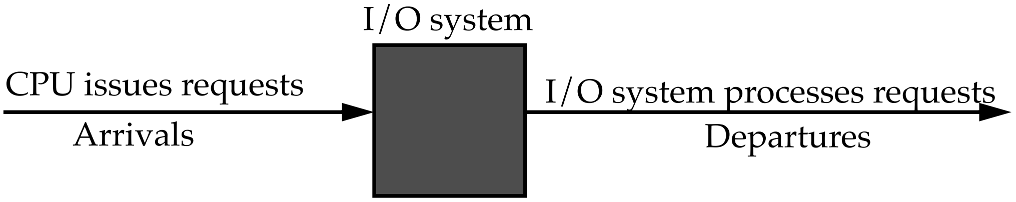
Queuing theory

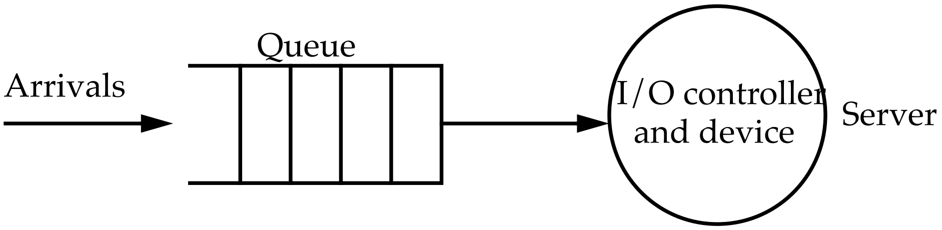
Given the important of response time (and throughput), we need a means of computing values for these metrics.

Our black box model:



Let’s assume our system is in steady-state (input rate = output rate).

The contents of our black box.



I/O requests “depart” by being completed by the server.

Queuing theory

Elements of a queuing system:

- *Request & arrival rate*

  This is a single “request for service”.
  The rate at which requests are generated is the *arrival rate*.

- *Server & service rate*

  This is the part of the system that services requests.
  The rate at which requests are serviced is called the service rate.

- *Queue*

  This is where requests wait between the time they arrive and the time their processing starts in the server.

Queuing theory

Useful statistics

- $\text{Length}_{\text{queue}}, \text{Time}_{\text{queue}}$

  These are the average length of the queue and the average time a request spends waiting in the **queue**.

- $\text{Length}_{\text{server}}, \text{Time}_{\text{server}}$

  These are the average number of tasks being serviced and the average time each task spends in the **server**.

  Note that a server may be able to serve *more than one* request at a time.

- $\text{Time}_{\text{system}}, \text{Length}_{\text{system}}$

  This is the average time a request (also called a task) spends in the **system**.

  It is the sum of the time spent in the queue and the time spent in the server.

  The length is just the average number of tasks anywhere in the system.

Queuing theory

Useful statistics

- *Little's Law*

  The **mean number of tasks in the system = arrival rate * mean response time**.

  $$\text{Length}_{\text{System}} = \text{Arrival Rate} \times \text{Time}_{\text{System}}$$

  This is true only for systems in equilibrium.

  We must assume any system we study (for this class) is in such a state.

- *Server utilization*

  This is just

  $$\text{Server utilization} = \frac{\text{Arrival Rate}}{\text{Server Rate}} \qquad \text{where Rate} = 1/\text{Time}$$

  This must be between 0 and 1.

  If it is larger than 1, the queue will grow infinitely long.

  This is also called *traffic intensity*.

Queuing theory

Queue discipline

This is the order in which requests are delivered to the server.

Common orders are FIFO, LIFO, and random.

For FIFO, we can figure out how long a request waits in the queue by:

$$\text{Time}_{\text{System}} = \text{Length}_{\text{Queue}} \times \text{Time}_{\text{Server}} + \text{Mean time for server to finish current tasks when request arrives}$$

The last parameter is the hardest to figure out.

We can just use the formula:

$$\text{Average residual service time} = \frac{1}{2} \times \text{Weighted mean time} \times (1 + C)$$

C is the coefficient of variance, whose derivation is in the book.

(don't worry about how to derive it - this isn't a class on queuing theory.)

Queuing theory

Example: Given:

- Processor sends 10 disk I/O per second (which are exponentially distributed).
- Average disk service time is 20 ms.

On average, how utilized is the disk?

$$\text{Server utilization} = \frac{\text{Arrival Rate}}{\text{Server Rate}} = \frac{10}{\frac{1}{0.02}} = 0.2$$

What is the average time spent in the queue?

When the service distribution is exponential, we can use a simplified formula for the average time spent waiting in line:

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 20ms \times \frac{0.2}{(1 - 0.2)} = 5ms$$

What is the average response time for a disk request (including queuing time and disk service time)?

$$\text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 5 + 20ms = 25ms$$

Queuing theory

Basic assumptions made about problems:

- System is in equilibrium.
- Interarrival time (time between two successive requests arriving) is exponentially distributed.
- Infinite number of requests.
- Server does not need to delay between servicing requests.
- No limit to the length of the queue and queue is FIFO.
- All requests must be completed at some point.

This is called an M/G/1 queue

M = exponential arrival

G = general service distribution (i.e. not exponential)

1 = server can serve 1 request at a time

It turns out this is a good model for computer science because many *arrival* processes turn out to be **exponential**.

Service times, however, may follow any of a number of distributions.

Disk Performance Benchmarks

We use these formulas to *predict* the performance of storage subsystems.

We also need to measure the performance of real systems to:

- Collect the values of parameters needed for prediction.
- To determine if the queuing theory assumptions hold (e.g., to determine if the queueing distribution model used is valid).

Benchmarks:

- *Transaction processing*

  The purpose of these benchmarks is to determine how many small (and usually random) requests a system can satisfy in a given period of time.

  This means the benchmark stresses **I/O rate** (number of disk accesses per second) rather than **data rate** (bytes of data per second).

  Banks, airlines, and other large customer service organizations are most interested in these systems, as they allow simultaneous updates to little pieces of data from many terminals.

Disk Performance Benchmarks

- *TPC-A and TPC-B*

  These are benchmarks designed by the people who do transaction processing.

  They measure a system's ability to do random updates to small pieces of data on disk.

  As the number of transactions is increased, so must the *number of requesters* and the *size of the account file*.

  These restrictions are imposed to ensure that the benchmark really measures disk I/O.

  They prevent vendors from adding more main memory as a database cache, artificially inflating TPS rates.

- *SPEC system-level file server (SFS)*

  This benchmark was designed to evaluate systems running Sun Microsystems network file service, NFS.

Disk Performance Benchmarks

- *SPEC system-level file server (SFS)*

  It was synthesized based on measurements of NFS systems to provide a reasonable mix of reads, writes and file operations.

  Similar to TPC-B, SFS **scales** the size of the file system according to the reported *throughput*, i.e.,

  It requires that for every 100 NFS operations per second, the size of the disk must be increased by 1 GB.

  It also limits average *response time* to 50ms.

- *Self-scaling I/O*

  This method of I/O benchmarking uses a program that **automatically scales several** parameters that govern performance.

  - Number of unique bytes touched.

    This parameter governs the total size of the data set.

    By making the value large, the effects of a cache can be counteracted.

Disk Performance Benchmarks

- *Self-scaling I/O*
  - Percentage of reads.
  - Average I/O request size.

    This is scalable since some systems may work better with large requests, and some with small.
  - Percentage of sequential requests.

    The percentage of requests that sequentially follow (address-wise) the prior request.

    As with request size, some systems are better at sequential and some are better at random requests.
  - Number of processes.

    This is varied to control concurrent requests, e.g., the number of tasks simultaneously issuing I/O requests.

Disk Performance Benchmarks

- *Self-scaling I/O*

  The benchmark first chooses a nominal value for each of the five parameters (based on the system's performance).

  It then varies each parameter in turn while holding the others at their nominal value.

  Performance can thus be graphed using any of five axes to show the effects of changing parameters on a system's performance.

Reliability, Availability and RAID

Some definitions:

- *Reliability*

  Refers to the dependability of individual components of a system.

- *Availability*

  Is the system still available to the user after a failure of one or more of its components ?

Adding hardware can therefore improve *availability* but it can **NOT** improve *reliability*.

Disk arrays:

The basic idea behind **disk arrays** is that by adding disks and therefore more disk arms working in parallel, *bandwidth* is improved.

Individual process-request seek latencies can be overlapped in time.

This is cost effective since price/megabyte is independent of disk size.

However, *latency* for small requests is not improved because it still takes all of the usual latency to get to a randomly selected block.

Reliability, Availability and RAID

- *Striping*

  In disk arrays, the data from files can be **striped** across several disks.

  This *increases bandwidth* by allowing a file to be read from *more than one* disk at a time.

  The data is distributed round robin between the disks.

Problems with disk arrays:

- *Reliability*

  Disks have a *mean time to failure* of about 20 years.

  However, a collection of 8 disks will experience a failure (on average) every 2.5 years.

  N devices generally have 1/N the reliability of a single device.

Reliability, Availability and RAID

Problems with disk arrays:

- *Reliability*

  Increase the number to 1000 disks, and a failure occurs every 1/50th of a year or every week !

  When a disk fails, it takes its data with it.

- *Availability*

  The other problem with disk arrays is that the disk array becomes unusable after a single failure.

We need a scheme to prevent data loss when a disk fails, and to allow the system to recover from the failure (remain available.)

Reliability, Availability and RAID

The solution: **Redundant Arrays of Inexpensive Disks.**

Improves availability by adding *redundant* disks.

When a disk fails, the lost information can be reconstructed from redundant information.

This works since the *mean time to failure* (MTTF) of disks is long (years) and the *mean time to repair* (MTTR) is short (hours).

The idea behind RAID is to use some of the disks to store error correction information for the rest of the disks.

RAIDs do NOT have to detect errors.

The ECC kept on each sector by each disk allows the disk electronics to check and detect disk failures.

RAID Levels

There are 7 levels of RAID, each of which can be characterized by their availability and overhead.

Raid Level		Failures survived	Data Disks	Check Disks
0	Nonredundant	0	8	0
1	Mirrored	1	8	8
2	Memory-style ECC	1	8	4
3	Bit-interleaved parity	1	8	1
4	Block-interleaved parity	1	8	1
5	Block-interleaved distributed parity	1	8	1
6	P+Q redundancy	2	8	2

- *RAID 0*

  This is no redundancy at all.

  However, it is the fastest and cheapest RAID level.

  This refers to the nonredundant disk array discussed previously.

RAID Levels

- *RAID 1*

  Disks in this configuration are mirrored or copied to another disk.

  With this arrangement, the data on a failed disk can be easily replaced by reading it from the other disk.

  In addition, **reading** is actually *faster* than it is for RAID 0 because read requests to the same disk can be split between the two disks.

  Writing is a little *slower*, though, because the file system must wait for the slower of the two requests.

  Remember, both disks must be updated.

  Also, seek time is different between the two disks since they are not synchronized.

  The main problem with RAID 1 is that it imposes a 50% space penalty.

  Therefore, it is the most expensive solution.

RAID Levels

- *RAID 3 (Bit interleaved parity)*

In this scheme, data is striped across disks in *very small units*.

These units are so small that all disks must work together on both reads and writes because a **single sector** actually spans **all** of the disks.

Redundancy is implemented by *calculating parity* and storing it on the check disk.

The overhead for **n** data disks is **1** disk, for a storage efficiency of **n/(n+1)**.

This level can survive a single disk failure and reconstruct data using the remaining disks.

This RAID level is somewhat limited since the entire disk system can only handle one request at a time.
Thus, the sustainable request rate is no higher than that of a single disk.

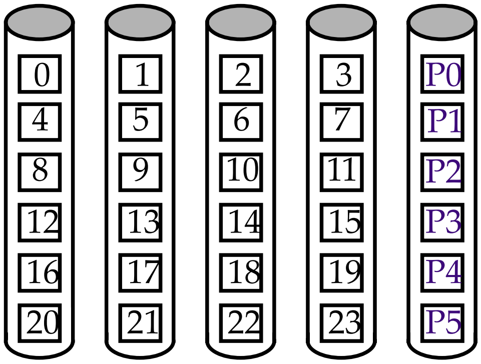
RAID Levels

- *RAID 4 (Block interleaved parity)*

This is similar to RAID 3, but individual disks each have a **block** (or more) of consecutive data within a stripe.

This means that *each disk* can handle an individual small read request if all disks are working.

However, writes are still a problem since the *parity disk* must participate in all write operations, which creates a bottleneck.



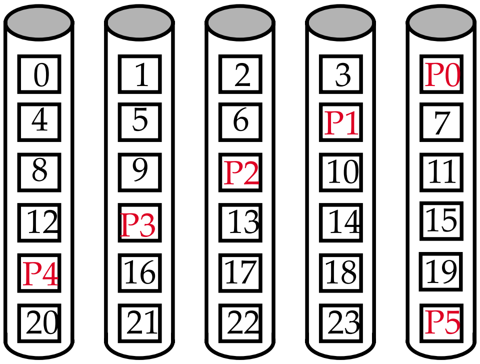
RAID 4

RAID Levels

- *RAID 5 (Rotated parity)*

The biggest problem with RAID 4 is that the parity disk is a bottleneck.

In RAID 5, the parity information may be stored on any disk.



RAID 5

Advantages:

Reads can be spread among all n+1 disks.

For writes, the parity disk bottleneck is eliminated.

RAID Levels

- *RAID 5 (Rotated parity)*

With respect to the parity calculation and update, RAID 5 does have a problem writing small pieces of data.

If a full stripe is written, the parity calculation is simple to implement.

XOR all the data together to get the parity.

If only a **single block** is written, though, the system must somehow figure out the new parity.

The most straightforward solution is to read the rest of the blocks in the stripe and XOR them together.

A better solution involves only four accesses:

- Read the old data.
- Compare old data to new data (to determine which bits change).
- Read and change the old parity.
- Write the new data and new parity.

RAID Levels

- *RAID 6 (P+Q parity)*

  The objective of RAID 6 is to survive two disk failures.

  This is done using schemes such that only **n** out of **n+2** disks are necessary to reconstruct the data.

  This RAID level is not common today because disk failures are still relatively rare.

If a disk in a parity-based RAID fails, no data is lost if a new disk is installed and "updated" before another crash takes place.

Since stripes tend to be relatively small (usually less than 32 disks), the chance that another disk will fail is relatively low.

  Most people don't worry about it.

RAID Issues:

- *Mapping data to disks.*

  The first problem in a RAID system is how to map data to disks.

  Usually, this is done using simple modulo arithmetic.

  It is more complex for RAID 5, but there is still a formula that allows you to determine where a given block is located.

- *Reconstruction*

  After a disk has failed, it is replaced.

  But we are not done since the new disk must be reconstructed with the lost information in order to allow the array to withstand another crash.

  Since the lost information is "embedded" in the remaining disks, reconstruction is possible and carried out immediately.

Designing an I/O system: Basics

Now that we've seen how to *estimate* performance on an I/O system.

And how to actually *measure* performance.

We are ready to talk about how to build one.

The objective is to find a design that is expandable and that meets goals for **cost** and **variety of devices** while *avoiding bottlenecks* to I/O performance.

In designing an I/O system, analyze *performance*, *cost* and *capacity* using *various I/O connection schemes* and *different numbers* of I/O devices of each type.

Here are the steps to follow in designing an I/O system:

- *List the types of I/O devices and buses, and their costs.*

- *List the physical requirements of each device.*

  These include volume, power, connectors, bus slots, etc.

  This won't be a problem for paper examples, but it certainly will be for real systems.

Designing an I/O system: Basics

- *Figure out the CPU resource demands for each I/O device.*

  Clock cycles to initiate, support the operation of, and complete requests.

  Clock stalls from I/O access to memory.

  Clock cycles to recover from an I/O activity, such as a cache flush.

- *List memory and I/O bus resource demands for each device.*

  Bandwidth of main memory and the I/O bus can often be a bottleneck, particularly when many devices are connected to a single bus.

- *Compute performance for various configurations of devices, buses, etc.*

  This can really only be done by building the system and measuring it.

  If this is not feasible (which is usually the case), the next best thing is a detailed simulation.

  Queuing models can be used to get a rough estimate of performance.

Designing an I/O system: Basics

Remember, performance can be measured as:

- Megabytes per second.
- I/Os per second.

This is dependent on the needs of the applications.

The goals for the design should also be clear:

- Is it a design to maximize performance at any cost ?
- Is it the cheapest system that will satisfy minimum requirements ?
- Is it the best price/performance ?

Look over the examples in the text !

Tertiary storage

Many modern computer systems now use **removable media** to store their data.

Advantages:

- *Inexpensive*

  Removable media cost a lot less because you only pay once for the machinery to read, write, and transport the medium.

  Since removable media use similar technology to non-removable media, the media costs are similar but the mechanism cost is much lower.

- *Low power*

  Disks use power to rotate.

  A major advantage of removable media is that they do not consume power.

- *Unerasability*

  Some removable media (WORM) can not be erased even if the system requests it.

Tertiary storage

Components:

- *Tape robots*

  These systems typically hold thousands of tape cartridges and can load any cartridge in under 20 seconds.

  IBM 3490 cartridges (the most common today) hold 9 GB of data per cartridge and transfer at 9 MB/sec.

- *Optical disk jukeboxes*

  These are usually used for two purposes:

  Small randomly-accessed data.

  Data that should not ever be overwritten (even accidentally).

  The cost per GB is higher than for tape, but seek time is much better.

Tertiary storage

How it works:

- *Moving data from disk to tertiary*

  Data is moved from disk to tertiary storage when the disk gets full.

  This is called *file migration* or simply *migration*.

  Migrating data is done to free up disk space.

  Files are picked for migration according to several factors:

  - How big they are.
  - When they were last used.
  - Cost to retrieve the file.
  - Other factors (possibly file type and/or user).

  The device to which the file is migrated can also depend on these factors.

  Small files might go to optical disk while large files are sent to a tape robot.

  Migration can also occur from one tertiary storage device to another.

Tertiary storage

- *Moving data from tertiary to disk*

  Data is moved from tertiary to disk when it is needed.

  It may also be prefetched if the system believes that the file might be used soon.

File migration issues

Tertiary storage is very much an *ad hoc* art these days.

System designers build their systems based on what others have done because there is relatively little concrete research on what works and what does not.

Tertiary storage

File migration issues

- *When should a file be migrated ?*

  How should the system choose the files to move from disk to tape ?

  This is very important because a user notices even a single miss.

  It may take close to a minute to retrieve a file from tape !

  Migrating just one file that should not have been moved can adversely impact a user’s session.

- *When should a file be deleted from disk ?*

  Just because a file has been migrated to tape does not mean it should be deleted from disk.

  When should its space be reclaimed ?

Tertiary storage

File migration issues

- *When should a file be moved from tape to disk ?*

  For demand fetches, this is obvious.

  However, there is also **prefetching** and **clustering** that might help improve performance.

- *What kinds of devices and layouts work best for various kinds of files ?*

  Again, clustering is important, as is transfer time versus time to first byte.

Fallacies and Pitfalls

- *Media cost is not the same as actual cost.*

  Just because a disk costs $0.20/MB does **NOT** mean you can pay $200,000 and get a terabyte of disk.

  There are lots of other costs associated with I/O systems.

  For example, disks need controllers, I/O buses, system buses, power supplies, and mounting hardware.

  This supporting structure becomes much more expensive as the number of disks supported grows.

  The same is true for removable media.

  It only costs $1000 to buy a terabyte of magnetic tape.

  But that does not include tape readers, a robot, software, and all the other pieces necessary to build a full system.

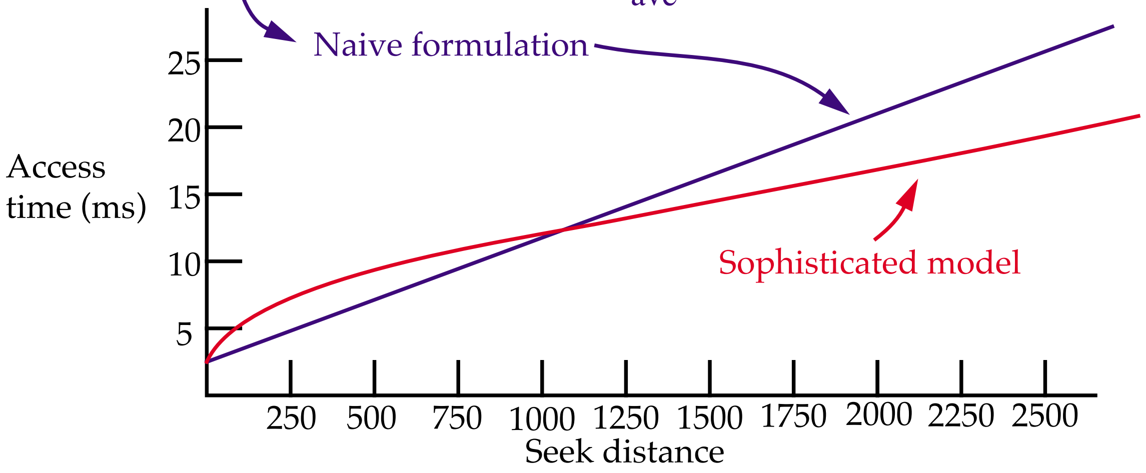
Fallacies and Pitfalls

- *Disk seek time is not linear.*

A disk head must accelerate to maximum velocity, travel across tracks, decelerate, and settle.

Most of the head's time is spent accelerating and decelerating, a non-linear activity.

$$\text{Time}_{\text{seek}} = \text{Time}_{\text{min}} + \frac{\text{Distance}}{\text{Distance}_{\text{ave}}} \times (\text{Time}_{\text{ave}} - \text{Time}_{\text{min}})$$



Fallacies and Pitfalls

- *Disk seek time is not linear.*

For disks with more than 200 cylinders, Chen and Lee [1995] modeled the seek distance as:

$$\text{Seek time (distance)} = a \times \sqrt{\text{Distance} - 1} + b \times (\text{Distance} - 1) + c$$

where a, b and c were computed as:

$$a = \frac{-10 \times \text{Time}_{min} + 15 \times \text{Time}_{ave} - 5 \times \text{Time}_{max}}{3 \times \sqrt{\text{Number of cylinders}}}$$

$$b = \frac{7 \times \text{Time}_{min} - 15 \times \text{Time}_{ave} + 8 \times \text{Time}_{max}}{3 \times \text{Number of cylinders}}$$

$$c = \text{Time}_{min}$$

The curve represented by this model in shown on the previous slide in red.

Fallacies and Pitfalls

- *I/O will become more important with time.*

  As our society stores more and more information on computer media, the ability to get to that information will become ever more important.

  The NASA Mission to Project Earth will capture more than a terabyte of data per day from satellites.

  How can we find a needle in that haystack ?

  Similarly, future libraries may dispense with physical books and instead keep information online.

  This makes it easier to distribute the information, but getting data to and from storage will be a bottleneck unless progress is made.