

Chapter 3: Input and Output

Input/Output

- Overview of IO function
- Polling (loop awaiting an event)
- Program to read keys from keyboard
- Keyboard driver
- Interrupts alternative to polling
- Device driver structure: top/bottom halves.

Logical Structure of IO

- **Local** peripheral, stream of bytes
 - **Logical IO**- device abstraction, device identifier, simple commands (open, read etc) – user level
 - **Device IO** – operations + data are converted to sequences of IO instns. May use buffering techniques – systems level
 - Scheduling and control – interrupts, interacts with IO module and HW
- **Secondary storage** devices with filesystem
 - Logical IO divided into:
 - Directory management – symbolic file names converted to id, user operations e.g. add, delete
 - File system – logical structure of files (open close read)
 - Physical organisation – conversion of data to actual references on disk.
 - Device IO and then Scheduling&control layers

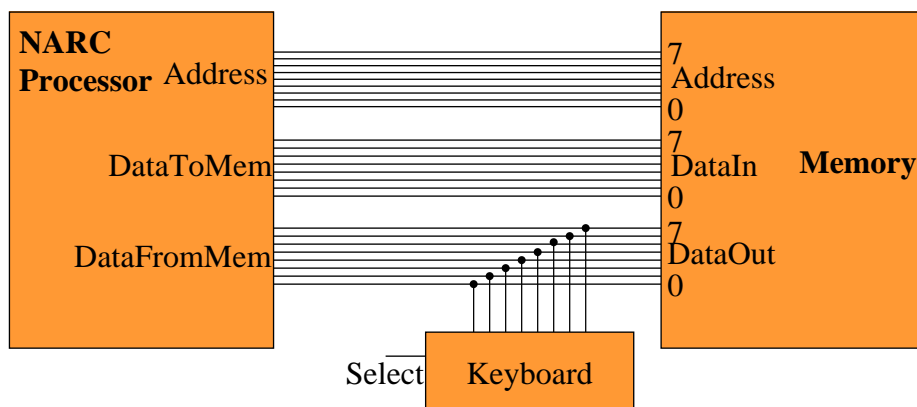
IO function Overview

- **Programmed I/O** – processor issues IO command for processes to IO module
 - Process '**busy waits**' for the operation to be completed before proceeding
- **Interrupt-driven I/O** – processor issues IO, continues to execute subsequent instns
 - interrupted by IO module when finished
 - Process has continued if not need to wait on IO, else it suspend pending interrupt (other job processed)
- **Direct Memory access (DMA)** – a DMA module controls exchange of data between memory and IO module.
 - Processor sends request for data block to DMA and interrupted only after entire block transferred.
- DMA is found in most systems

Memory Mapped IO

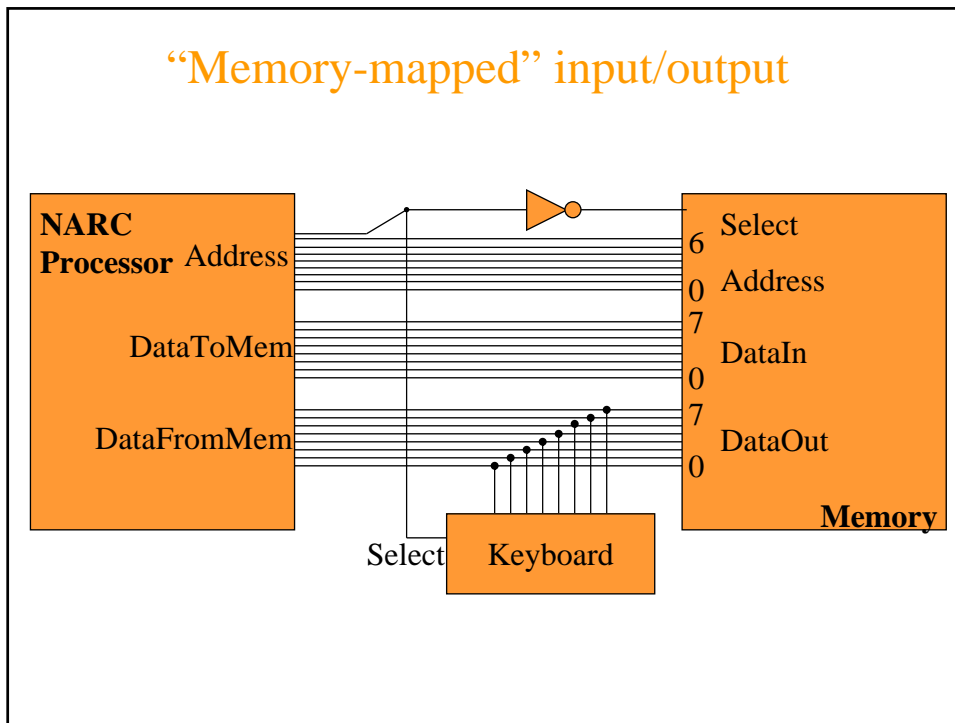
Hitching a ride on the memory bus

- Let's attach the keyboard so it can send data to the processor just like the memory can:

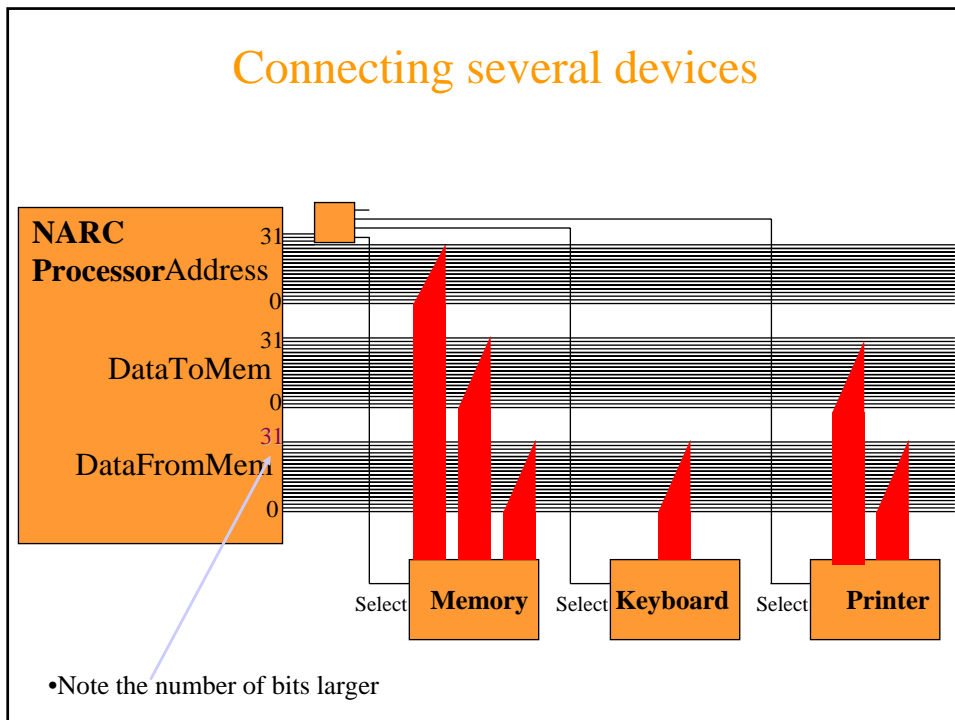


- "Select" signal controls whether the Keyboard signals are sent

“Memory-mapped” input/output



Connecting several devices



Accessing the keyboard

- Need a way to translate bit pattern resulting from key press into standard ASCII representation of the key

```
char scanToAscii(int scan) {
    int i;
    for (i=0; i<32; i++) { // for each 32 bits of scan
        if ((scan >> i) == 1) // shift scan by i places
            return i+'a'; // bit 0 means 'a', etc
    }
    return 0; // if no bit is set, no key pressed
}
```

- Suppose each key 'a', 'b', 'c' etc corresponds to one bit of the 32-bit scan vector input from the keyboard
- Find the index of the first non-zero bit

Bitwise shift right

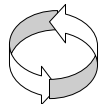
How can we write a program to read the keyboard?

- Need to wait for key to be pressed, then report which key was pressed - try this:

```
#define KBD_PORT_ADDRESS 0x8000000
```

```
char getchar() {
    int scan;
    do {
        scan = *KBD_PORT_ADDRESS;
    } while (scan == 0);
    return scanToAscii(scan);
}
```

Star: use value as a pointer



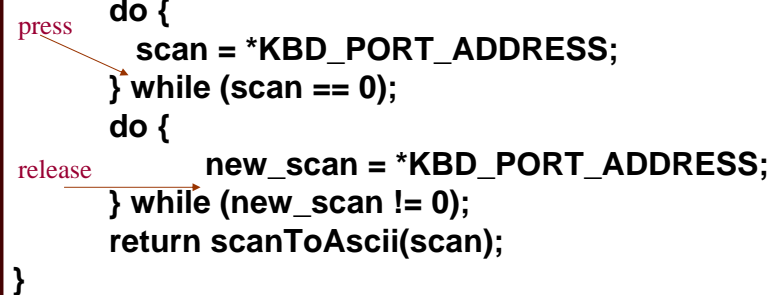
Does this do what we want?

can we write program to read the keyboard correctly?

- Need to wait for key to be pressed, then wait for it to be released, then report which key was pressed - try this:

```
#define KBD_PORT_ADDRESS 0x800000

char getchar() {
    int scan, new_scan;
    do {
        scan = *KBD_PORT_ADDRESS;
    } while (scan == 0);
    do {
        new_scan = *KBD_PORT_ADDRESS;
    } while (new_scan != 0);
    return scanToAscii(scan);
}
```



A function to read a character typed on the keyboard

- Need to wait for key to be released, even if another key has been pressed in the meantime:

```
#define KBD_PORT_ADDRESS 0x800000

char getchar() {
    int scan, changedscan;
    do {
        scan = *KBD_PORT_ADDRESS;
    } while (scan == 0);
    do {
        changedscan = *KBD_PORT_ADDRESS;
    } while (changedscan == scan);
    return scanToAscii(scan);
}
```

Polling has a problem...

- The keyboard driver on the previous slide has a problem. Example:

```
void getword(char* word) {  
    for (int i=0; i<SIZE; ++i) {  
        char ch=getchar();  
        if (isletter(ch))  
            word[i] = ch;  
        else  
            word[i] = '\0';  
        break;  
    }  
    return;  
}
```

```
void nanosoftWord() {  
    while(1) {  
        char word[SIZE];  
        getword(word);  
        if (!isSpelledRight(word))  
            underline(word);  
        addToDocument(word);  
    }  
}
```

What goes wrong?

Polling has a problem...

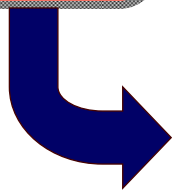
- The keyboard driver on the previous slide has a problem - it misses some keystrokes:

Any keystrokes which take place while we're checking the spelling etc aren't noticed

```
void nanosoftWord() {  
    while(1) {  
        char buf[SIZE];  
        getword(buf);  
        if (!isSpelledRight(buf))  
            underline(buf);  
        addToDocument(buf);  
    }  
}
```


Modifying the fetch-execute cycle

```
PC = 0;  
do {  
  fetch( );  
  PC=PC+1;  
  execute( );  
} forever;
```



```
PC = 0;  
do {  
  fetch( );  
  PC=PC+1;  
  execute( );  
  if (InterruptRequest) {  
    save(PC);  
    PC = ipc;  
  }  
} forever;
```

Address of interrupt
handler function

Not finished
yet... why?

Enabling and disabling interrupts

```
PC = 0;  
do {  
  fetch( );  
  PC=PC+1;  
  execute( );  
  if (InterruptRequest &&  
      InterruptEnabled) {  
    InterruptEnabled = false;  
    save(PC);  
    PC = ipc;  
  }  
} forever;
```

Block further
interrupts;
re-enable
after dealing
with this one

When do
interrupts
get re-enabled?

```

PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;

```

Saving the PC during an interrupt, and restoring it on return

Save the current PC somewhere (so we can return later), and branch to the address of the interrupt handler

```

PC = 0;
do {
  fetch( );
  PC=PC+1;
  execute( );
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[1];
  }
} forever;

```

The interrupt handler

```

void InterruptHandler() {
  saveProcessorState();
  char ch= *KBD_PORT_ADDRESS;
  KbdBuffer.add(scanToAscii(ch));
  restoreProcessorState();
  InterruptEnabled = true;
  PC=Mem[0];
}

```

Is this always going to work?

What if another interrupt occurs right away?

```

PC = 0;
do {
  fetch();
  PC=PC+1;
  execute();
  if (InterruptRequest &&
      InterruptEnabled) {
    InterruptEnabled = false;
    Mem[0] = PC;
    PC = Mem[0];
  }
} forever;

```

```

void InterruptHandler() {
  saveProcessorState();
  char ch= *KBD_PORT_ADDRESS;
  KbdBuffer.add(scanToAscii(ch));
  restoreProcessorState();
  rti; // restore PC from Mem[0] and
      // re-enable interrupts
}

```

We need a special instruction to do this indivisibly

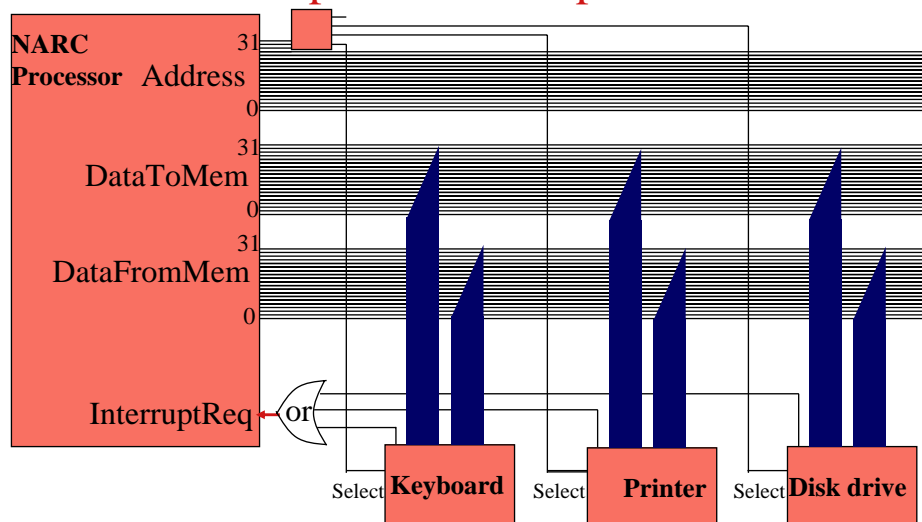
Basic interrupts - summary

- We introduced **interrupts** to avoid missing the next keystroke while processing the previous ones
- We use **interrupts instead of “polling”**, so the processor’s time is not wasted repeatedly testing to see if a key has been pressed
- After processing each instruction, the processor checks to see **if an interrupt can be requested**
- If so, it **saves the PC and jumps to the interrupt handler**

Basic interrupts - summary, continued

- In the interrupt handler, the **registers are saved**, and the device is serviced
- Finally the registers are restored, interrupts are re-enabled, and the PC is restored so that the **interrupted computation continues**
- Some care is needed to make sure that:
 - If **another interrupt arises** (perhaps from another device) simultaneously or shortly afterwards, it is handled properly
 - If another interrupt occurs while a preceding one is being handled, the **saved PC is not overwritten**

Interrupts with multiple devices



- Interrupt handler writes to interrupting device's control register to clear its request.

In real life

- In most modern processors, PC is saved on the stack - so saved PC is not overwritten if another interrupt arrived before old PC has been restored
- This can be useful if some interrupts are more urgent than others - so a high-priority interrupt can interrupt a low-priority interrupt's handler
- It's still very hard to prevent interrupts being missed if they arrive from the same device in very quick succession
- Most interrupts signal completion of a request made by the processor - so the device won't raise another interrupt until the processor requests it to do something again
- But not all - e.g. incoming network data

```
char getchar() {  
    while (KbdBuffer.isEmpty())  
        wait; // maybe let another process run  
    return KbdBuffer.remove();  
}
```

“Top
half”

Device
driver
structure

“Bottom
half”

```
void InterruptHandler() {  
    saveProcessorState();  
    switch (InterruptSource()) {  
    case  
        KBD:  
        char ch= *KBD_PORT_ADDRESS;  
        *KBD_PORT_ADDRESS=0; // acknowledge interrupt  
        if (!KbdBuffer.isFull())  
            KbdBuffer.add(scanToAscii(ch));  
            break;  
        default: // code to handle other devices  
    }  
    restoreProcessorState();  
    rti;  
}
```

Device driver structure - top-half, bottom-half

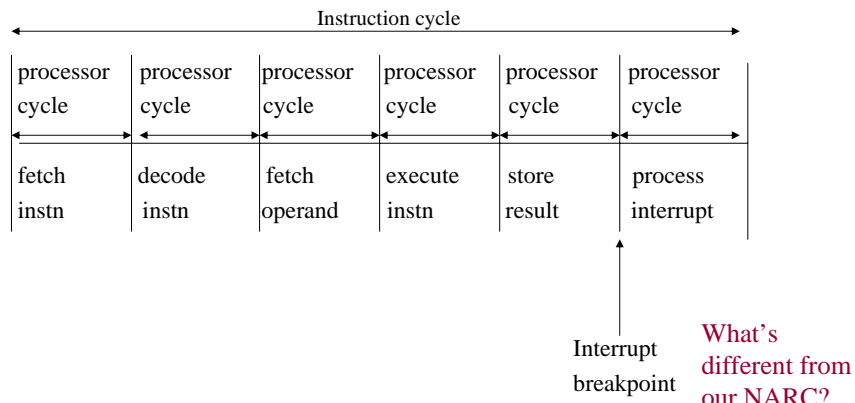
- Top half of device driver is called by client process. It initiates device operation
 - The two halves interact via a queue
 - With an input device, the bottom half adds items to the queue, the top half removes them
 - With an output device, the top half adds items to be output, the bottom half removes them
- Bottom half is initiated by interrupt signalling device completion
 - Bottom half reinitiates device operation if there is more work to do
 - **The top half can allow other processes to run while it is waiting**

Direct Memory Access

- DMA module is a **processor mimic**. Transfers data blocks directly from/to memory
- For DMA to use bus it *steals a cycle* – interrupts the processor (forces it to suspend temporarily)
- When processor wishes to read/write a block of data, it issues a command to the DMA module by sending:
 - Request is **read or write** as read and write have different control lines between processor and DMA module
 - **Address of IO device**, communicated on data lines
 - Starting **location in memory** to read from/write to (on data lines stored by DMA's address register)
 - **Number of words** read/written stored in data count register
- Processor continues with other work
- When transfer complete sends interrupt to processor

Instruction Cycle and DMA

- Processor is suspended just before it needs to use the bus
- DMA transfers one word returning control to processor
- NOT AN INTERRUPT (interrupts mean processor save state) i.e. its **pause** for a cycle

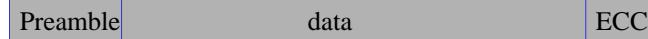


The Hard Drive



The Hard Drive

- Consists of aluminium, alloy or glass platters (< 6cm for notebooks) coated with magnetizable metal oxide.
- It is blank
- Low-level format → software → tracks, sectors and gaps between sector



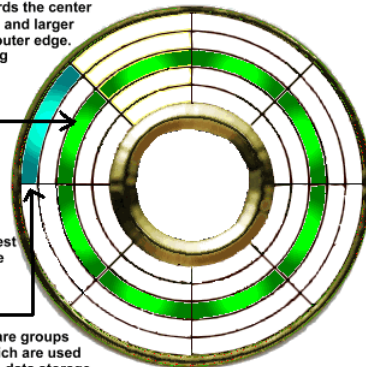
- Preamble indicates start of sector, contains cylinder and sector numbers etc.
- Size of data is determined by the software formatter → 512-bytes.
- ECC = Error Checking Corr. is used for error recovery.
- Many hard disks have spare sectors in the event of failure.
- Position of sector 0 on each track is offset from the previous track when formatted
- Offset → **Cylinder Skew**

Cylinder Skew

TRACK - the concentric circles on the platter. The circles are smaller towards the center of the platter, and larger towards the outer edge. The green ring is one track.

SECTOR - is one section of a track. It is colored blue in the diagram. It is the smallest unit of storage on the platter.

CLUSTERS - are groups of sectors which are used to allocate the data storage area.



- Request needs 18 sectors.
- Starts at 0 innermost part of disk
- 1st 16 sectors read with one disk rotation
- Seek required to get to 17th sector
- By that time disk rotated, so we await a further rotation = expensive waste
- How could we overcome this?
 - Change the Drive Geometry!

Drive Geometry

- Amount of cylinder skew depends on the drive geo.
- e.g. 10,000 rpm drive rotates in 6 msec
 - If a track contains 300 sectors, a new sector passes under the head every 20 μ sec
 - If Track to track seek time is 800 μ sec, 40 sectors pass during a seek
 - Therefore the cylinder skew = 40
 - i.e. if track 1 is at 0 the next track is in 41st sector and a whole rotation is not necessary.

Disk Capacity

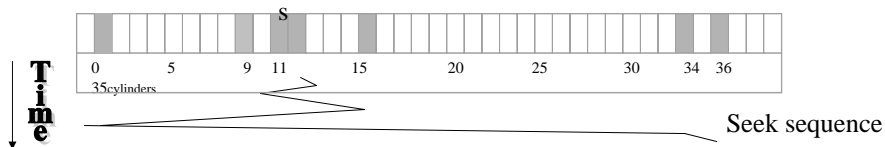
- Commercial interest in describing Disk Capacity
- E.g. Airgate Disk Drives Ltd. Ship a drive whose unformatted capacity is 20×10^9 bytes. What would they call it? •20 GB disk
- After formatting only 17.2×10^9 bytes are available for data, and the OS reports as 16GB (cos $1 \text{GB} = 2^{30}$ not 10^9)
- Affects performance \rightarrow 10,000 rpm disk has 300 sectors/track of 512 bytes each, 6 msec to read 153,000 bytes on a track for a data rates 24.4MB/sec.
- Can't go faster even if SCSI interface with 80 or 160 MB/sec!
- Therefore need a buffer in the controller, while next sector being read buffer is being transferred to memory. Controller must be able to buffer an entire track
- After low-level partitioning \rightarrow disk is partitioned (drives C:,D: etc). E.g. Pentium sector 0 contains the **master boot record** holding partition table.

Disk Arm Scheduling Algorithms

- Time to read/write depends on:
 - Seek Time (move arm to cylinder)
 - Rotational Delay (sector under head)
 - Transfer time
 - Which takes the longest time? Seek!
- Easy Schedule : (FCFS) (alternative = Shortest Seek First (SSF))
 - E.g. Airgate Disk has 40 cylinders.
 - Read block on cylinder 11
 - While seek on cylinder 11, new requests for cylinders 1,36,16,34,9,12 arrive
 - A table (linked list) keeps pending requests

FCFS & SSF

- When the current request (cylinder 11) finished:
 - FCFS would pick up 1 then 36 etc which means arm motions of 10,35, 20, 18, 25, 3 for 111 cylinders



- SSF gets closest request to minimise seek.
- 1,3,7, 15,33,2 for 61 cylinders i.e. cutting arm motion $\frac{1}{2}$
- Problem with SSF is
 - that if the rate of requests increases the disk arm can end up hovering over the middle.
- Elevator algorithm overcomes this: goes up and down the list in one direction (like a lift!) trading off, what? Fairness and response time

RAID

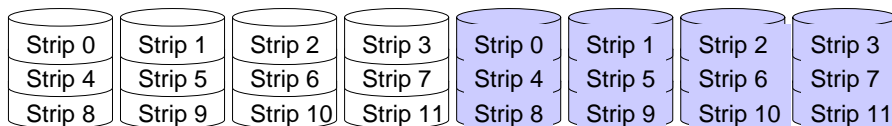
- Technology to try and accommodate for increases in CPU speed over disk speed, uses parallelism (suggested by Patterson in 1988)
- **Redundant Array of Inexpensive Disks** (which industry changed to Independent!)
- Appears to the OS as a single disk (virtual disk)
- RAID (which is described in levels 0 through to 5) is effectively a method allowing the data to be spread seamlessly over a number of disks.
- Level 0: Virtual disk is divided into strips of k sectors arranged over the disks in a round-robin fashion → striping.
- OK if we have 4 disks on our RAID system and we want data stored on 3 *consecutive* strips is required and it takes 10 μ sec to read a strip, how long would it take to read all 3?

10 μ sec



True RAID

- Level 0 has fault tolerance problems (i.e. what happens when a disk fails) *therefore not true RAID*
- Level 1 = True RAID → full replication! I.e redundancy



- Write goes to two disks, read from either copy (higher speed read, slower write)
- Subsequent RAID levels look at word or strip granularity and use parity checking (e.g. hamming code) to check if a word/strip is wrong.

Clocks

- Clocks tell time of day and stop CPU hogging (in multiprogrammed computers)
- Clock is a device! (has hardware and software)
- Hardware
 - Old machines it was tied to 110 or 220 volt power line
 - Cause an interrupt on every voltage cycle (50 or 60 Hz)
 - New: crystal oscillator, a counter and holding register
 - Crystal generates v. accurate signal 100s of MHz → 1000 MHz or more
 - Signal starts counter, when down to 0 then interrupt!
 - Interrupt Frequency (count down and restart=*clock tick*)
 - Programmable clock: the software does restart
 - interrupt freq controlled by software

Clock Software

- Clock Driver
 - Maintain time of day
 - Prevent processes from running longer than allowed
 - Accounting for CPU usage
 - Alarm system call from user process
 - Watchdog timers for system parts (stop disk rotation if not used)
 - Profiling, monitoring and statistics gathering
 - » I.e. what systems programs relate to this?
Stuff in /proc and top etc

Chapter summary

- Device can be accessed via **processor's memory bus**, either using ordinary load/store instructions, or special in/out instructions
- **Polling**: repeated probing of device to see if ready
- **Interrupt**: signal from device causes processor to jump to handler routine; when completed, returns to what it was doing
- Further interrupts must be **disabled** until handler has finished (or at least saved state)
- A **device driver** consists of a function called by the **client** process (top half), and an **interrupt handler** (bottom half) - normally connected by a queue
- We covered disk capacity and performance looking at RAID
- Clocks as a device – *used in multi-processing !!!*