- CPU initiates I/O
- I/O device performs the I/O
- On completion, device updates a status register
- No interrupts occur
- CPU loops until it detects operation is complete
  - Wastes CPU cycles

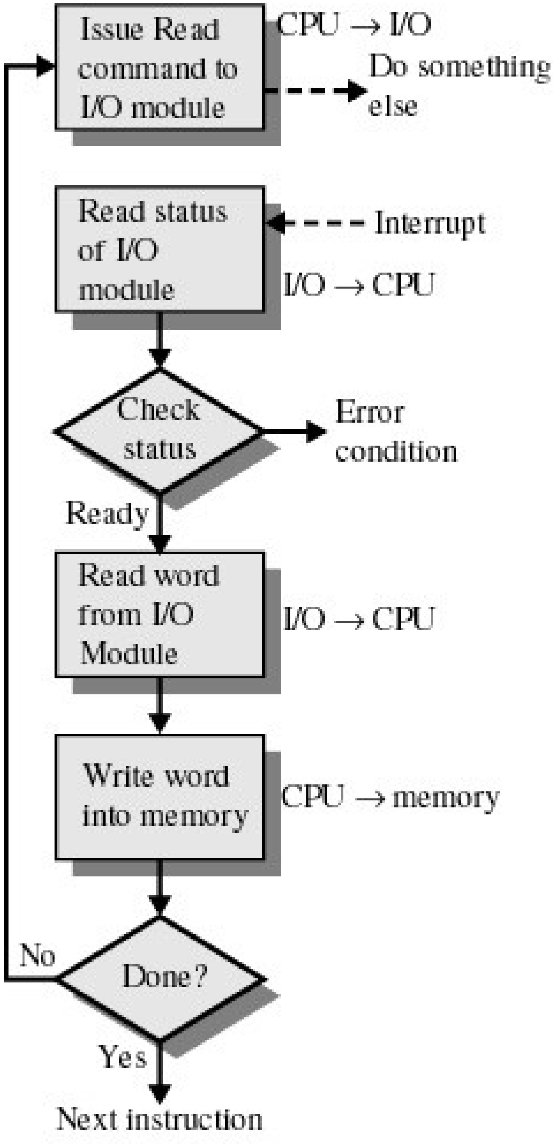


(a) Programmed I/O

Interrupt-Driven I/O

- Processor is interrupted when I/O module (controller) ready to exchange data
- Processor is free to do other work
- No needless waiting

- Copying data is still slow

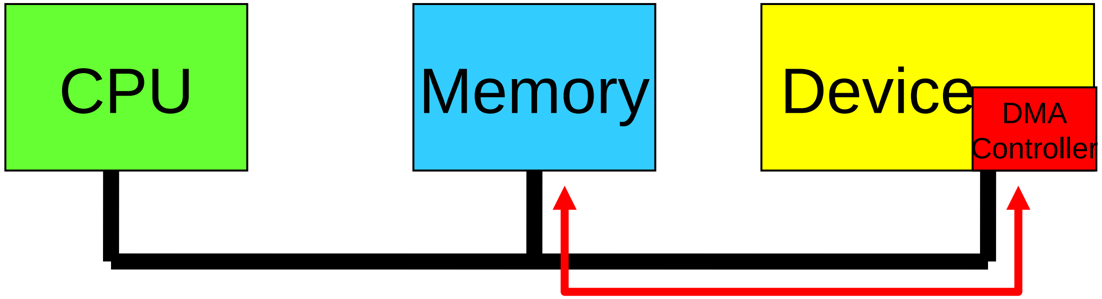


(b) Interrupt-driven I/O

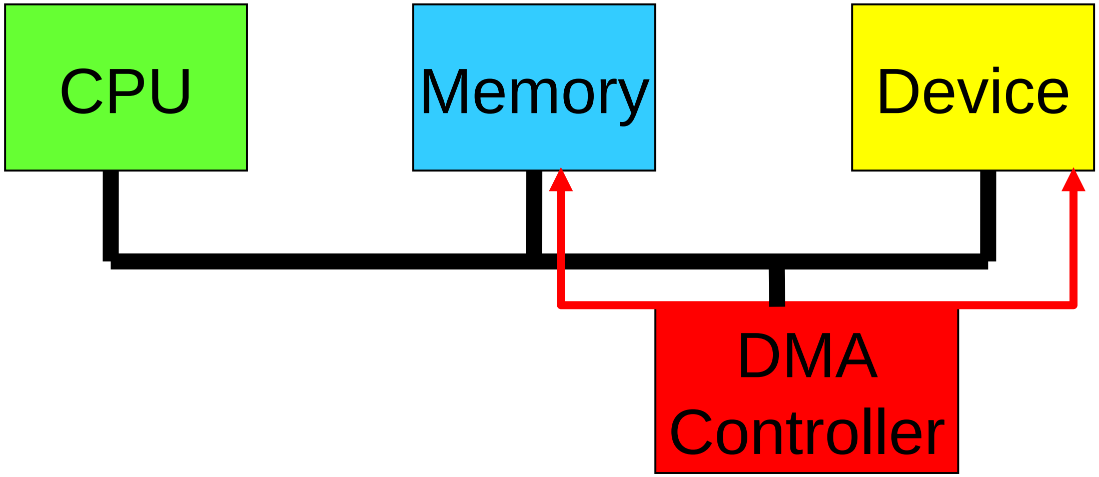
Direct Memory Access

- Transfers data directly between Memory and Device
- CPU not needed for copying

DMA Controller in Device

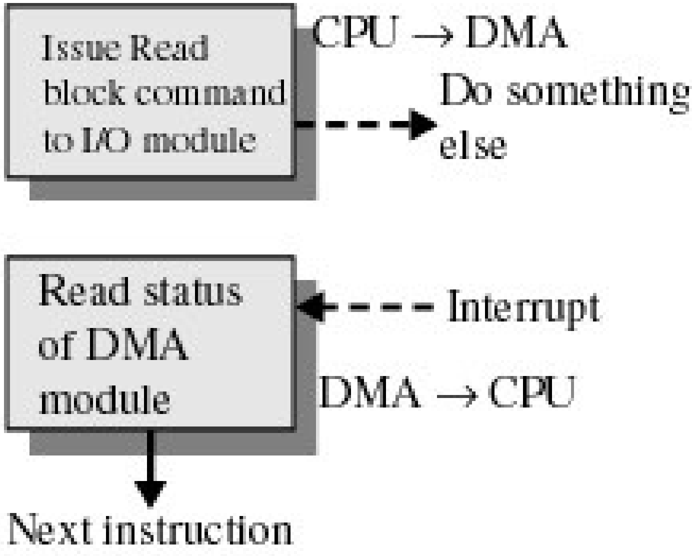


Separate DMA Controller



Direct Memory Access

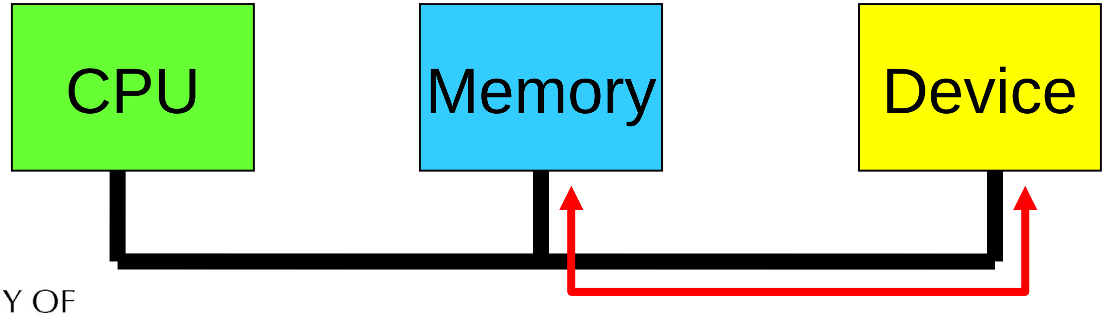
- Transfers a block of data directly to or from memory
- An interrupt is sent when the task is complete
- The processor is only involved at the beginning and end of the transfer


Issue Read block command to I/O module
CPU → DMA
Do something else
Read status of DMA module
Interrupt
DMA → CPU
Next instruction

(c) Direct memory access

DMA Considerations

- ✓ Reduces number of interrupts
  - Less (expensive) context switches or kernel entry-exits
- ✘ Requires contiguous regions (buffers)
  - Copying
  - Some hardware supports “Scatter-gather”
- Synchronous/Asynchronous
- Shared bus must be arbitrated (hardware)
  - CPU cache reduces (but not eliminates) CPU need for bus
- Buggy device can read/write memory



Interrupt Handlers

- **Interrupt handlers**
  - Can execute at (almost) any time
    - Raise (complex) concurrency issues in the kernel
    - Can propagate to userspace (signals, upcalls), causing similar issues
    - Generally structured so I/O operations block until interrupts notify them of completion
      - `kern/dev/lamebus/lhd.c`

Interrupt Handler Example

```
static int
lhd_io(struct device *d,
       struct uio *uio)
{
...
 /* Loop over all the sectors
  * we were asked to do. */
 for (i=0; i<len; i++) {
  /* Wait until nobody else
   * is using the device. */
  P(lh->lh_clear);
  ...
   /* Tell it what sector we want... */
  lhd_wreg(lh, LHD_REG_SECT, sector+i);
  /* and start the operation. */
  lhd_wreg(lh, LHD_REG_STAT, statval);
  /* Now wait until the interrupt
   * handler tells us we're done. */
  P(lh->lh_done);

   /* Get the result value
   * saved by the interrupt handler. */
  result = lh->lh_result;
}
```

INT

SLEEP

```
lhd_iodone(struct lhd_softc *lh, int err)
{
        lh->lh_result = err;
        V(lh->lh_done);
}

void
lhd_irq(void *vlh)
{
 ...
 val = lhd_rdreg(lh, LHD_REG_STAT);

 switch (val & LHD_STATEMASK) {
  case LHD_IDLE:
  case LHD_WORKING:
    break;
  case LHD_OK:
  case LHD_INVSECT:
  case LHD_MEDIA:
   lhd_wreg(lh, LHD_REG_STAT, 0);
   lhd_iodone(lh,
              lhd_code_to_errno(lh, val));
   break;
 }
}
```

Interrupt Handler Steps

- **Save Registers** not already saved by hardware interrupt mechanism
- (Optionally) **set up context** for interrupt service procedure
  - Typically, handler runs in the context of the currently running process
    - No expensive context switch
- **Set up stack** for interrupt service procedure
  - Handler usually runs on the kernel stack of current process
  - Or “nests” if already in kernel mode running on kernel stack
- **Ack/Mask interrupt controller**, re-enable other interrupts
  - Implies potential for interrupt nesting.

Interrupt Handler Steps

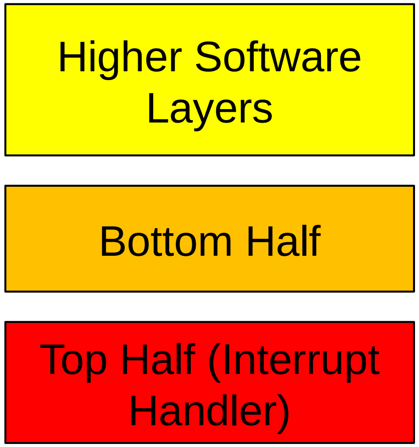
- **Run interrupt service procedure**
  - Acknowledges interrupt at device level
  - Figures out what caused the interrupt
    - Received a network packet, disk read finished, UART transmit queue empty
  - If needed, it signals blocked device driver
- **In some cases, will have woken up a higher priority blocked thread**
  - Choose newly woken thread to schedule next.
  - Set up MMU context for process to run next
  - What if we are nested?
- **Load new/original process' registers**
- **Re-enable interrupt**; Start running the new process

Blocking in Interrupts

- An interrupt generally has no **context** (runs on current kernel stack)
  - Unfair to sleep on interrupted process (deadlock possible)
  - Where to get context for long running operation?
  - What goes into the ready queue?

- What to do?
  - Top and Bottom Half
  - Linux implements with `tasklets` and `workqueues`
  - Generically, in-kernel thread(s) handle long running kernel operations.

Top/Half Bottom Half



Higher Software Layers

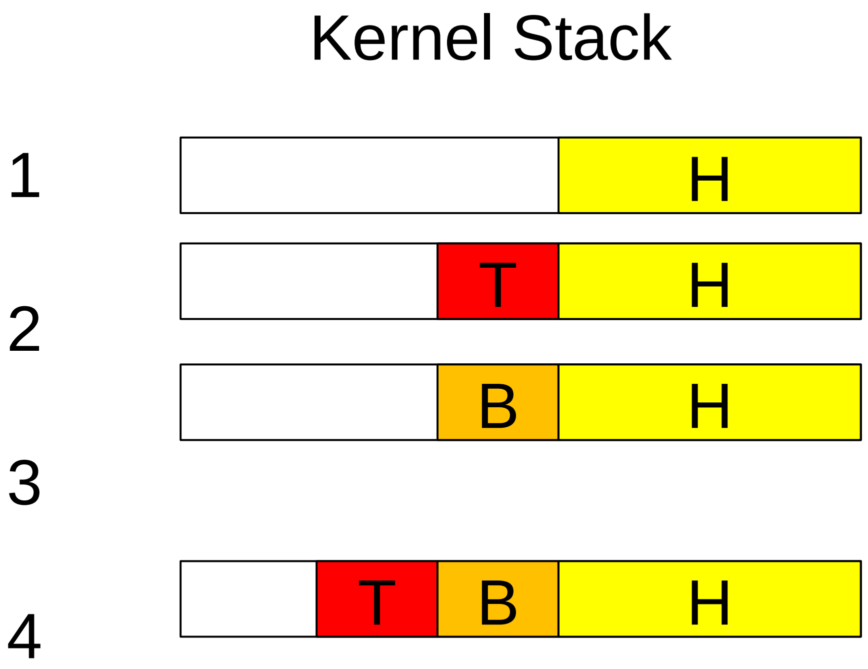
Bottom Half

Top Half (Interrupt Handler)

- Top Half
  - Interrupt handler
  - remains short
- Bottom half
  - Is preemptable by top half (interrupts)
  - performs deferred work (e.g. IP stack processing)
  - Is checked prior to every kernel exit
  - signals blocked processes/threads to continue
- Enables low interrupt latency
- Bottom half can’t block

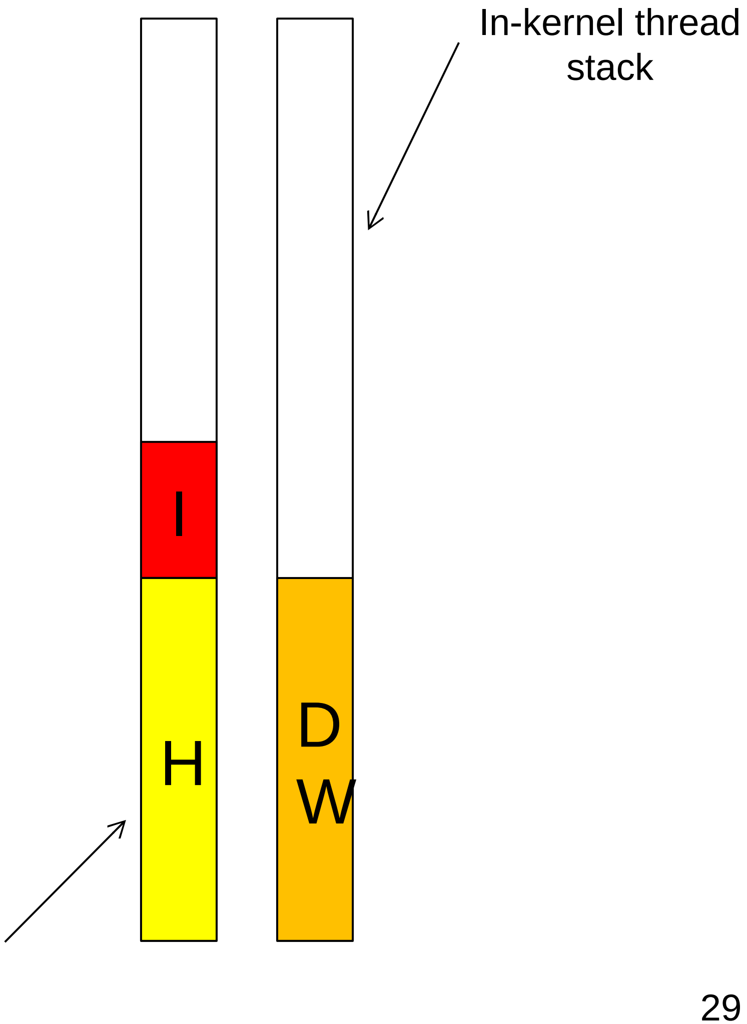
Stack Usage

1. Higher-level software
2. Interrupt processing (interrupts disabled)
3. Deferred processing (interrupt re-enabled)
4. Interrupt while in bottom half

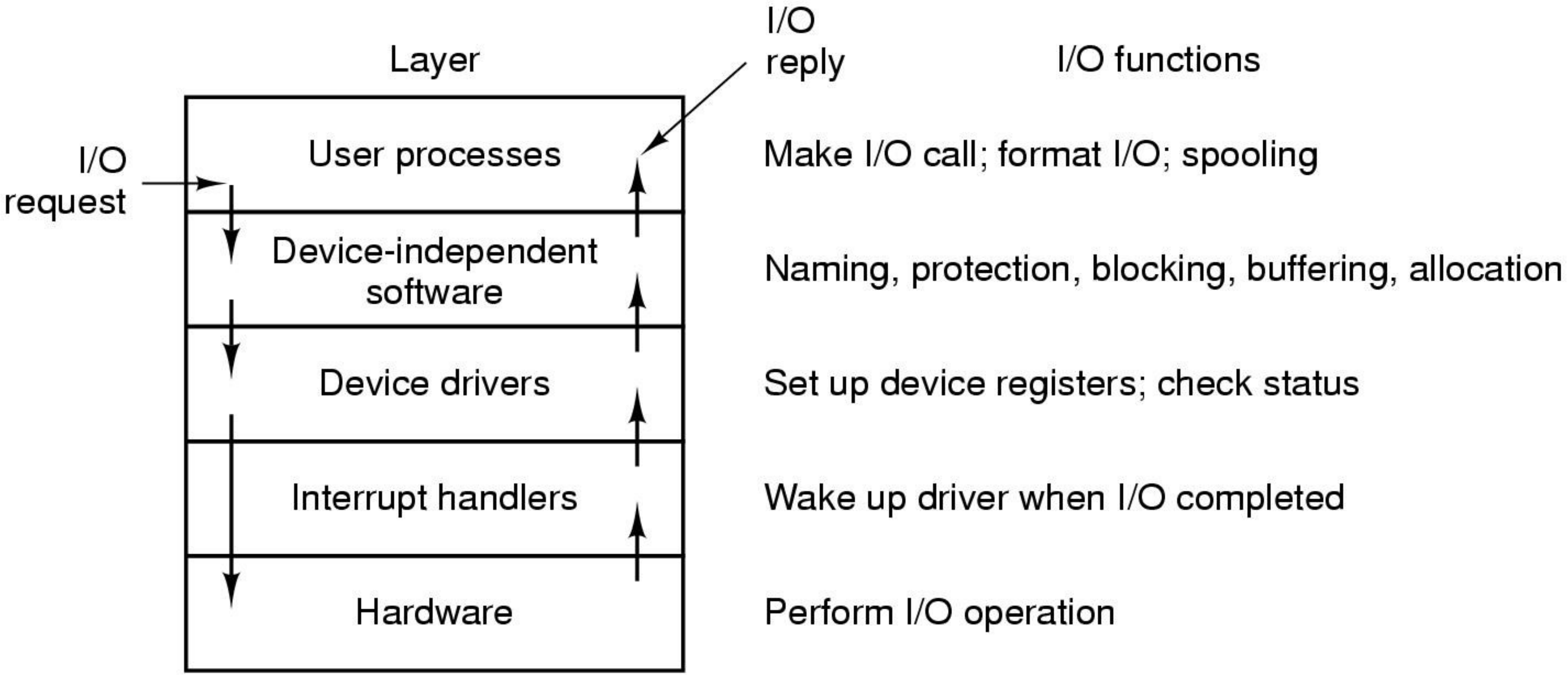


Deferring Work on In-kernel Threads

- Interrupt
  - handler defers work onto in-kernel thread
- In-kernel thread handles deferred work (DW)
  - Scheduled normally
  - Can block
- Both low interrupt latency and blocking operations



I/O Software Summary


Layer
I/O reply
I/O functions
I/O request
User processes
Make I/O call; format I/O; spooling
Device-independent software
Naming, protection, blocking, buffering, allocation
Device drivers
Set up device registers; check status
Interrupt handlers
Wake up driver when I/O completed
Hardware
Perform I/O operation

Layers of the I/O system and the main functions of each layer