The Lower Levels of the Memory Hierarchy: Storage Systems

Overview

We use the term ***storage system*** to describe persistent memory, meaning memory devices that retain information when power is not applied, such as magnetic disks, magnetic tapes, optical disks of various kinds, and flash memory devices. The concepts of input and output have traditionally been used to refer to the transfer of data from non-persistent internal stores, i.e., memory, and storage systems or devices used to receive data from transient sources (keyboards, networks) or to display data on a transient device (monitors and other visual displays). Although in theory the transfer of data between levels of a hierarchy can be treated in a uniform way, in practice, there are significantly different issues when data has to migrate to or from devices or be stored on them.

Because we use storage systems to ***preserve*** our data for us, it is of the utmost importance that we can ***depend*** upon them to do so. This concept of ***dependability*** will be made precise shortly.

The standard for dependability is higher for storage systems than it is for computation. It is one thing for a processor to hang or even crash, but another if a hard disk fails. A processor crash is usually just an inconvenience but a disk failure might mean irreplaceable data loss. Therefore, storage systems place greater emphasis on:

- dependability
- cost
- expandability

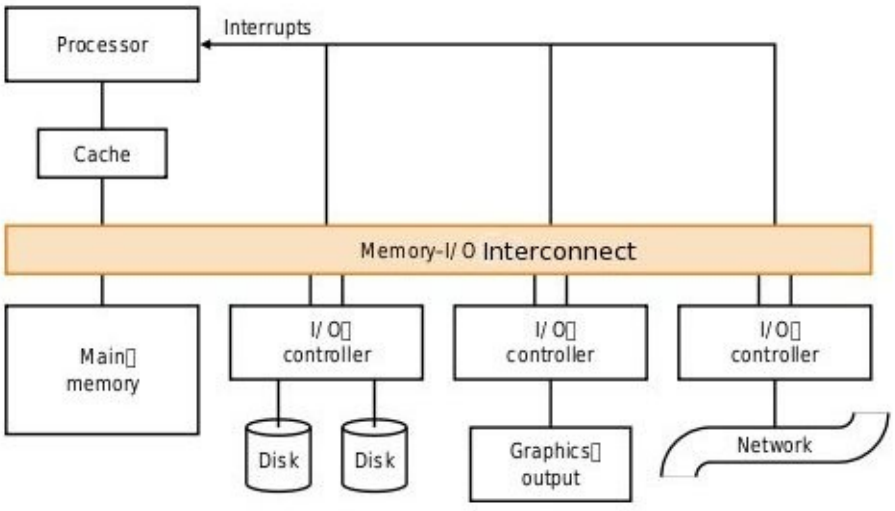


Figure 1: Typical collection of storage and I/O devices

than they do on performance. Performance is more complex to analyze for storage and I/O devices than it is for processors. It depends on several factors, including:

- device physical characteristics, such as mechanical components, electronic components
- type of connection between the device and internal system
- location within the memory hierarchy
- operating system interface to the device

Assessing Performance

Should performance be measured by

- system throughput (jobs completed per unit time), or
- response time?

Throughput can be measured by bandwidth, either

(1) number of completed I/O operations per unit time, or
(2) number of bytes transferred per unit time

Examples.

1. Transaction processing systems such as on-line commerce applications and on-line banking systems have to handle a large number of small transactions, so (1) is a better measure than (2).
2. A system for delivering streaming video, on the other hand, is better measured by the volume of bytes transferred per unit time (2) than by the number of completed transactions.

Throughput is not always the best measure. Sometimes response time is more important. Often not just the mean response time, but the standard deviation of response times (indicates how reliable the system is). For single-user workstations, response time is typically the most important performance measure.

Sometimes both are equally important:

3. ATM networks, file servers, and web servers must have short response times for their customers and must also process large amounts of data quickly.

Dependability, Reliability, and Availability

Dependability is the quality of a system such that reliance can be placed on its service. The service delivered by a system is its observed actual behavior as perceived by another system that interacts with it. The other system may be a machine, a person, or both.

Dependability is therefore dependent on a reference point, since it may be different to different observers. To remove this subjectivity, there needs to be a reference standard, called a ***system specification***, that specifies expected behavior.

When such a specification exists, one can define:

1. ***Service accomplishment*** -- the state during which service is being delivered (meets specification).
2. ***Service interruption*** -- the state in which the delivered service is different from the specified service. This could mean no service at all, or service that fails to meet the specification.

A ***failure*** is a transition from state 1 to state 2. A transition from state 2 to state 1 is called a ***restoration***.

Failures can be permanent or intermittent. Intermittent failures are harder to diagnose.

Operator	Software	Hardware	System	Year data collected
42%	25%	18%	Datacenter (Tandem)	1985
15%	55%	14%	Datacenter (Tandem)	1989
18%	44%	39%	Datacenter (DEC VAX)	1985
50%	20%	30%	Datacenter (DEC VAX)	1993
50%	14%	19%	U.S. public telephone network	1996
54%	7%	30%	U.S. public telephone network	2000
60%	25%	15%	Internet services	2002

Figure 2: Summary of studies of causes of failure, from Patterson/Hennessy

Reliability is a measure of continuous service accomplishment. Equivalently, it can be measured by the ***mean time to failure*** (***MTTF***). (The amount of time it is available without interruption, on average.)

The MTTF is usually an unintuitive measure of the reliability of a device. For example, today's disk drives have typical MTTFs larger than 1,000,000 hours. This translates to more than 114 years. Does this mean that if you purchased such a drive it would not fail in your lifetime? No, of course not, because the MTTF is a statistic, not an absolute guarantee. For this reason, the inverse of the MTTF is often used instead. The inverse is the number of failures per unit time that can be expected. The typical unit of time is one year – the number of failures that can be expected in a single year – and it is called the ***annual failure rate***, or ***AFR***.

The AFR is the average number of failures per year, i.e., the average number of devices, that would fail in a year. For example, if the MTTF is 3 months, then the AFR is 4 times per year. If the MTTF is 5 years, then the AFR would be 1/5th of a year, meaning that one fifth of a device would fail in a year, which makes no sense! The way it is interpreted is that it is the fraction of devices that fail in a year. If the MTTF is 5 years, and we owned 100 such devices, then we would expect 1/5th of them, or 20, to fail in a year. Since there are 8760 hours in one year, on average we would expect a failure every 438 hours, assuming the devices operate continuously.

In fact, disk drive manufacturers typically report the AFR for their devices rather than the MTTF, and determine the MTTF of a new disk drive from its AFR, which is estimated based on accelerated life and stress tests or based on field data from earlier products[1]. The MTTF is then estimated by dividing the number of available hours per year by the AFR. Usually the AFR is less than 1%.

Example. If a disk drive has a 1,500,000 MTTF, then, recalling that there are 8760 hours in a year, its AFR is 8760/1500000= .00584. If a company owns 100,000 such disk drives, then we would expect 0.00584*100000=584 disk failures each year, which is more than one disk failing every day.

[1] G. Cole. Estimating drive reliability in desktop computers and consumer electronics systems. TP-338.1. Seagate. 2000.

Service interruption is measured by how long a system is unavailable, which is characterized by the ***mean time to repair*** (***MTTR***). Mean time to repair is the average amount of time that it takes for a device to be restored to the available state from the unavailable state. Sometimes this can be just minutes or hours, but sometimes it can be days, depending on many factors.

The MTTF alone does not characterize the availability of a device, because if the interval from the moment of failure to the moment at which it is restored is long, the fraction of available time is diminished. A more comprehensive characterization is the sum of the MTTF and the MTTR and is called the ***mean time between failures*** (***MTBF***). You can think of it as the time to the next failure plus the time it takes to repair that failure and bring the system back to the available state.

Availability is defined as

Availability = MTTF/ (MTTF + MTTR) = MTTF/MTBF

Then 0 <= Availability <= 1.0. It is the fraction of time that a system is achieving service accomplishment.

Example

A system's performance is observed over the course of 185 days and the following data was recorded:

Day	10	12	92	95	140	145	180	185
Event	Failed	Restarted	Failed	Restarted	Failed	Restarted	Failed	Restarted

after which it stayed up. Assuming that the events occurred at the ends of the given days, the MTTF is

(10 + (92-12) + (140-95 + (180-145)) /4 = (10+80+45+35)/4 = 42.5

The MTTR is (2+3+5+5)/4 = 3.75

Availability is 42.5/(42.5+3.75) = 0.92.

When using the formula

MTTF/ (MTTF + MTTR)

the number of data points in the MTTF and MTTR should be the same, otherwise the formula is less accurate than using the average of the mean times between failures obtained directly from the in service times themselves. The degree of inaccuracy will diminish though as the number of data points becomes very large (in the hundreds to thousands).

To improve MTTF, either one improves the quality of the components or makes the entire system more ***fault tolerant***. A system is fault tolerant if a failure does not lead to unavailability.

A ***fault*** is a failure of a component.

Hence MTTF can be improved by either:

1. ***Fault avoidance*** -- make the components better so that they do not fail as frequently.
2. ***Fault tolerance*** -- use redundancy to allow service to continue to meet the specification despite the occurrence of faults. RAID is an example of building fault tolerance into a storage system.

3. ***Fault forecasting*** -- predicting the presence and creation of faults, both hardware and software, to allow for replacing it before the failure occurs.

Fault Tolerance in the Memory Hierarchy

> *I cnduo't bvleiee taht I culod aulaclty uesdtannrd waht I was rdnaieg. Unisg the icndeblire pweor of the hmuan mnid, aocdcrnig to rseecrah at Cmabrigde Uinervtisy, it dseno't mttaer in waht oderr the lterets in a wrod are, the olny irpoamtnt tihng is taht the frsit and lsat ltteer be in the rhgit pclae. The rset can be a taotl mses and you can sitll raed it whoutit a pboerlm. Tihs is bucseae the huamn mnid deos not raed ervey ltteer by istlef, but the wrod as a wlohe. Aaznmig, huh? Yaeh and I awlyas tghhuot slelinpg was ipmorantt! See if yuor fdreins can raed tihs too.*[2]

People are able to read this text because they do not read it letter by letter. We tolerate small "bit errors" in text. Computers do not behave this way. Instead, they use redundancy schemes to detect and correct errors at all levels of the memory hierarchy, including static and dynamic RAM, disk devices, and network interfaces.

Bits are sometimes flipped in transmission, perhaps on a bus, or when being written to memory, or being transferred to disk, or when written to disk. It can happen anywhere they are "in flight." For example, the bit sequence 01101001 might get garbled when it is written to memory and become 01111001. This is a single bit flip.

In 1968, Richard Hamming invented a redundancy scheme for memory based on the idea of parity. (He received the Turing Award for this in 1968.)

Definition: A bit string has ***odd parity*** if the number of 1's in the string is odd. A bit string has ***even parity*** if the number of 1's in the string is even.

A simple redundancy scheme based on parity is to count the 1-bits in the word when it is written to memory. An extra bit called a parity bit is stored with the word in memory. If the word has odd parity, the parity bit is set to 1, otherwise it is set to 0. The sum of the parity of the word and the parity bit is always even. When the word is read, the parity bit is read also. If the parity of the word and the parity bit is not even, it implies that an odd number of bits were flipped. Since the probability that three or more bits were flipped is close to zero, it really implies that a single bit was flipped. If two bits get flipped this scheme does not detect it, because the parity will not change. (Make sure you convince yourself of this.) Because bit flips have very low probability in general, the occurrence of two or more bit flips in a word is a tolerable error event.

This is a form of ***error detection code*** (***EDC***). It can detect the presence of an error but not its location, and hence it cannot be used to correct the error. Thus, by itself it is not a method of fault tolerance. For fault toleramce we need an ***error correction code*** (***ECC***).

Definition. The ***Hamming distance*** between two strings is the least number of bits that must be changed in one string to make it identical to the other string.

Examples. Let d(x,y) be the Hamming distance between bit strings x and y.

d(010101, 000000) = 3

[2] http://www.ecenglish.com/learnenglish/lessons/can-you-read

d(11001100, 00111100) = 4

d(01101, 01001) = 1

Hamming distance is a useful measure and it is the basis of various error correcting codes. The basic idea is that there are correct bit patterns and incorrect bit patterns. Incorrect bit patterns correspond to bit strings that have errors, and correct bit patterns are those that are guaranteed to be error free.

Suppose that the Hamming distance between any pair of correct bit patterns is 3. Then it takes three bit flips to make any correct pattern look like another correct pattern. Since three bit flips are highly improbable, if one could develop a mapping of data into a distance 3 code, then it would lead to a method of error correction.

Example.

If we have data that is just two bits long, there are four possible patterns: 00, 01, 10, and 11. Suppose that, to transmit a 2-bit data value xy, we send the 5-bit code xypxy, where p is a parity bit for the string xy. For 2-bit patterns 00, 01, 10, and 11, the code words respectively are 00000, 01101, 10110, 11011. You can check that the Hamming distance between any pair of these code words is at least 3. If a bit pattern is received that is not one of these code words, then it represents an error. The correction algorithm chooses a code word whose Hamming distance from the incorrect pattern is least.

For example if the pattern 10101 is received, since

d(10101, 00000) = 3

d(10101, 01101) = 2

d(10101, 10110) = 2

d(10101, 11011) = 3

either 01101 or 10110 could be chosen, since they have the same Hamming distance to the received word.

Hamming's ECC for memory that detects and corrects a single bit error is as follows. Prior to transmitting or writing the data, we do the following:

1. Start numbering bits with 1 as the leftmost bit (not 0, and not rightmost)
2. Write the bit positions as binary numerals: 1, 10, 11, 100, 101, ...
3. All bit positions that are powers of 2 are marked as parity bits. Thus, you mark 1, 2, 4, 8, 16 and so on as parity bits. These are the binary numerals with a single 1-bit.
4. The unmarked bit positions are for the actual data bits. These are 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, 18, ...
5. The position of each parity bit determines for which data bits it is the parity bit according to the rules:
    1. Bit 1 checks all bits whose position number (its address in the string) has a least significant bit of 1: 1, 3, 5, 7, 9, 11, and so on/ I.e., all odd numbers.

2. Bit 2 checks all bits whose position number has a second-least significant bit of 1 (all numbers that end in 10 or 11 in the binary representation): 2, 3, 6, 7, 10, 11, 14, 15, ...
3. Bit 4 checks all bits whose position number has a third-least significant bit of 1 (all numbers whose binary representation ends in one of 100, 101, 110, 111): 4, 5, 6, 7, 12, 13, 14, 15, 20, 21, 22, 23,, ...
4. Bit 8 checks all bits whose position number has a fourth-least significant bit of 1: 8 through 15, 24 through 31, 40 through 47, and so on.
5. In general each parity bit covers all bits where the bitwise AND of the parity position and the bit position is non-zero.

6. Set all of the parity bits to create even parity for each group that they check.

In this scheme, every data bit is checked by at least two parity bits. This is how the location of the error can be detected and corrected. The following table illustrates coverage for a byte of data.

Bit Position		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverage	p1	x		x		x		x		x		x	
	p2		x	x			x	x			x	x	
	p3				x	x	x	x					x
	p4								x	x	x	x	x

The table shows that the 8 data bits are in positions 3,5,6,7,9,10,11,12. If you scan down each of these columns, you see at least two x's. Each x marks which data bits the corresponding parity bit in that row checks. So d1 is checked by p1 and p2, d2 by p1 and p3, and so on.

The error correction works as follows. When the transmitted value is received or it is time to read the data out of memory, the parity bits are checked against the bits they cover, including themselves. Before transmission each group of bits had even parity, so if there are no errors, each group should still have even parity. Therefore, for each parity bit, the parity of its group must be computed and the parity bit set to 0 if the sum is even and 1 if odd. If any turned odd, this indicates an error that can be corrected.

In the above example, if the computed parities of the 4 parity bit sequence (p8,p4,p2,p1) remains even, which means that the sequence is (0,0,0,0), then there is no error. Otherwise the values of the recalculated parity bits among (p8,p4,p2,p1) are used to find the position of the error. The decimal value of the binary number represented by the sequence (p8,p4,p2,p1) is the position of the erroneous bit. For example, if the sequence is $1001_2 = 9_{10}$ then bit 9 (d5) is an error and must be corrected. Why this works is not justified here, but roughly, each bit that is a 1 is a parity bit whose group contains the erroneous bit, and each group consists of positions that have a 1-bit in their binary numbers for the given parity bit.

Hamming also developed a single error correction, double error detection code (SEC/DED).

Modern memory technology uses various ECC's in both SRAM and DRAM. Because ECC slows down a memory, there is a trade-off between reliability and performance. In spite of this, many servers use some form of ECC memory chips (some use SEC/DED). In these chips, the entire row of data is checked at once. For example, an 8-byte memory block in a single 64-bit wide row would be extended with 8 bits to form a 72-bit wide memory. Some motherboards do not support ECC memory modules and some do. You can check their specifications to determine this.

Example

Suppose the byte 10110110 is transmitted and on the way it becomes 10010110, i.e., data bit 3, counting from the left starting at 1, is flipped. We show how the Hamming single error ECC works.

The 8-bit sequence needs 4 parity bits ate positions 1, 2, 4, and 8. The actual data is therefore

_ _ 1_011_0110.

1. Position 1 checks positions 1,3,5,7,9,11 which are in bold: _ _ **1**_ **0**1**1** _ **0**1**1**0. There are three 1's so it is odd parity and position 1 is a 1-bit.
2. Position 2 checks positions 2,3,6,7,10,11, which are in bold: _ _ **1** _ 0**11** _ 0**11**0. There are five 1's, so it is odd parity and position 2 is set to a 1-bit.
3. Position 4 checks positions 4,5,6,7,12 which are in bold: _ _ 1_ **011** _ 011**0**. There are two 1's, so it is even parity and position 4 is set to a 0-bit.
4. Position 8 checks positions 8,9,10,11,12 which are in bold: _ _ 1_ 011 _ **0110**. There are two 1's, so it is even parity and position 4 is set to a 0-bit.

The 12-bit sequence that is transmitted is therefore 111001100110. On receipt it is 101000100110. The error detection code checks the parity groups:

1. The parity of the group for bit 1 shown in bold **1**1**1**0**0**0**1**0**0**1**1**0 is 0. Parity bit 1 stays 0.
2. The parity of the group for bit 2 shown in bold 1**11**00**0**10**0**1**1**0 is 1. Parity bit 2 is set to 1.
3. The parity of the group for bit 4 shown in bold 111**000**100110 is 1. Parity bit 4 is set to 1.
4. The parity of the group for bit 8 shown in bold 1110001**0011**0 is 0. Parity bit 8 stays 0.

The sequence (p8,p4,p2,p1) is 0110 = 6_{10} so bit 6, which is data bit 3, is corrected: 10100**1**100110 and the extracted byte is 10110110.

Disk Storage

Magnetic Disks

All magnetic disks have the following properties in common:

- One or more rotating platters (they rotate in unison)
- Magnetic coated surface
- Movable read/write head

- Nonvolatility
- Rotation speed (also called spindle speed these days) from 5400 to 15000 RPM

There is some important terminology to remember regarding the structure of a magnetic disk drive:

- A ***platter*** is one disk of a set of one or more coaxial disks.
- A ***surface*** is one of the two surfaces of any platter. Each surface is divided into a set of concentric circles, called tracks.
- A ***track*** is one of the concentric rings on a surface on which the data is recorded.
- A ***sector*** is the consecutive sequence of bits on a track within a pre-specified arc of the circle.
- A ***cylinder*** is the set of all tracks under the read/write heads, on all surfaces, at a given time

Tracks are not necessarily divided into the same number of sectors. Inner tracks may have fewer sectors than outer tracks. The idea is to keep the bits uniformly spaced in each track. This is called ***zone bit recording*** (ZBR). Early hard disk drives and modern CDs and DVDs write data at a constant number of bits per second. They do this by varying the speed of the drive depending on which track is being accessed. The result is that all tracks have the same amount of data (density) per track. Modern hard drives use ZBR, increasing the write speed from the inner to the outer zone and thereby storing more data per track in the outer zones.

Performance Costs

To perform a read or a write on a disk, three steps must be taken:

1. The read/write heads must be moved in or out until they are positioned over the appropriate track. This is called ***seeking***.
2. The head then waits until the right sector has moved under the head.
3. The data is transferred to or from the disk.

Each of these steps has an associated cost in terms of time:

- ***Seek time*** The amount of time for the heads to move to the right track
- ***Rotational latency*** The time spent waiting for the right sector to rotate under the heads. (Also called ***rotational delay***.)
- ***Transfer time*** The time to transfer a block of bits, which depends upon sector size, rotation speed, and the recording density of the track.

  The ***external data transfer rate*** is the speed of communication between the system memory and the internal buffer or cache built into the drive. The ***internal data transfer rate*** is the speed at which the hard disk can physically write or read data to or from the surface of the platter and then transfer it to the internal drive cache or read buffer. The internal data transfer rate is influenced by:

  - the rate at which the head can read bits from the medium,

- the rate at which the next head starts reading data in case the data is spread across multiple surfaces,
- the rate at which the head can advance to the next cylinder when it finishes reading data in the current cylinder.

In addition, there is a fourth component to the overall time overhead that is related to the disk controller. The disk controller is the disk's processing unit and is responsible for receiving instructions and data and controlling the activities of the disk. The controller's execution time is another factor in the overhead of disk I/O:

- ***Controller overhead*** The time spent by controller to initiate and finalize I/O transfers.

The complication in true performance estimation is that modern controllers come with very large caches and do anticipatory caching of sectors. Transfer time from the cache to the processor is much smaller because there is no mechanical operation involved; the data are read from or written to the cache directly.

Common Measures

Disk manufacturers usually report minimum seek time, maximum seek time, and average seek time. ***Average seek time*** depends upon many factors and so is not always useful. Typical average seek times are between 3 ms and 13 ms. High-end servers usually have average seek times less than 4 ms, desktop computers around 8 to 10 ms, and mobile devices around 12 to 14 ms. Manufacturers often estimate average seek times based on statistical usage. From a theoretical point of view, the average seek time is the time to traverse one-third of the tracks.

Average rotational latency is between

½ rotation / minimum rotations per sec = 0.5 /5400 RPM

= 0.5/90 sec = 0.0056 sec = 5.6 ms

and

½ rotation / maximum rotations per sec = 0.5 /15000 RPM

= 0.5/250 sec = 0.002 sec = 2.0 ms

Modern disk drives have RPMs in the higher end of that range.

Transfer time depends on sector size, recording density, and rotation speed. Typical transfer rates are between 125 MB/sec and 200 MB/sec, but when transfers are measured from the cache, they can be as much as 400 MB/sec.

Example

What is the average time to read or write a 512 byte sector for a disk rotating at 5400 RPM, given advertised

- average seek time = 12 ms
- transfer rate = 5 MB/sec
- controller overhead = 2 ms

Time to read a 512 byte sector is

= seek time + rotational latency + transfer time + controller overhead
= 12 ms + 0.5/(5400/60) sec + 512 bytes/(5 MB/sec) + 2 ms
= 12 ms + 5.6 ms + 0.1 ms + 2 ms
= 19.7 ms

Rotational delay can be the dominating cost if seek time is much less.

Reliability

Disks eventually go bad. The magnetically stored bits gradually change, and information is lost. Disk drives are capable of detecting when bits go bad, and therefore the controller includes extra data, known as an ***error correction code***, when it writes information to the disk. When the controller reads back this information, it can detect whether errors have occurred in the data. The error correction codes act as redundant information that is used to verify the integrity of the data. The most commonly used ECC is called the ***Reed-Solomon algorithm***[3]. The better the ECC, the more computation time is needed. The number of bits of correction code associated with a sector is a design decision that determines the robustness of the error detection and the overhead as well.

Flash Storage

Flash memory is a type of EEPROM (electrically erasable programmable read-only memory) that has grown in popularity because:

- it is durable (resistant to shock)
- it is compact
- it is power-efficient
- it has much smaller latency than hard disks

In addition, its capacity is usually large enough for many applications (cameras, phones, media players) but is available in small capacities for small devices. Because of these features, it is the first technology that is competitive with magnetic disks in the secondary storage market.

There are two types of flash memory: NOR flash and NAND flash. The original technology was NOR flash. NOR flash provided random access memory. NAND flash provided much more storage capacity but could only be read and written in blocks. In addition it is much less expensive than NOR flash.

The limiting factor is that bits wear out with writes over time. The table shows that NAND flash will wear out faster than NOR flash. Flash drive controllers reduce the chance of wear-outs by trying to distribute writes uniformly across the memory. They do this by moving logical blocks that have been written a lot to different physical blocks. This is called ***level wearing***.

[3] Based on an article written by Reed and Solomon in 1960, "Polynomial Codes over Certain Finite Fields."

Characteristics	NOR flash	NAND flash
Typical use	BIOS	USB key
Minimum access size (bytes)	512	2048
Read time (msec)	0.08	25
Write time (msec)	10.00	1500 to erase + 250
Read bandwidth (MB/sec)	10	40
Write bandwidth (MB/sec)	0.4	8
Wear-out (writes per cell)	100,000	10,000 to 100,000
Best price/GB (2008)	$65	$4

Table 1: NOR versus NAND flash memory comparison (from Paterson & Hennessey).

Connecting Processors, Memory, and I/O Devices

A ***bus*** is a shared communication path whose purpose is to allow the transfer of data among the devices connected to it. A bus includes address, control, and data lines as well as lines needed to support interrupts and bus arbitration. Control lines include a read/write line.

A ***bus protocol*** is a set of rules that govern the behavior of the bus. Devices connected to the bus have to use these rules to transfer data. The rules specify when to place data on the bus, when to assert and de-assert control signals, and so on.

Among methods of interconnecting components of a computer system, buses are

- versatile
- low-cost
- a major bottleneck

Bus speed is limited by:

- bus length
- number of attached devices

Buses can be ***synchronous*** or ***asynchronous***; synchronous buses use a clock to control transfers, whereas asynchronous buses do not.

Because of clock skew and signal reflection, it is difficult to design buses with many parallel wires at high speed. ***Clock skew*** is the difference in absolute time between when two state elements see a clock edge. Clock skew arises because the clock signal will often use two different paths, with slightly different delays, to reach two different state elements. When the length of the wires is long and clock speeds are fast, skew can become a problem. The reflection of the clock signal is also a problem as clock speeds get faster.

Buses are gradually being replaced by serial point-to-point interconnection networks with switches. Nonetheless, we begin with a brief overview of buses.

Bus Types

Different kinds of buses are used to connect different parts of a computer system to each other.

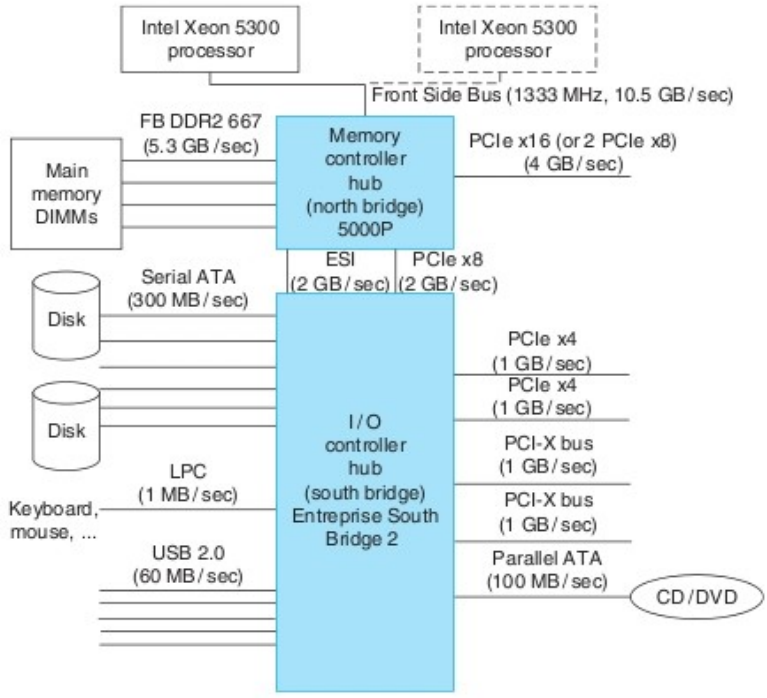


Figure 3: Schematic organization of the Intel 5000P I/O system (from [Paterson/Hennessey])

Processor-Memory Bus

These are sometimes called ***processor buses***. They are short, high speed buses, usually designed for a specific memory and processor to maximize bandwidth. They can be used when all devices that can be attached are known in advance and have known characteristics. Typically these are ***proprietary*** buses. Also called a front side bus.

I/O Bus

I/O buses have varying length and usually allow many different kinds of devices with varying speeds and data block sizes to be attached. I/O buses may be connected to the processor bus via a bridge or a separate controller interface, but usually they are connected to what is often called a system, or ***backplane***, bus, described below. I/O buses are almost always **standard, off-the-shelf** components. Examples include SCSI, USB, PCI Express, Serial ATA (SATA), and Serial attached SCSI (SAS) .

Backplane, or System, Bus

Most modern computers use a single, general-purpose bus to interconnect a variety of internal devices, including network interface cards, DMA controllers, and hard disk drives. These buses used to be called backplane buses but are now often called system buses. Usually, I/O buses are connected to the system bus, which in turn connects to the processor bus via a bridge.

The I/O Interconnects of the x86 Processors

The I/O system depicted in Figure 3 is typical of that of all x86 processors. There are two controller hubs, called the ***north bridge*** and the ***south bridge***. The processor connects to memory and peripheral devices via these hubs.

The north bridge is the memory controller hub, and connects the processor to the memory, the south bridge, and sometimes the graphics card (via a PCIe bus.) The south bridge is the I/O controller hub and connects the processor to all I/O devices, through a number of different kinds of attached buses such as the USB, PCI, SATA and so on.

Some processors, such as the AMD Opteron X4, incorporate the north bridge into the processor chip instead of its being a separate chip.

Interfacing I/O Devices to the Processor, Memory, and OS

This section answers the following questions:

- How is an I/O request transformed into device-specific commands?
- How is an I/O request actually communicated to the I/O device?
- How is data transferred to or from the memory?
- What are the respective roles of the application level software, the operating system, and the hardware?

The answers to these questions will vary depending upon how the computer system is designed to be used. Most modern computers and operating systems support multiprocessing, interactive use, and multiple users. To provide these features, the operating system must perform under the following constraints:

- The I/O system is shared among multiple processes.
- If processes are allowed to control devices directly, then throughput and response time will not be under the control of the operating system, and the computer system will perform poorly.
- If user processes are required to issue I/O requests directly to devices, then the programming task becomes much more tedious for the software engineer.
- I/O requests should be handled by the operating system, and the operating system must determine when the requested I/O has completed.
- Because I/O is handled by the operating system, the operating system must provide equitable access to I/O resources and device abstraction.

To satisfy these constraints, the operating system must be able to communicate with devices and prevent user programs from communicating with these devices. There are three different types of communication:

- Issuing commands to devices (e.g., read, write, seek, start, stop)
- Receiving notifications when devices need attention (e.g., I/O completion, error, load media)
- Transferring data between memory and a device.

The most common paradigm used in operating systems is a multi-layered approach to resources. Users make requests for I/O to application software. Application software translates these requests into calls to the operating system. The device drivers are the operating system's lowest level software; they are really two parts. One part issues I/O commands to the devices. The other part responds to signals sent by the device to the processor when they need attention. Together they maintain the queues needed to coordinate the requests. The operating system notifies applications when the I/O is completed.

Giving Commands to I/O Devices

In order for the processor to issue a command to an I/O device, it has to be able to address the device and it has to be able to deliver one or more commands to it. There are two different methods of addressing devices: ***memory-mapped I/O*** and ***special I/O instructions***.

Memory-Mapped I/O

In memory-mapped I/O, parts of the address space are reserved for I/O device ports. Device port locations are mapped to specific memory addresses in such a way that when the processor accesses these locations, it is really accessing the device ports. Ordinary instructions are written to these addresses in order to perform I/O as if they were data. When the processor references an address in this special part of the address space, the memory controller ignores the address; instead the hardware translates this to the corresponding I/O device register. The I/O device controller receives the data and interprets it as a command which it can then carry out.

Certain addresses are used for specific commands. For example, one address might be for reads from the device and another, for writes. In the Intel IA-32 instruction set, the identifiers `DATAIN` and `DATAOUT` are mnemonic names for memory addresses that are mapped to I/O device registers. The ordinary `MOV` machine instruction

```
MOV AL, DATAIN
```

moves a character from the device register into register `AL` and the instruction

```
MOV DATAOUT,AL
```

moves a character from register AL to the device register. An example of an IA-32 I/O program to read from the keyboard and echo the characters on the screen is shown below. IA-32 is a 32-bit Intel instruction format.

Example (Intel IA-32 instructions)

In the Intel-32 instruction set, there are two device status registers, INSTATUS and OUTSTATUS. The following program assumes that the keyboard synchronization flag is stored in bit 3 of INSTATUS and the display synchronization flag is stored in bit 3 of OUTSTATUS.

```
      LEA   EBP,LOC       # Register EBP points to LOC, the memory area
READ: BT    INSTATUS,3    # INSTATUS bit 3 is set if there is data in
      JNC   READ          # DATAIN; this loops waiting for data
      MOV   AL,DATAIN     # Transfer char into register AL
      MOV   [EBP],AL      # Transfer AL contents to address in EBP
      INC   EBP           # and increment EBP pointer
ECHO: BT    OUTSTATUS,3   # Wait for display to be ready
      JNC   ECHO
```

```
MOV   DATAOUT,AL   # Send char to display
CMP   AL,CR        # If not carriage return,
JNE   READ         #  read more chars
```

Notes.

- LEA *reg, addr* is an instruction to load the address *addr* into the pointer register *reg*.
- *BT i*s a bit-test instruction. It loads the value in bit 3 of the specified register into the carry bit; JNC will branch if the carry bit is 0.

Figure 4 illustrates the basic structure of the connections when memory-mapped I/O is used. The four bus lines are for data, addresses, a read signal, and a write signal. Figure 5, in contrast, shows the structure when special I/O instructions are used. In this case there are separate control lines from the CPU to memory and from the CPU to the I/O devices.

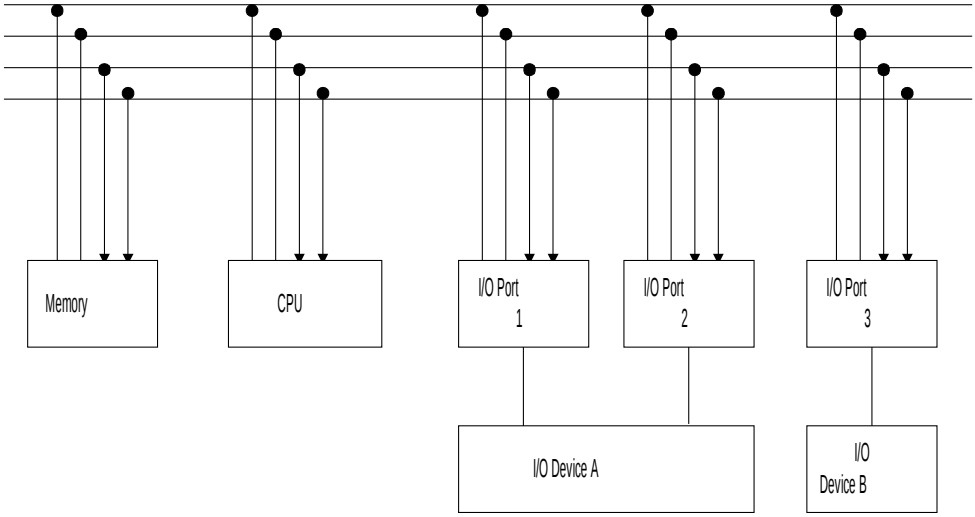


Figure 4: Lines used in memory-mapped I/O

Communicating Via Special I/O Instructions

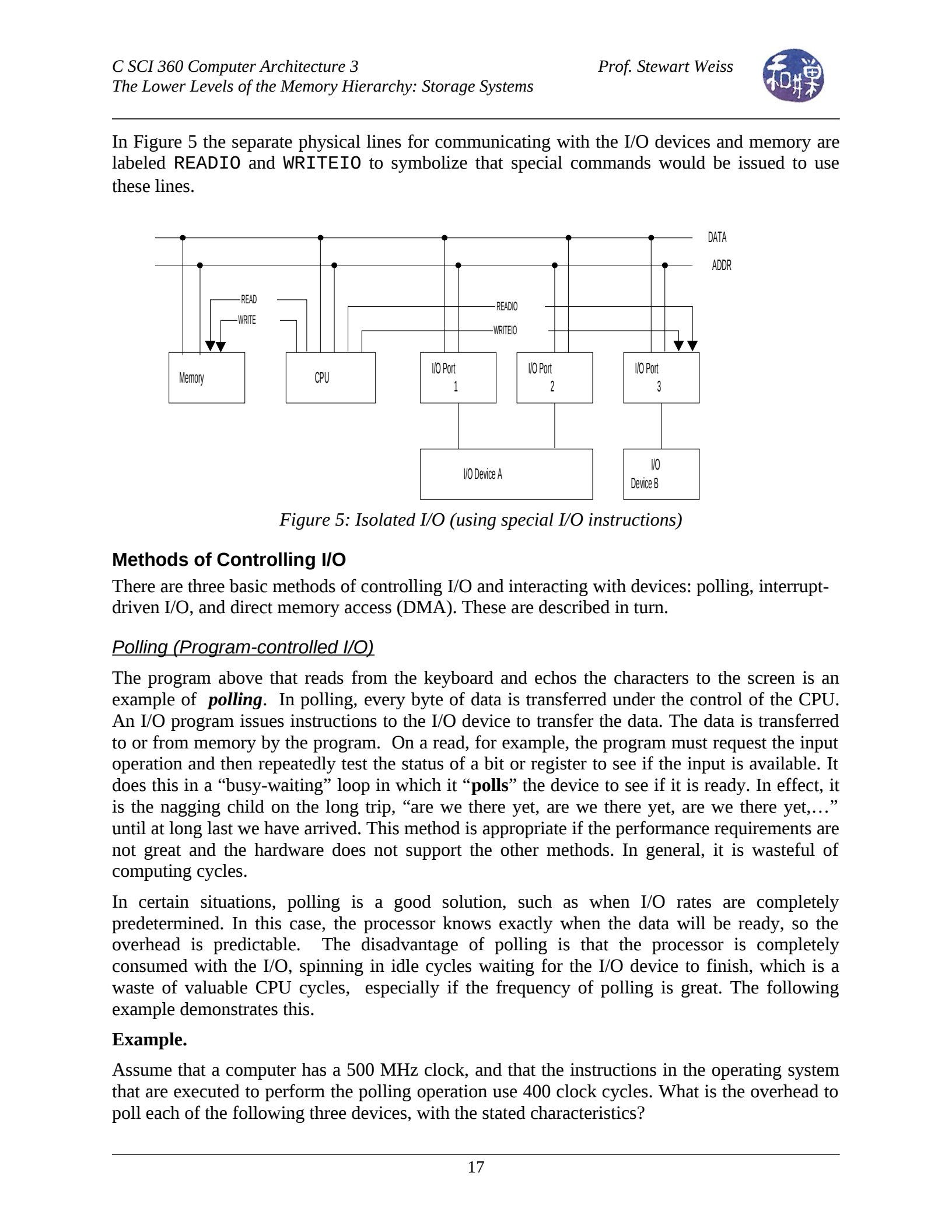
In **isolated I/O**, or **I/O-mapped I/O**, the processor has special instructions that perform I/O and a separate address space for the I/O devices. In isolated I/O, the same address lines are used to address memory and the I/O devices; the processor asserts a control line to indicate that the I/O devices should read the address lines. All I/O devices read the address but only one responds. For example, in the IA-32 instruction format,

```
IN REGISTER,DEVICE_ADDR
```

and

```
OUT DEVICE_ADDR,REGISTER
```

are the input and output instructions respectively. `DEVICE_ADDR` is an 8-bit address, and `REGISTER` is either `AL` or `EAX`. Some instruction sets, such as the IA-32, support both memory-mapped and isolated I/O.



In Figure 5 the separate physical lines for communicating with the I/O devices and memory are labeled `READIO` and `WRITEIO` to symbolize that special commands would be issued to use these lines.

Figure 5: Isolated I/O (using special I/O instructions)

Methods of Controlling I/O

There are three basic methods of controlling I/O and interacting with devices: polling, interrupt-driven I/O, and direct memory access (DMA). These are described in turn.

Polling (Program-controlled I/O)

The program above that reads from the keyboard and echos the characters to the screen is an example of ***polling***. In polling, every byte of data is transferred under the control of the CPU. An I/O program issues instructions to the I/O device to transfer the data. The data is transferred to or from memory by the program. On a read, for example, the program must request the input operation and then repeatedly test the status of a bit or register to see if the input is available. It does this in a "busy-waiting" loop in which it "**polls**" the device to see if it is ready. In effect, it is the nagging child on the long trip, "are we there yet, are we there yet, are we there yet,…" until at long last we have arrived. This method is appropriate if the performance requirements are not great and the hardware does not support the other methods. In general, it is wasteful of computing cycles.

In certain situations, polling is a good solution, such as when I/O rates are completely predetermined. In this case, the processor knows exactly when the data will be ready, so the overhead is predictable. The disadvantage of polling is that the processor is completely consumed with the I/O, spinning in idle cycles waiting for the I/O device to finish, which is a waste of valuable CPU cycles, especially if the frequency of polling is great. The following example demonstrates this.

Example.

Assume that a computer has a 500 MHz clock, and that the instructions in the operating system that are executed to perform the polling operation use 400 clock cycles. What is the overhead to poll each of the following three devices, with the stated characteristics?

1. Mouse: can produce an event 30 times per second.
2. Floppy disk: can transfer 2-byte blocks at a rate of 50KB/second.
3. Hard disk: can transfer 4-word blocks at a rate of 4MB/second.

Solution.

The operating system uses 400 2ns clock cycles per polling operation, or 800 ns.

Mouse: 30*800 ns /second = 0.000024 or 0.0024%.

Floppy Disk: 50KBytes/sec at 2 bytes per transfer implies that it performs 25,000 transfers per second. The overhead is 25,000*800 ns/second = 0.02 or 2%.

Hard Disk: 4MB/second at 4 words per transfer implies that it transfers 250,000 times per second. The overhead is 250,000*800 ns/ second = 0.2 or 20%.

This shows that the faster the device, the more overhead is involved in polling as a means of controlling I/O.

Interrupt-driven I/O

An alternative to programmed I/O is In ***interrupt-driven I/O***. In interrupt-driven I/O, a program running on the processor issues an I/O command to a device and then goes into a waiting state. Another process is run on the processor while the I/O is carried out. When the I/O completes, the I/O device sends a signal, called an ***interrupt***, to the processor. This signal is like an exception except that it is asynchronous.

This requires adding more signals to the bus and more logic to the I/O devices to allow them to notify the CPU when the device is ready for a new I/O operation, or when an I/O operation is complete. There are various ways to arrange this, some more complex and flexible than others. Regardless of the method, it has to provide a means of deciding which device caused the interrupt.

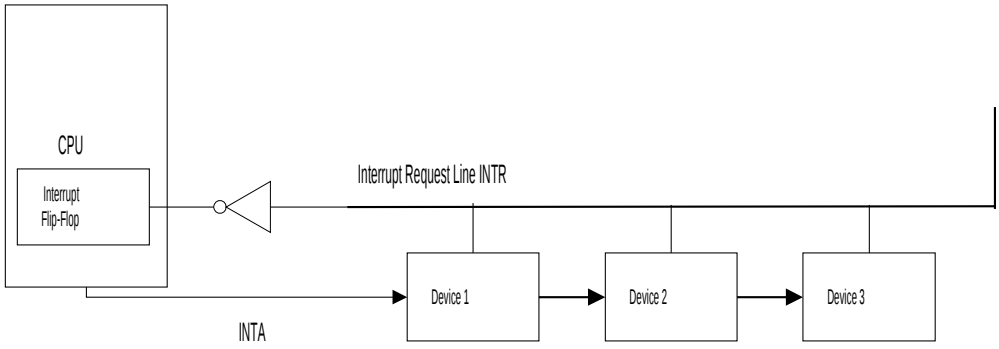


Figure 6: Interrupts with a single interrupt request line

The simplest scheme is a single control line on the bus, usually denoted `INTR`. All devices share this line. The line is the logical OR of the interrupt requests of the attached devices. If any device has issued an interrupt, the processor receives the signal. The processor then has to determine which device issued the request, which it does by *polling the status registers of the devices*. Once it has determined which device issued the request, it can initiate the transfer of data.

If interrupts arrive at the same time, the processor has to decide which to accept. Generally, it does this by assigning priorities to the devices. With a single interrupt line, priorities can be assigned by using a ***daisy chain*** scheme like the kind used for bus arbitration. In a daisy chain, the devices are arranged in a sequence such that each device is connected to the next one and passes messages down the line if they are not addressed to it. That is the purpose of the `INTA` line in Figure 6, which runs through the devices. The processor sends an Interrupt Acknowledge signal through the line. The closest device that has an outstanding interrupt request intercepts the signal.

Another solution is to use a multiple-line interrupt system. In this case each device has a dedicated `INTR` line and a dedicated `INTA` line. This makes it easy to decide which device caused the interrupt. The lines run into a priority arbitration circuit in the CPU. The CPU chooses the highest priority interrupt.

Handling Interrupts

When an interrupt occurs, the CPU needs to execute the subroutine that handles, or *services*, the interrupt. To do this, the contents of some of its registers must be saved. Modern processors typically allow two types of interrupts, one that forces a save of the entire register set, and another that saves only a minimal set of registers. The latter is the more efficient solution since only a few registers are saved, and the subroutine can choose to save whatever other registers it needs to in order to execute. This is faster than saving all registers, and avoids unnecessary work if there is no need to save them. The registers may be saved in special dedicated storage, or put on the system stack. The ***interrupt service routine*** (ISR) runs, saves whatever CPU state it must, and on completion, either the previous instruction sequence is resumed, or the scheduler is run to pick a different process to run.

How does the operating system know which ISR to execute? The answer is that once it knows which device caused the interrupt it knows which subroutine to run. In vectored interrupts, there is a portion of memory that contains an array, i.e., a vector, each of whose cells is the starting address of an interrupt service routine (ISR). Each device has an entry in this vector. When an interrupt occurs, the address of the device is used as an index into this array, and the starting address of the interrupt service routine is automatically loaded, once the registers have been saved.

There are other issues related to interrupts. They include:

1. Should interrupts be disabled while an interrupt service routine is running? If so, is the interrupt lost, or is it just that the response is delayed?
2. If not disabled, what happens if an interrupt occurs while an interrupt service routine is running?

There are various answers to these questions, depending on the complexity of the system. There are usually two registers, a ***Cause Register*** and a ***Status Register,*** used to solve the first question. The Cause Register has a bit for each different interrupt. If an interrupt occurs, the bit is set to 1, otherwise it is 0. The Status Register bits are used as an ***interrupt mask.*** If an interrupt is enabled, there must be a 1-bit in the corresponding position in the Status Register. AND-ing the two registers gives the set of enabled interrupts that have occurred.

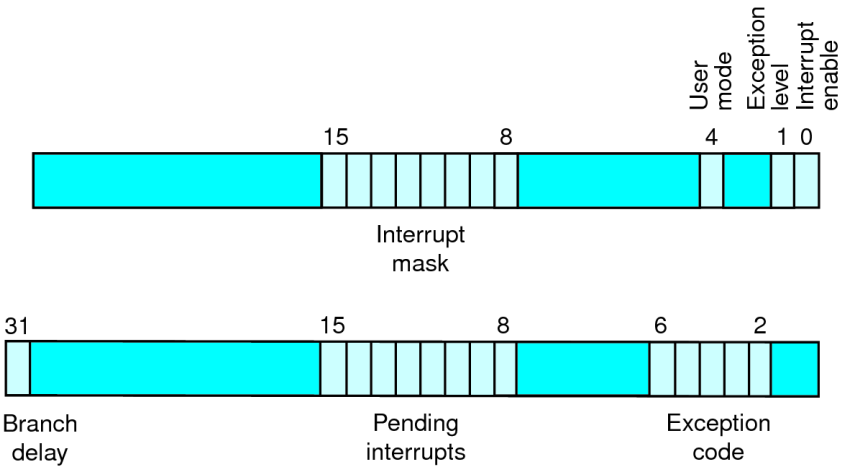


FIGURE 6.11 The Cause and Status registers. *This version of the Cause register corresponds to the MIPS-32 architecture. The earlier MIPS I architecture had three nested sets of kernel/user and interrupt enable bits to support nested interrupts. Section B.7 in Appendix B has more details about these registers.*

The mask can be used to set the interrupt priority level of the processor by a left-to-right ordering of the mask bits. If an interrupt occurs whose bit is to the left of another one, it has higher priority. By turning off all bits to the right of a given bit, the processor masks all lower level interrupts than a given level. If an interrupt occurs that is lower priority than the current priority level, it is ignored. If one occurs that is a higher or equal priority, the currently running process is preempted in favor of the interrupt service routine that handles the new interrupt, regardless of what it was doing. Each device has an associated priority level, and the ISR for that device runs at that priority level. For example, the power supply can send an interrupt if it senses an impending loss of power. This is the highest priority level on many machines. The system timer is also very high priority; it must keep accurate time and uses very little CPU time when it runs, so it is reasonable to allow it to run whenever it needs to, which is on the order of 60 times per second.

Overhead

The overhead of interrupts is much lower than that of program-controlled I/O. To illustrate, consider the hard disk from the preceding example, which could transfer 4-word blocks at a rate of 4 MB/second. Suppose that the operating system needs 500 clock cycles of processing time to handle an interrupt and that the disk is only busy 5% of the time.

For each of the 250,000 transfers per second that the disk could generate, the interrupt service routine, uses 500 clocks * 2 ns/clock = 1000 ns. But since it is busy only 5% of the time, the overhead is 5% of 250,000 transfers * 1000 ns per transfer per second = 0.05 * 250,000,000 ns per second = 12,500,000 ns/second , which is 1.25%. Recall that polling used 20% of the CPU time, so this is 93.75% reduction in overhead (1.25/20 = 0.0625, so 1.25% is 6.25% of 20%.) The difference is that in polling, the polling program must run whether or not there is a transfer to be done, whereas with interrupts, the ISR runs only when there is actually a transfer to be made.

Direct Memory Access (DMA)

Direct Memory Access* (DMA)** is a method of transferring data at a very high bandwidth with low overhead. The processor, under program control, effectively authorizes a special device to take charge of the I/O transfers to memory, allowing it to be the bus master until the I/O is completed. A device with this capability is called a ***DMA controller.

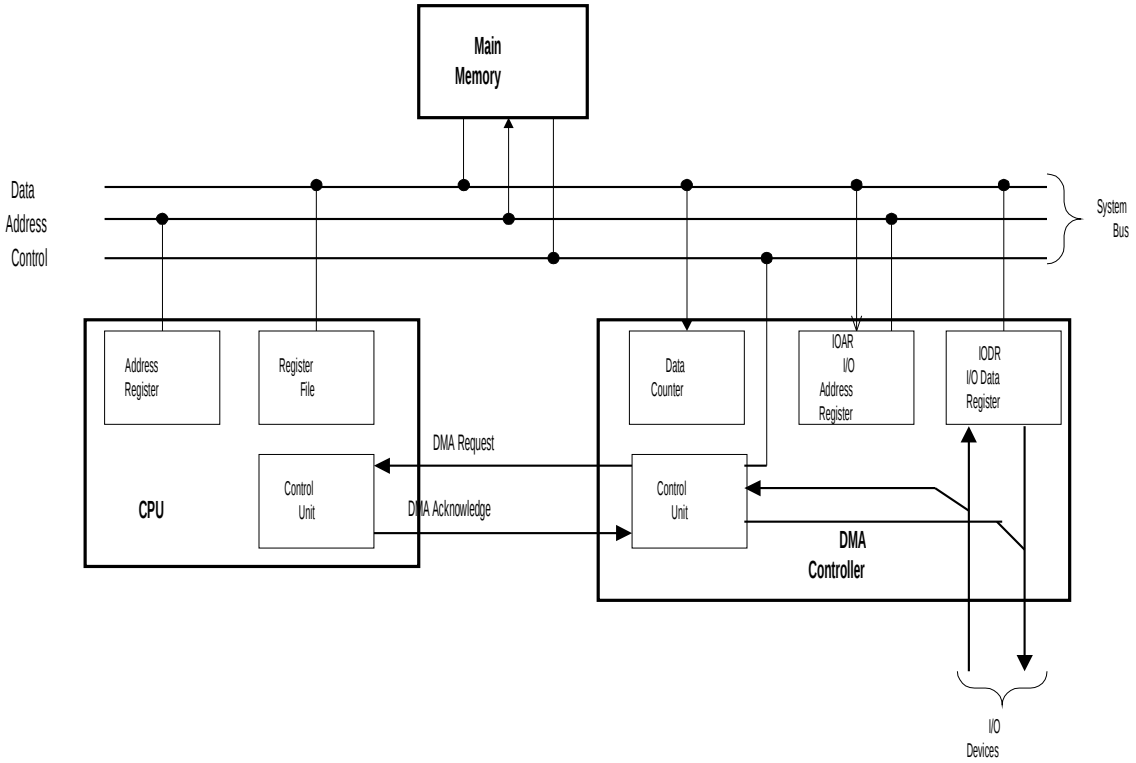


Figure 7: Circuitry for DMA interface

DMA significantly reduces the involvement of the processor in data transfers between memory and I/O devices. A DMA controller is an I/O processor that has the ability to communicate directly with memory, transferring large blocks of data between memory and the I/O devices to which it is attached. It achieves this because it is attached to the processor-memory bus on one side, and either an I/O bus or a dedicated device on the other, and it can be bus master on the memory bus. Typically, a single DMA controller will service multiple I/O devices. Certain devices, usually high-speed devices such as hard disks, CD-ROM drives, or network interfaces, may be equipped with DMA controllers. For example, a SCSI bus controller will have a DMA controller in its interface, making it possible for all devices on the SCSI bus to transfer data directly to or from memory with little CPU involvement.

A program running on the CPU will give the DMA controller a memory address, the number of bytes to transfer, and a flag indicating whether it is a read or a write. It will also give it the address of the I/O device involved in the I/O. The DMA controller becomes the bus master on the memory bus. If it is an input operation, the device will then start sending data to the DMA controller, which will buffer the data, and store it in successive memory locations as it becomes

available. If it is an output operation, it buffers the data from memory and sends it to the I/O device as it becomes ready to receive it. In effect, it does what the CPU would do, but the CPU is free to do other things in the meanwhile. Error: Reference source not found depicts the circuitry in a typical DMA controller interface.

The sequence just described is usually known as ***burst mode***. The typical sequence of operations in a burst mode input transfer would be:

1. The CPU executes two instructions to load the DMA controller's `IOAR` and `Data Counter`. The `IOAR` gets the start address in memory of the first byte to be stored and the `Data Counter` gets a count of the number of bytes to be transferred.
2. When the DMA controller is ready, it activates the DMA request signal[4]. This tells the CPU that it wants to use the bus.
3. The CPU relinquishes control of the bus and activates DMA Acknowledge[5] as part of the handshake. This tells the DMA controller that it can use the bus.
4. The DMA controller begins the transfer of data to memory using a hardware loop to update the `IOAR` and `Data Counter`.
5. If the I/O device is not ready, but the transfer is not complete, the DMA controller relinquishes the bus so that the processor can use it.
6. If the I/O device was not ready and it becomes ready, the DMA controller re-acquires the bus in the same way it did in step 2 above.
7. When the `Data Counter` reaches 0, the DMA controller releases the bus and sends an interrupt to the CPU.

Because the DMA controller owns the bus during a transfer, the CPU will not be able to access memory. If the CPU or the cache controller needs to access memory, it will be delayed. For small data transfers this may be acceptable, but not for large transfers, because it defeats the purpose of using DMA in the first place. Therefore, DMA controllers usually operate in two modes, one for small transfers and one for larger transfers, in which "cycle-stealing" is allowed.

Cycle-stealing mode is a compromise between the burst mode just described and a programmed I/O mode. In cycle stealing, the DMA controller relinquishes the bus after each byte or word of data, giving the processor the chance to use the bus. If the processor needs the bus, it uses it. If not, the CPU sends the DMA Acknowledge back to the DMA controller. The CPU is not interrupted to do this. The CPU and DMA controller are basically handshaking using the DMA Request and Acknowledge signals.

DMA Overhead Example

Suppose that a system uses DMA for its hard disk. The system characteristics are:

1. System Clock: 500 MHz (2 ns per cycle)
2. Hard Disk can transfer at 4MB/second using an 8KB block size.
3. 1000 clock cycles are used in the CPU to setup the I/O and 500 clock cycles are used afterwards in the CPU. What is the overhead of DMA transfers?

[4] Also called the Bus Request (BR) line.
[5] Also called the Bus Grant (BG) line.

Each transfer takes 8KB / (4MB/ second) = 0.002 seconds (2 ms). Thus there are
1.0 / 0.002 = 500 transfers per second.

If the disk is busy then it takes (1000 + 500) * 2 ns per transfer, which is 3000 ns per transfer. Since there are as many as 500 transfers per second, the total overhead is 500*3000 ns per second, or 1.5 ms per second, which is 0.15%.

DMA and Virtual Memory

In a system with virtual memory, DMA poses a problem – should the DMA controller use virtual or physical addresses? If it uses physical addresses, then it cannot perform reads or writes that cross page boundaries. To see this, imagine that pages are each 1 KB. If the DMA controller tries to write 3KB of data, then it will write into 3 or 4 physical pages. These pages may not belong to the same process, and may not be logically adjacent, and should be placed in the logically correct memory locations. On the other hand, if it uses virtual addresses, it will need to translate every address, slowing things down considerably and requiring a large RAM of its own. One solution is for the DMA controller to keep a cache of translations in its memory and update it using a replacement strategy such as LRU.

This is still inadequate because the page translations it has may go stale if the processor updates the page tables independently. For DMA to work properly, the processor must be prevented from changing the page tables during a DMA transfer.

DMA and Cache

DMA also creates problems with the cache. If the DMA controller is reading data directly from the disk into memory, then cache blocks may become ***stale***, because the cache blocks will not be consistent with their corresponding memory blocks, which are newer. This is called the ***stale data problem***. Similarly, the DMA controller might read from memory and get stale data because the system has a write-update cache that has not yet been flushed to memory. There are a few solutions:

- Route all I/O through the cache.
- Flush the cache for I/O writes and invalidate it for I/O reads.

To make this efficient, special hardware is provided for flushing the cache. regardless, there is overhead added because of the time needed to invalidate the cache or route the I/O through it.

In some systems, there is no special hardware and the operating system must provide the cache coherence by ensuring that the cache blocks are flushed before an outgoing DMA transfer is started and invalidated before a memory range affected by an incoming DMA transfer is accessed. This solution introduces adds software overhead to the DMA operation.

Example of Impact of I/O on System Performance

We illustrate the impact of I/O performance on overall system performance. Suppose that a benchmark program executes in 100 seconds of elapsed time in which 90% of the time is spent in the CPU and 10% is in waiting for I/O. Suppose that the number of processors doubles every two years, but they remain the same speed, and the I/O time does not change. How much faster will the program run at the end of six years?

Elapsed time is CPU time plus I/O time. The CPU time is divided by the number of processors, but the I/O time remains the same. The following table illustrates the changes in elapsed time and CPU time over the 6-year period of the problem.

Years elapsed	**CPU Time (seconds)**	**I/O Time**	**Elapsed Time**	**% I/O Time**
0	90	10	100	10
2	90/2 = 45	10	55	18
4	45/2 = 22.5	10	32.5	31
6	22.5/2 = 11.25	10	21.25	48

After six years, the elapsed time is 11 seconds; the speed increase is 100/21.25 = 4.7 (470%) even though there are 8 times as many processors. The theoretical improvement, had I/O kept pace with CPU improvements would be 800%.

Redundant Arrays of Inexpensive Disks (RAID)

RAID was originally invented to improve I/O performance. The idea was to replace a few large disks by many smaller disks. This would result in more read heads, fewer seeks, more independent simultaneous accesses, reduced power consumption, and smaller space requirements. It was also believed that smaller disks were less reliable, so to compensate, redundancy was added in the form of additional disks. The result was RAID. RAID is now used primarily as a means of increasing reliability and dependability, not performance.

Summary of RAID

RAID is categorized by the RAID level, an integer from 0 to 6. The different levels differ by how much redundancy there is and how errors are checked. RAID 1 and RAID 5 are the most widely used. The following is a brief synopsis.

RAID 0 - There is no redundancy; it is designed to improve performance by the use of striping. ***Striping*** is the distribution of consecutive blocks of data across multiple disk drives. Striping combines several disk drives into a single logical volume. In many cases, this is done through the use of hardware controllers. The advantage of striping is that several different devices can be performing the I/O simultaneously, making the I/O faster. But it is no more reliable than an ordinary non-RAID disk.

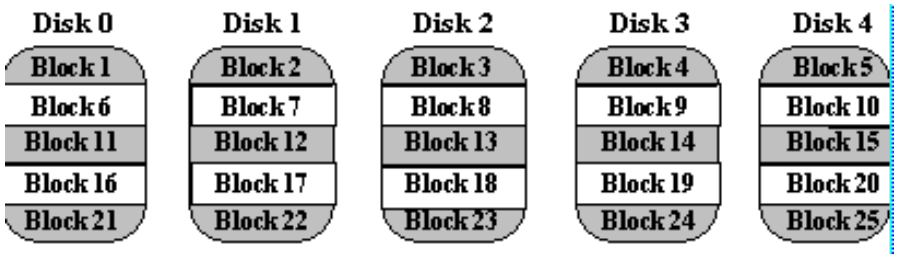


Figure 8: RAID 0. [6]

RAID 1 - This uses ***mirroring***, which is a technique in which the disks are doubled and each write is replicated on a second disk. Mirrors are usually used to guard against data loss due to drive failure. Each drive in a mirror contains an identical copy of the data. When an individual drive fails, the mirror continues to work, providing data from the drives that are still functioning. The computer keeps running, and the administrator has time to replace the failed drive without user interruption.

RAID 2 - This uses error detecting and correcting code (bit-level striping with dedicated Hamming-code parity). A RAID 2 system would normally have as many data disks as the word size of the computer, e.g., 32 or 64. RAID 2 requires the use of extra disks to store an error-correcting code for redundancy. With 32 data disks, a RAID 2 system would require 7 additional disks for a Hamming-code ECC.

For various reasons, RAID 2 is not used in practice, having been replaced by higher level RAID schemes.

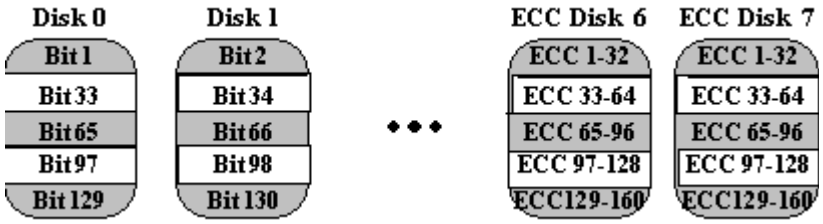


Figure 9: RAID 2

RAID 3 - This uses bit-interleaved parity; improves on RAID 1 by adding only enough redundant data to the secondary disks to be able to restore the lost data on the primary. Parity is a simple example of RAID 3 -- for a group of N disks there will be an extra disk, which stores the parity sum of the data from those disks. If a disk fails, it can be reconstructed by adding the other disks and subtracting from the parity disk. If two go bad, this fails. RAID 3 is not used very much in practice any longer. Like RAID 2, the number of disks should be equal to the word size of the machine.

[6] The figures depicting RAID are from http://www.ecs.umass.edu/ece/koren/architecture/Raid/basicRAID.html.

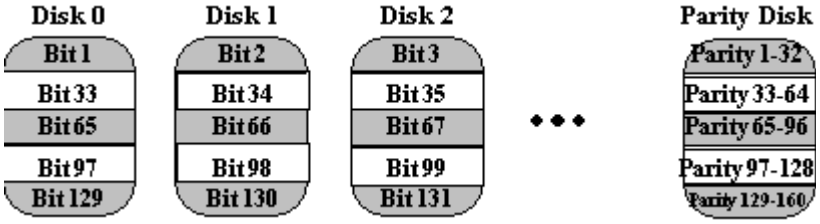


Figure 10: RAID 3

RAID 4 - This is similar to RAID 3, but it uses block-interleaved parity instead, and data accesses occur differently. The block-interleaved, parity disk array is similar to the bit-interleaved, parity disk array except that data is interleaved across disks in blocks rather than in bits. The size of these blocks is called the ***striping unit***.

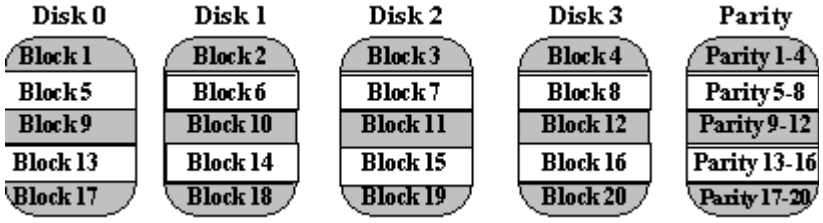


Figure 11: RAID 4

RAID 5 - This uses a ***distributed block-interleaved parity***, which is an enhancement of RAID 4. The problem with RAID 4 is that the parity disk must be updated on every write. In RAID 5, the parity information is distributed across all of the disks and requires that all drives but one be present to operate. The array is not destroyed by a single drive failure, but performance is degraded.

RAID 6 - Uses P+Q redundancy, which is a method of error correction that allows the disks to recover from two simultaneous failures. The P+Q redundant disk arrays are structurally very similar to the block-interleaved distributed-parity disk arrays of RAID 5. RAID 6 provides fault tolerance up to two failed drives Using Reed-Solomon codes.) This makes larger RAID groups more practical, especially for high-availability systems. Like RAID 5, a single drive failure results in reduced performance of the entire array until the failed drive has been replaced and the associated data rebuilt.

RAID 10 – This is a relatively new RAID scheme, essentially combining the striping of RAID 0 and the mirroring of RAID 1.

Alphabetical Index