Third I.N.F.N. International School on

"Architectures, tools and methodologies for developing efficient large scale scientific computing applications"

Ce.U.B. Bertinoro (FC) 23 - 29 October 2011

Efficient I/O

Vincenzo Vagnoni
(INFN Bologna)

Outline

- Introduction
  - What is I/O and overview of I/O models
- Storage devices
  - Overview of common use technologies
- Page caching
  - Brief explanation and practical demonstration of how it works
- I/O monitoring
  - Discussion on common tools useful to understand I/O of your applications
  - Practical demonstration
- Advanced I/O API
  - Scatter/Gather, Memory mapping, Asynchronous I/O
  - Practical demonstration

Disclaimer

- This lecture is far from being a complete overview of all aspects of I/O
  - There is a world behind it
  - I can just touch what are in my opinion some topical points (and not all of them), focusing on file I/O
- In particular I will not be covering networking technologies and techniques
  - This is yet another world and would need at least a lecture on its own
  - By the way many concepts and even API functions are in common with file I/O
    - Will only make one example to show this

Introduction

First of all let's state the obvious (and less obvious)

- The obvious statement: **computation "inside" computers is useful only if some results are communicated "outside"**
- Less obvious (but still obvious): **communication (I/O) must be as fast as the computation is, otherwise an I/O bottleneck arises**
- Two extreme cases:
  - **CPU bound problems**
    - The amount of data needed in input and produced in output is negligible → total time is dominated by computation (e.g. HEP MC simulations)
  - **I/O bound problems**
    - The data processing is trivial and involve a limited number of cycles → total time is dominated by I/O (e.g. some light algorithms of HEP analysis jobs)
- Many real use cases live in some point between the two
- Rule of thumb: when you run a data processing job of any kind, and the **CPU load of the job is not approaching 100%**, it means that very likely there is an I/O bottleneck in your application
  - Corollary: if the CPU load is <<100% then you are wasting a lot of precious CPU cycles, and you need to improve the efficiency of your I/O

I/O abstraction and semantics

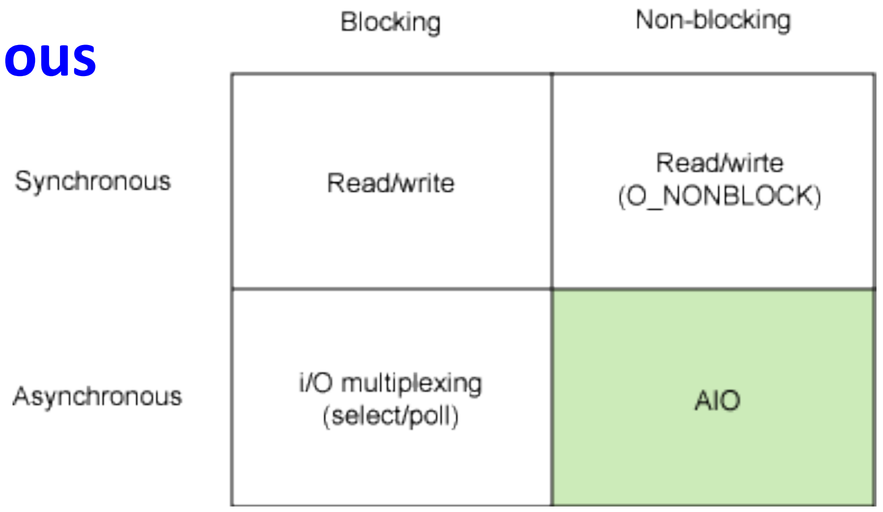
- The Operating System virtualizes a wide range of devices into a few simple abstractions, as
  - Storage (including hard drives, tape drives, etc.)
  - Networking (including 1-10 GE, etc.)
- OS provides consistent calls to access the abstractions
  - Otherwise, programming would too hard and unsafe
- There are various approaches for I/O processing
  - **I/O buffering**: buffered or unbuffered
    - i.e. file system access is usually buffered in kernel space via the page cache, and there can be even another buffering layers in the user space to limit user to kernel context switches
  - **I/O model**: blocking or non-blocking, synchronous or asynchronous
    - Simple minded blocking I/O is mostly used in common applications

Buffered I/O

- if the buffers on the hardware device are small, blocking I/O calls can become inefficient
  - e.g. frontend memory of a hard disk is usually very small
- buffered I/O allows the kernel to make a copy of the data and so adjust to different device speeds
  - with a buffered write(), bytes are written into a memory buffer and the process can continue quite soon
  - e.g. in Linux, all writes onto common filesystems go straight to the memory cache, and the data are flushed to disk asynchronously by the kernel
    - by the way, certain applications which implement their own memory caching (e.g. complex DB engines) can by-pass the OS cashing
    - open() semantics allow to do that via the O_DIRECT open flag
- buffered I/O is useful in case of burst writes, but almost useless when writes are sustained for long times
- for buffered reads(), the buffer cache can result useful to implement "read-ahead" mechanisms, as well as to reduce the number of physical accesses to the device in cases where the required file page is already in the cache

Various I/O models

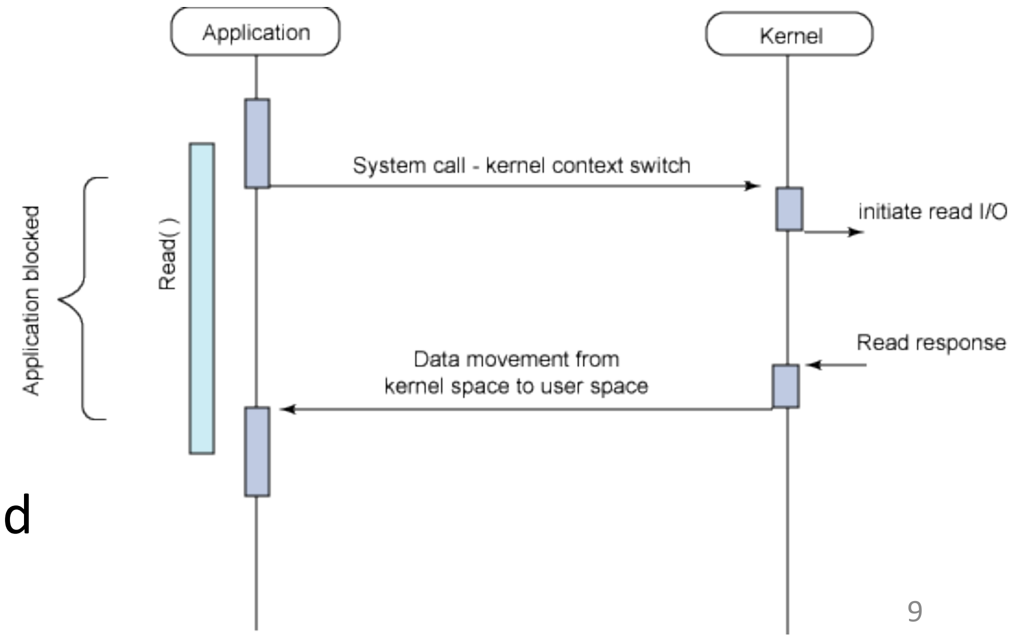
- Schematically, we can identify two orthogonal dimensions
  - **synchronous** and **asynchronous**
  - **blocking** and **non-blocking**
- Each of these I/O models has usage patterns that are advantageous for particular applications
- Understanding which to use is matter of experience in evaluating pros and cons for the specific problem to be solved
  - The design should take into account the trade off between performance gain and increased complexity and readability of the code



	Blocking	Non-blocking
Synchronous	Read/write	Read/wirte (O_NONBLOCK)
Asynchronous	i/O multiplexing (select/poll)	AIO

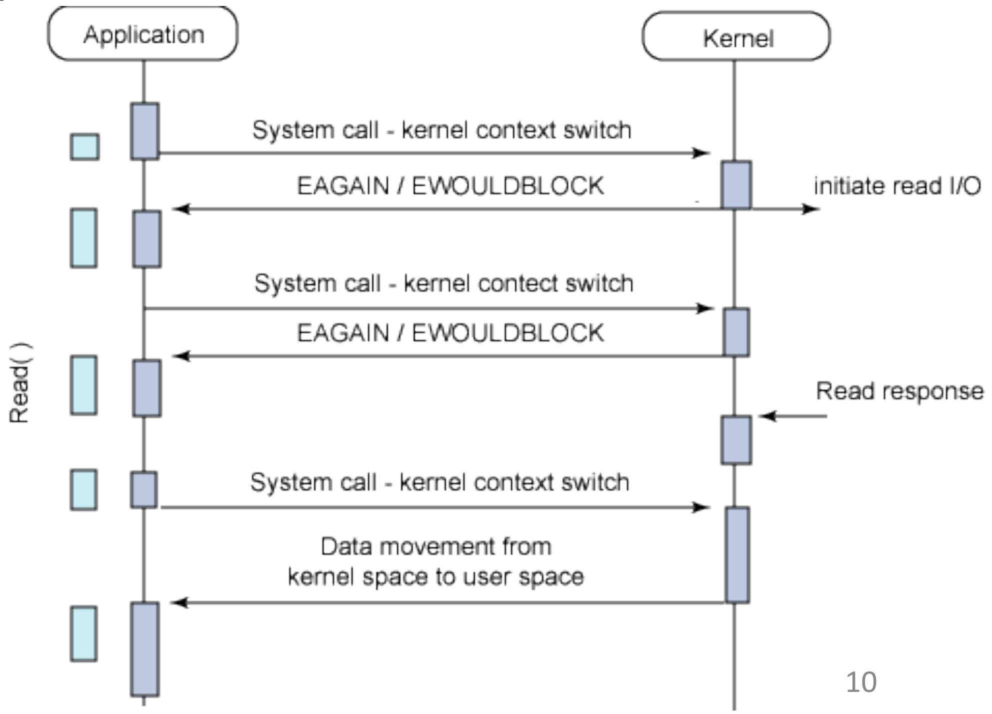
Synchronous blocking I/O

- Simplest approach, and most commonly used
  - application performs a system call and blocks until I/O operation is completed (either successfully or not)
- In this state the application does not load the CPU and simply awaits the response
  - From a computational point of view this is a dead time → depending on the application it can be large or even negligible
- Let's make the example of a read call
  - read system call is invoked and **context switches to the kernel**
  - read is then initiated
  - when finished, **kernel moves the data to the user** space and context is returned to the application


Application
Kernel
System call - kernel context switch
initiate read I/O
Read( )
Application blocked
Read response
Data movement from kernel space to user space

Synchronous non-blocking I/O

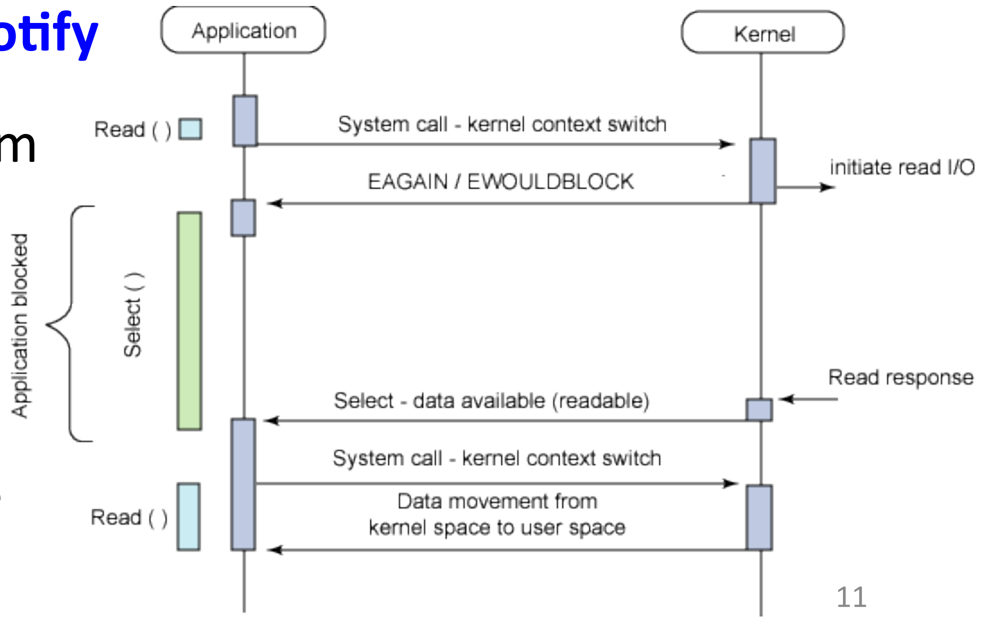
- Kind of a variant of synchronous blocking I/O
- In this approach, instead of completing an I/O immediately, a read may return an error code as the command could not be immediately satisfied
  - The application can decide to do something else instead of waiting for I/O and retry later
- By the way, the application can make several calls to await completion
  - can be inefficient since **increases the number of context switches** between kernel and user spaces
  - **latency in the I/O** because the application cannot know when data is ready



Application
Kernel
System call - kernel context switch
EAGAIN / EWOULDBLOCK
initiate read I/O
System call - kernel contect switch
EAGAIN / EWOULDBLOCK
Read( )
Read response
System call - kernel context switch
Data movement from kernel space to user space

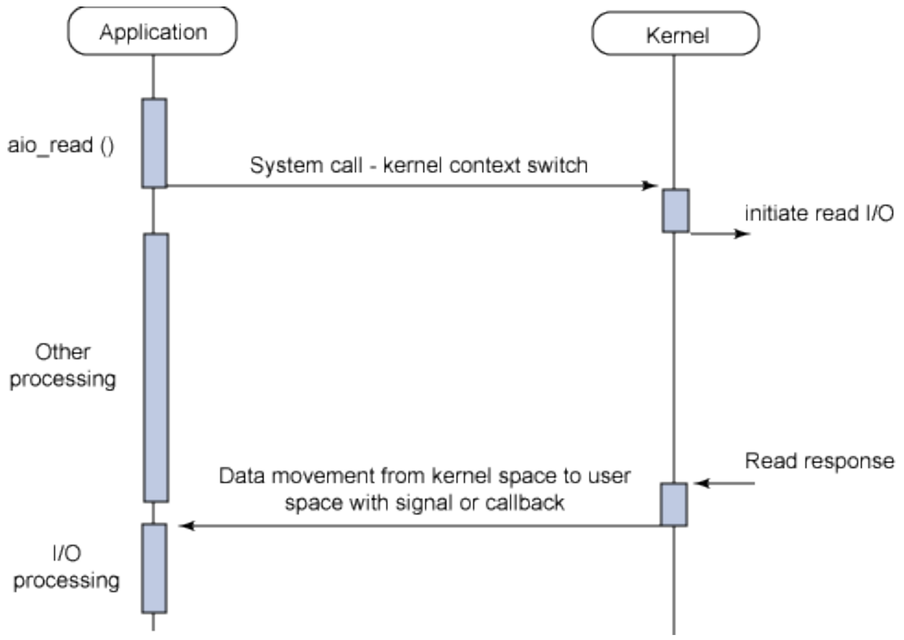
Asynchronous blocking I/O

- In this approach the I/O is initiated as non-blocking but the notification is received in a blocking fashion by means of the select() system call
  - The select() system call is able to determine if a given file descriptor is ready for I/O
- What is the convenience with respect to the simpler and well established synchronous I/O then?
  - the select() call is thought to **notify not only the activity on one file descriptor**, but a list of them
  - as soon as **one or more of the many file descriptors** in the list passed to the select() call **is ready** to be read, the call unblocks and the application can issue the read call, retrieve and process the data


Application
Kernel
Read ( )
System call - kernel context switch
initiate read I/O
EAGAIN / EWOULDBLOCK
Application blocked
Select ( )
Read response
Select - data available (readable)
System call - kernel context switch
Read ( )
Data movement from kernel space to user space

Asynchronous non-blocking I/O (AIO)

- With this approach, on which we will focus later, **data processing and I/O are fully overlapped**
  - In our usual read example, the read() call returns immediately
  - the application can perform computation while the actual read is executed in background by the kernel
- So far nothing new, but the novelty is that when the read response arrives, a **signal or a thread-based callback** is generated to complete the I/O
  - the application has not to loose time in checking if I/O is ready, since it is the kernel itself taking care of telling the application
  - the CPU can process data whose I/O has already completed and at the same time new I/Os can be initiated



Application
Kernel
aio_read ()
System call - kernel context switch
initiate read I/O
Other processing
Read response
Data movement from kernel space to user space with signal or callback
I/O processing

Summary of I/O models

- Blocking models **imply that the application blocks** when the I/O starts
  - Hence **I/O and processing cannot be executed at the same time**
- The synchronous non-blocking model **allows to overlap I/O and processing**
  - But the application **must repeatedly check the status of the I/O**
- Asynchronous non-blocking I/O (a.k.a. AIO) permits **full overlap of processing and I/O**, by means of asynchronous notifications of I/O completions via signals or callbacks
  - the select() function (used in what we called asynchronous blocking I/O) provides a similar functionality but it still blocks
  - However, it blocks on notifications instead of the I/O call and can manage multiple file descriptors at the same time

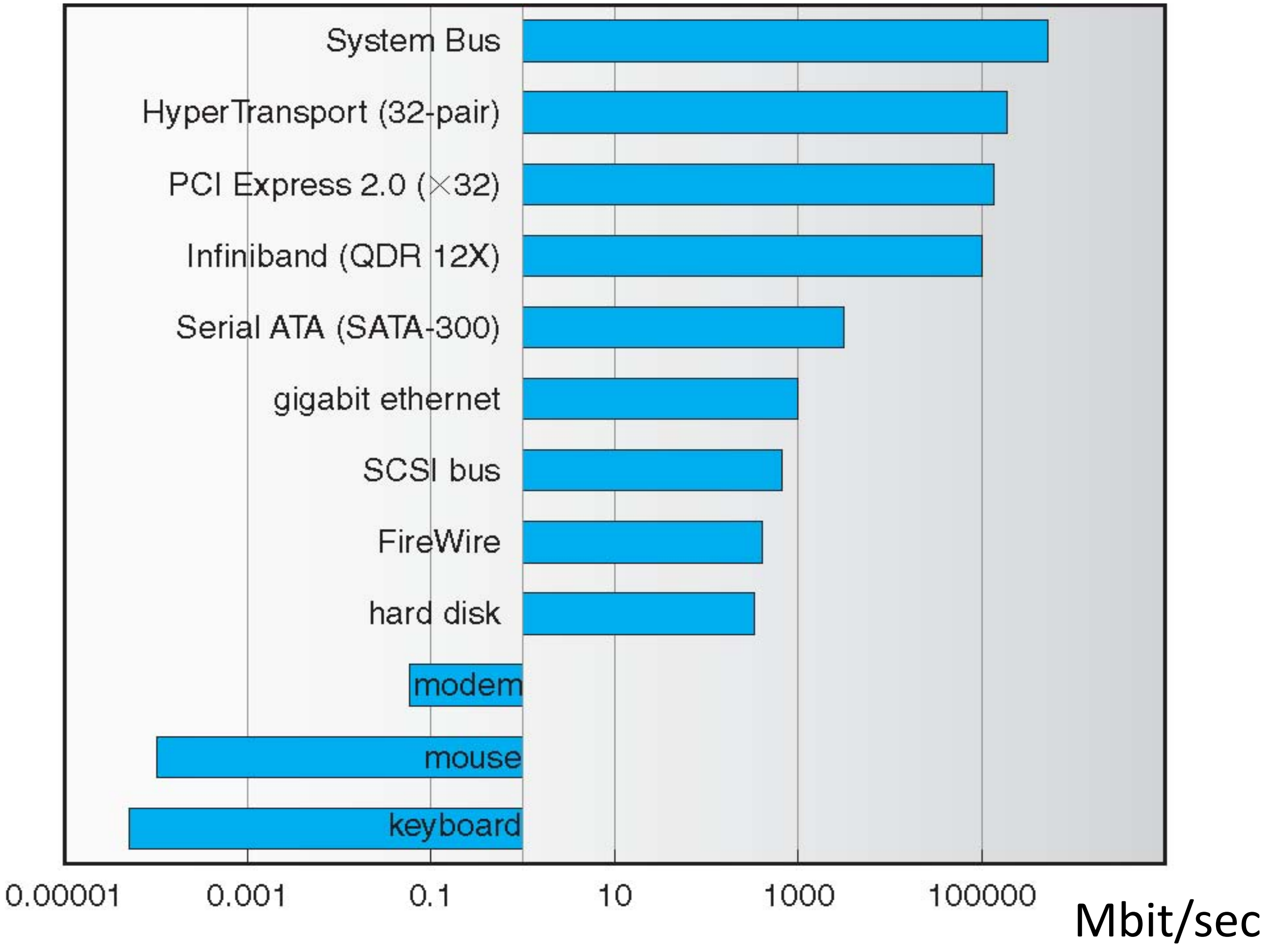
Storage devices

Device Types

Three main device types

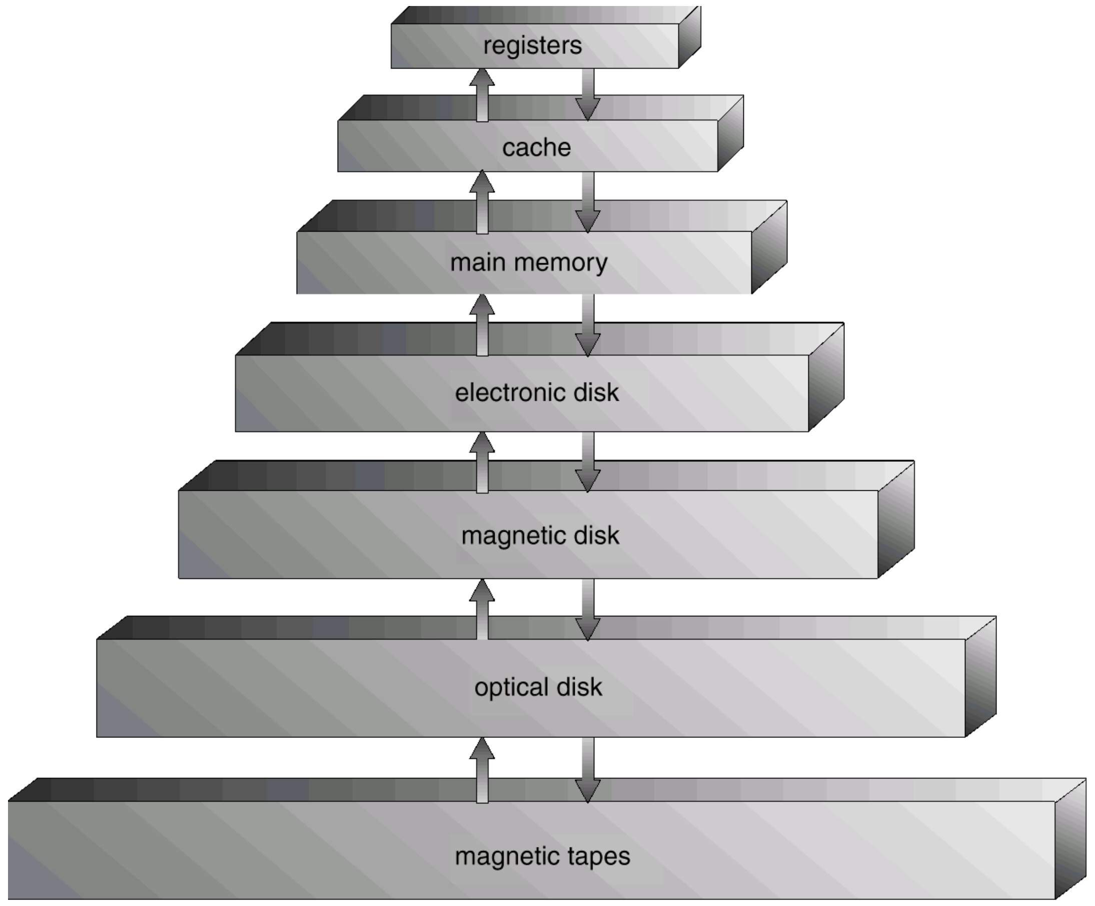
- **Character devices**
  - Basically serial lines
- **Block devices**
  - Mass-storage (e.g. disks, etc.)
  - Commands include read, write, seek
  - The device as a large array of blocks
    - User can read/write only in fixed-size blocks
  - Unlike other devices, block devices support random access
- **Network devices**
  - Network interfaces (e.g., network interface card)
  - Like block-based I/O devices, but each write call either sends the entire block (packet), up to some maximum fixed size, or none
  - On the receiver, the read call returns all the bytes in the block, or none

Transfer Rates



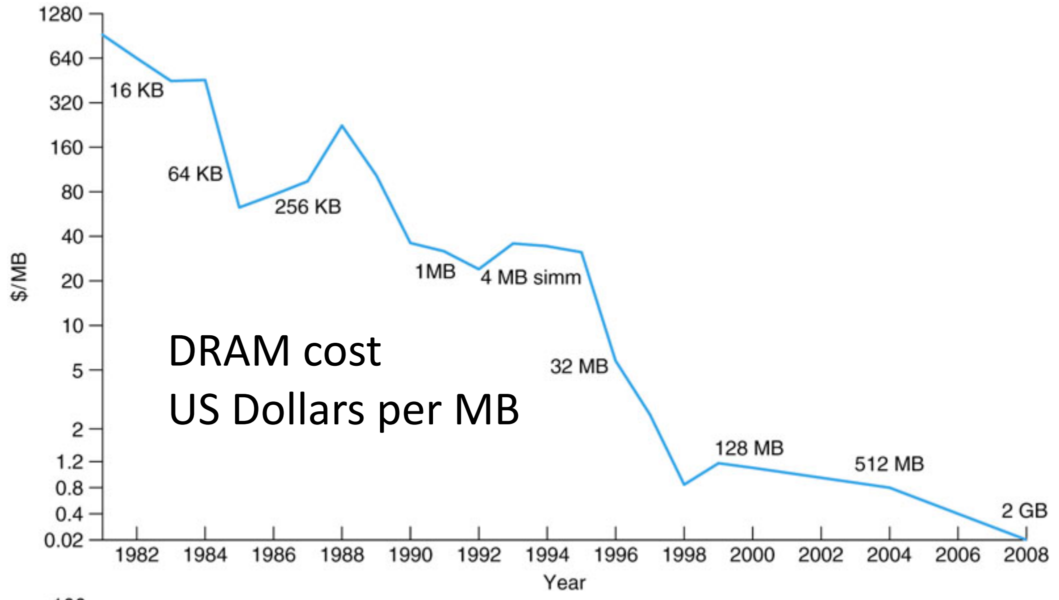
Various data transfer technologies nowadays available span many orders of magnitudes in transfer rate

Storage device hierarchy

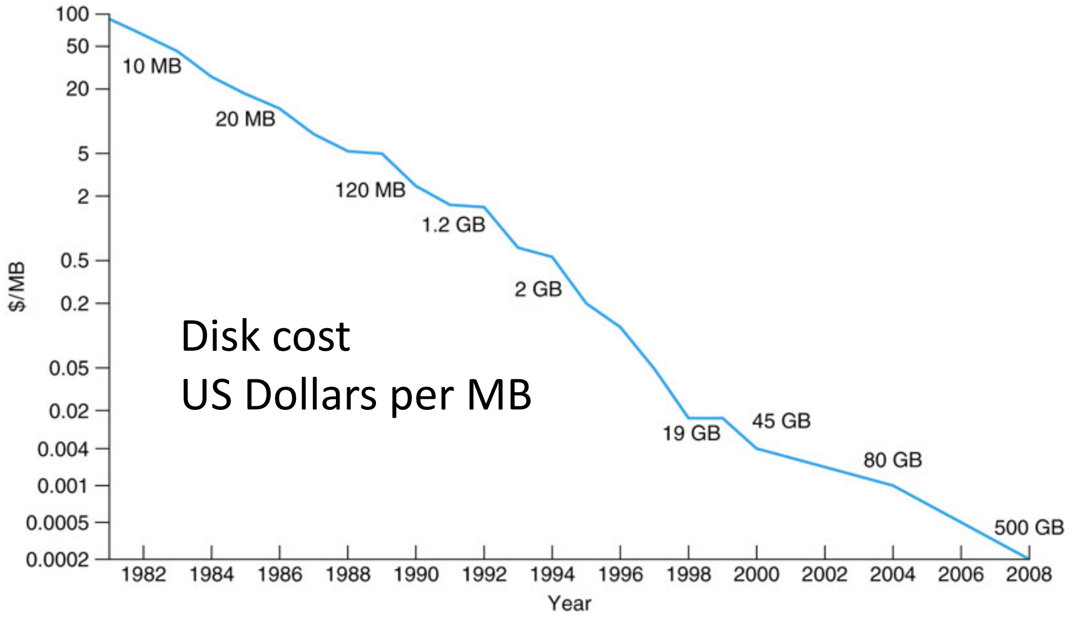


- Main memory is much more expensive than disk storage
- The cheapest tape drives and the cheapest disk drives have had about the same storage capacity over the years
- Tape storage gives a cost savings only when the number of cartridges is considerably larger than the number of drives

Cost of memory and disks (up to 2008)

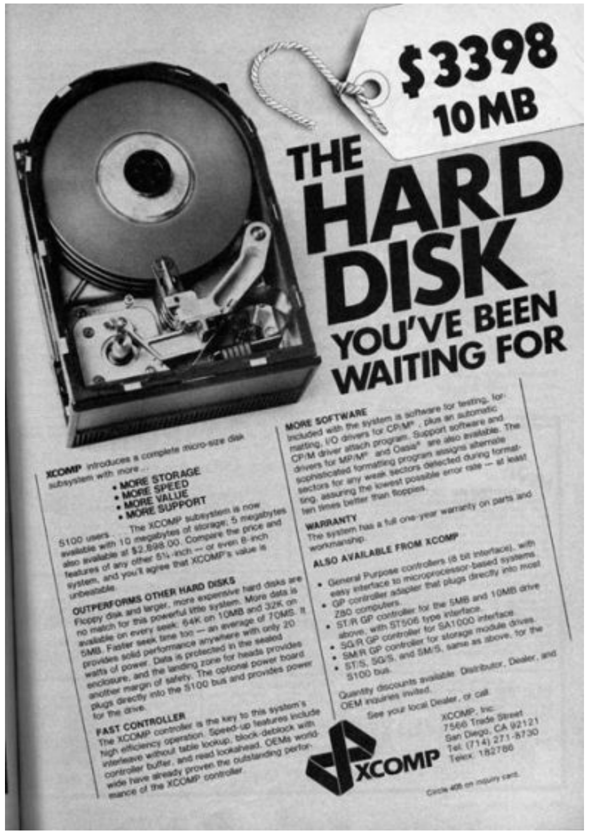


DRAM cost
US Dollars per MB



Disk cost
US Dollars per MB

This was in the 80's... amazing!



$3398
10 MB
THE HARD DISK YOU'VE BEEN WAITING FOR
XCOMP

XCOMP introduces a complete micro-size disk subsystem with more...

- MORE STORAGE
- MORE SPEED
- MORE VALUE
- MORE SUPPORT

S100 users. The XCOMP subsystem is now available with 10 megabytes of storage; 5 megabytes also available at $2,898.00. Compare the price and features of any other 5¼-inch — or even 8-inch system, and you'll agree that XCOMP's value is unbeatable.

OUTPERFORMS OTHER HARD DISKS
Floppy disk and larger, more expensive hard disks are no match for this powerful little system. More data is available on every seek: 64K on 10MB and 32K on 5MB. Faster seek time too — an average of 70MS. It provides solid performance anywhere with only 20 watts of power. Data is protected in the sealed enclosure, and the landing zone for heads provides another margin of safety. The optional power board plugs directly into the S100 bus and provides power for the drive.

FAST CONTROLLER
The XCOMP controller is the key to this system's high efficiency operation. Speed-up features include interleave without table lookup, block-deblock with controller buffer, and read lookahead. OEMs world-wide have already proven the outstanding performance of the XCOMP controller.

MORE SOFTWARE
Included with the system is software for testing, formatting, I/O drivers for CP/M®, plus an automatic CP/M driver attach program. Support software and drivers for MP/M® and Oasis® are also available. The sophisticated formatting program assigns alternate sectors for any weak sectors detected during formatting, assuring the lowest possible error rate — at least ten times better than floppies.

WARRANTY
The system has a full one-year warranty on parts and workmanship.

ALSO AVAILABLE FROM XCOMP

- General Purpose controllers (8 bit interface), with easy interface to microprocessor-based systems.
- GP controller adapter that plugs directly into most Z80 computers.
- ST/R GP controller for the 5MB and 10MB drive above, with ST506 type interface.
- SG/R GP controller for SA1000 interface
- SM/R GP controller for storage module drives.
- ST/S, SG/S, and SM/S, same as above, for the S100 bus.

Quantity discounts available. Distributor, Dealer, and OEM inquiries invited.

See your local Dealer, or call

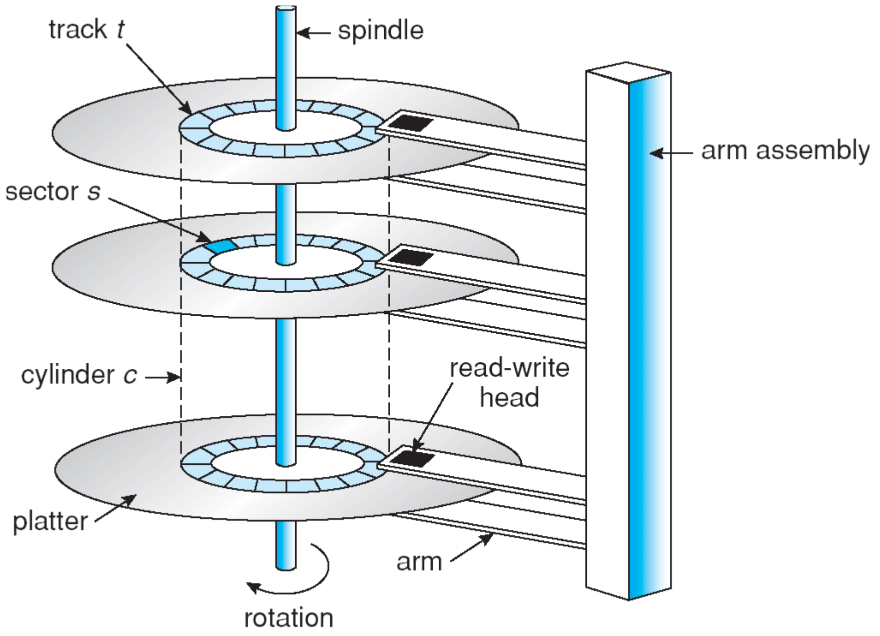
XCOMP, Inc.
7566 Trade Street
San Diego, CA 92121
Tel. (714) 271-8730
Telex: 182786

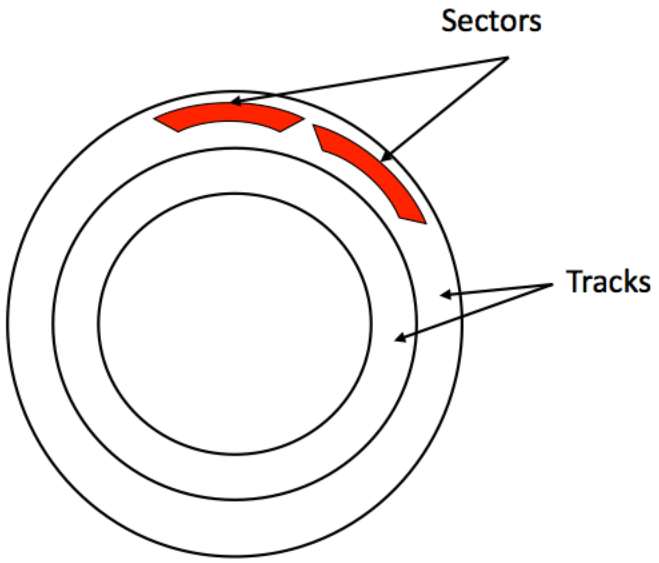
Circle 408 on inquiry card.

Unitary costs span 5-6 orders of magnitude in 25 years!

Still disk space costs a factor 100 less than memory space

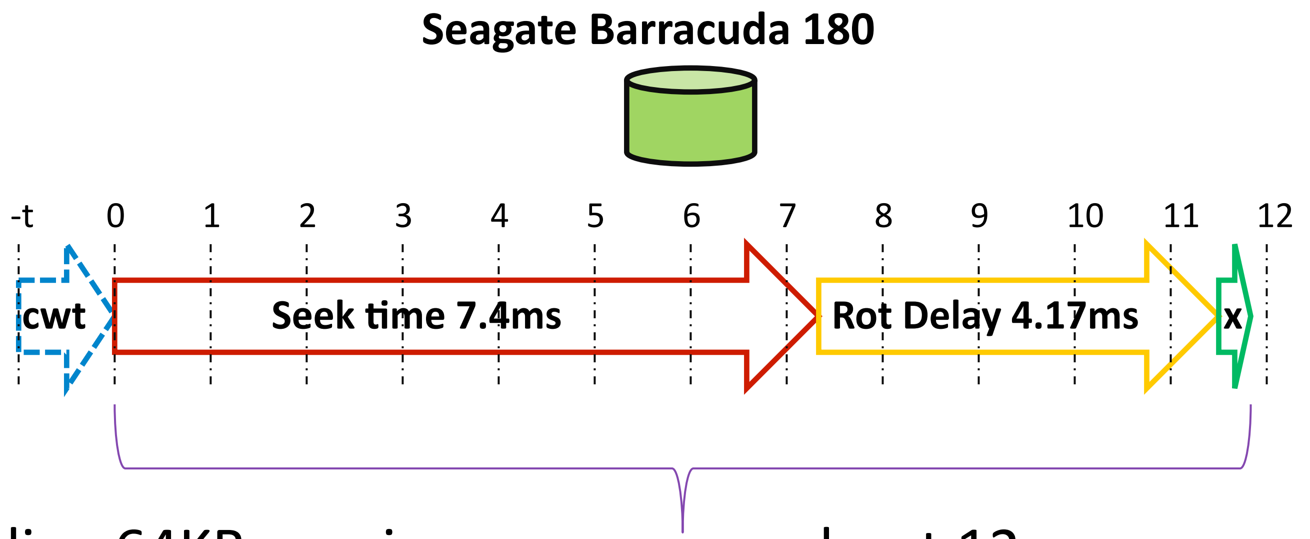
Moving-head Disk Mechanism





- Seek time: time to move the disk head to the desired track
- Rotational delay: time to reach desired sector once head is over the desired track
- Transfer rate: rate data read from/written to disk

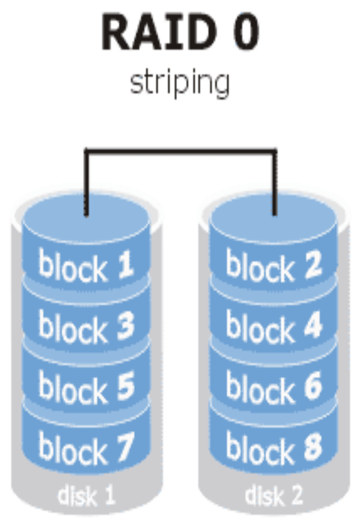
Mechanical movements are slow



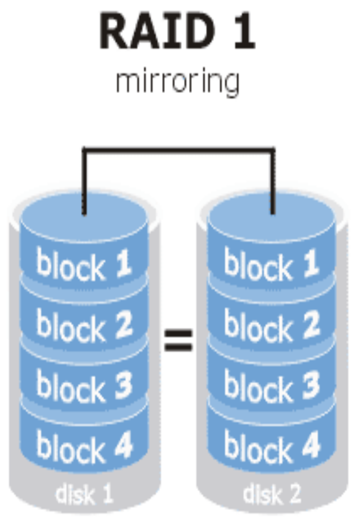
- Reading 64KB requires on average about 12 ms
  - channel wait time + seek time + rotational delay largely dominate
- Actual data transfer (x) occupies disk ~2% of the total time
- Ones needs to read almost 2 MB to achieve 50% channel utilization!
- The faster Cheetah drive accomplishes this 50% faster (~6 ms), but channel utilization drops to less than 1% requiring a read of 3 MB for 50% channel utilization

Redundant Array of Inexpensive Disks (RAID)

- In large or even moderate storage systems, individual disks are not usually addressed, but rather multiple disks are used in cooperation
- In a RAID configuration, a set of physical disk drives are viewed by the OS as a **single logical unit**, as a kind of "super drive" made by the aggregation of **many drives which are accessed in parallel**
  - this lead to an increase of the total space available in a single logical device as well as to improved performance
- **Redundant disk capacity** is used to compensate for the increase in the probability of failure due to the presence of multiple drives
  - The probability that at least one disk in the RAID set fails during the lifetime of the array increases roughly linearly with the number of drives
  - Without redundancy, an array of many disks would be unusable as the risk of loosing data for a single drive failure would be too high

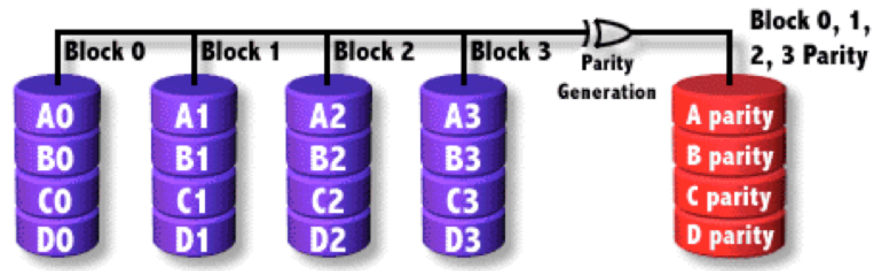


- Simplest configuration without redundancy
- Data blocks are striped across the available disks
- Total storage space across all disks is divided into strips which are mapped round-robin to consecutive disks
- Ideally can be made with an arbitrary number of disks
- RAID configuration with highest performance but lowest data protection



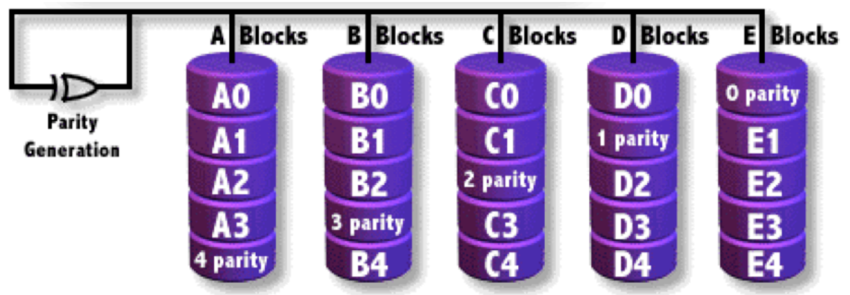
- Simplest configuration with redundancy, achieved by duplicating all the data
- Every disk has a mirror disk that stores exactly the same data
- A read can be serviced by either of the two disks which contains the requested data
- Write request must be done on both disks but can be done in parallel
- Recovery in case of single disk failure is simple but the cost is high (x2)

RAID 4



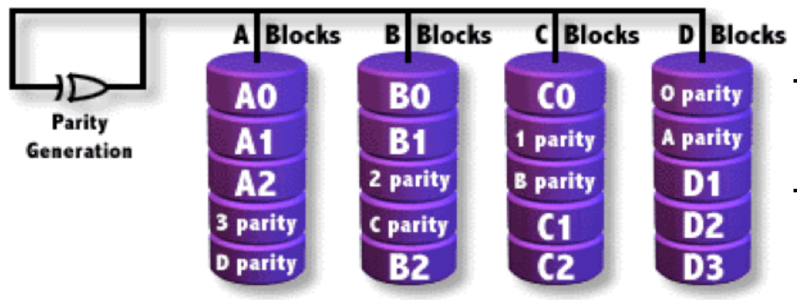
- Uses block-level striping with dedicated parity disk
- Requires minimum of 3 drives to be implemented
- Can survive the loss of one disk, the broken disk can be substituted and rebuild using the information of the remaining disks
- For certain write workloads, parity disk can become a bottleneck
- Total space available is given by the total number of disks minus one

RAID 5



- Very similar to RAID 4, but parity is distributed across all disks with a predefined fixed pattern, overcoming the single parity disk bottleneck
- Also RAID 5 can survive the loss of one disk, which can be rebuilt using the information of the other disks
- Total space available is given by the total number of disks minus one
- Very popular configuration

RAID 6



- As the size of disks increases, the time needed to rebuild a disk in case of a failure increases as well
- During this time, a failure of a second disk would disrupt a RAID 5 array
- For this reason RAID 6, similar to RAID 5 but with double distributed parity, has become increasingly popular during the last few years

How a large high performance storage system looks like in today's real life

Lots of variables influencing the overall I/O performance

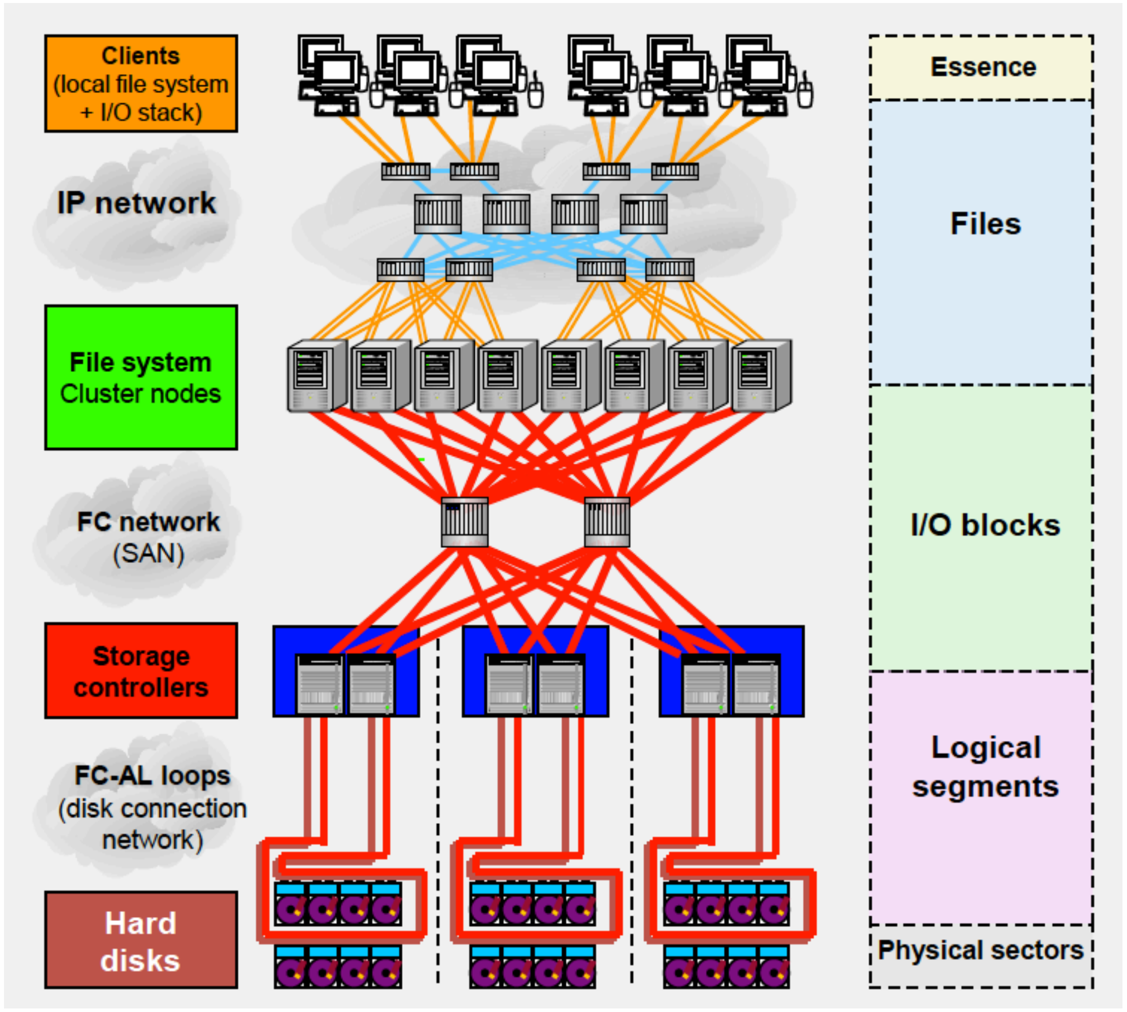
Client hardware
Operating System
Application parameters
Transport protocol parameters

Server hardware
Operating System
File System settings
Transport protocol parameters

Network topology and technology
Network latency
Network protocol parameters

Controller configuration
RAID settings
LUN settings

Number of disks
Disk size
Disk cache

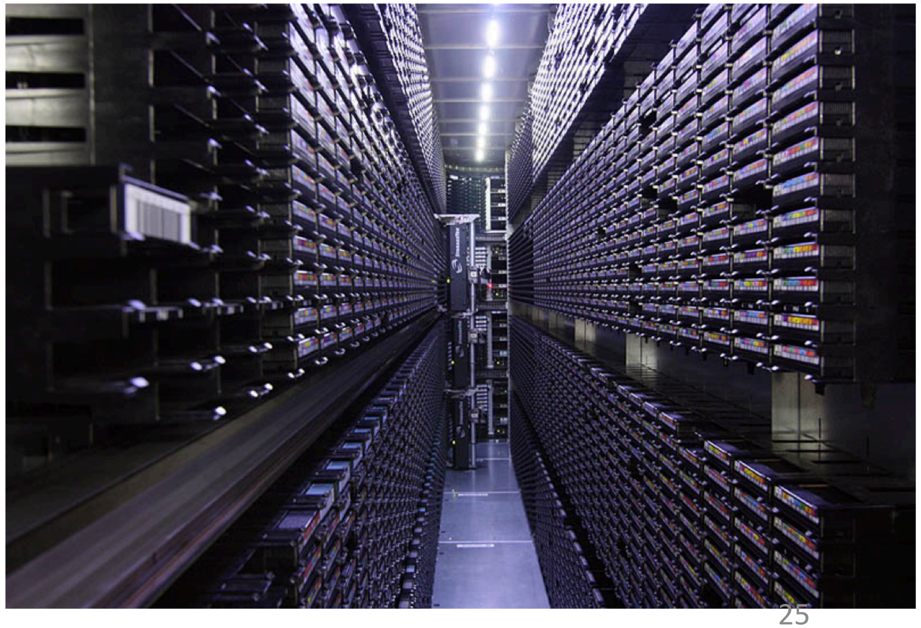


But don't worry: these will never be under your direct control
You have just to hope that your sys admins made it right ☺

Magnetic tape drives

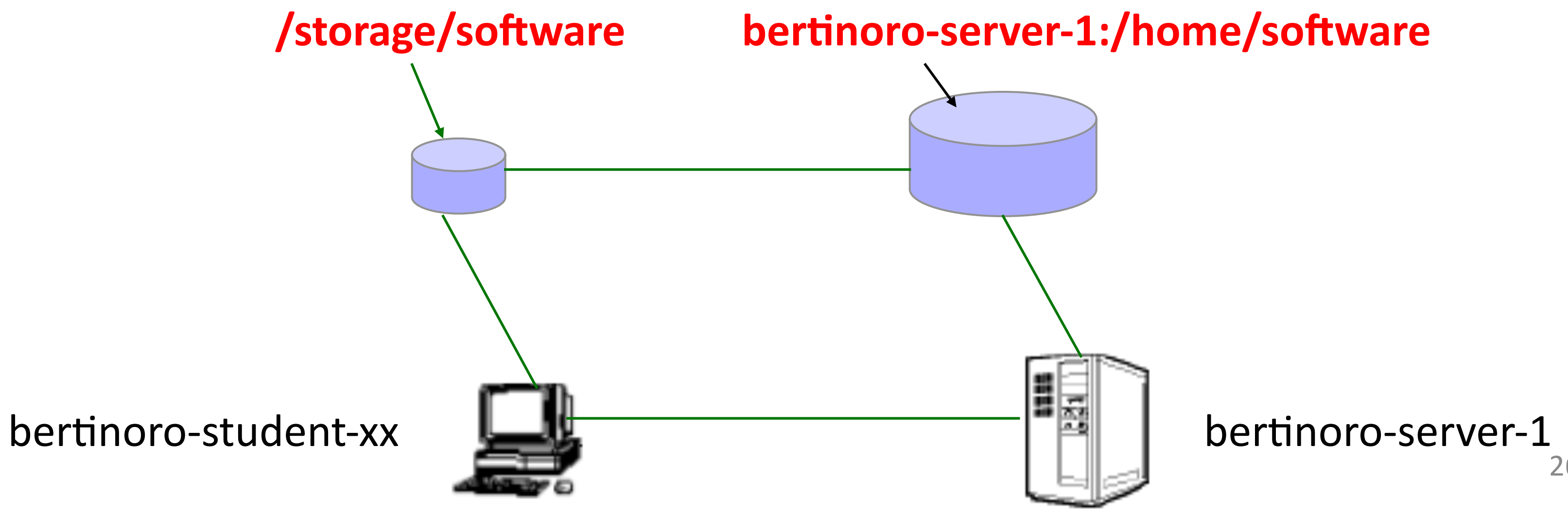
- Relatively permanent and holds large quantities of data, but access time is slow
  - "random" access 1000 times slower than disk
- Mainly used for backup or long term custodial, but also storage of infrequently used data
- Modern cartridges in the 1-5 TB size range

- Tape drives are "append-only" devices
- Once data under head, transfer rates comparable to disk
- In data centres tape drives are hosted in large libraries with mechanical arms which pick up tapes and moves them to drives
  - A modern large library can have tens of tape drives and tens of PetaBytes available nearline



Network File System (NFS)

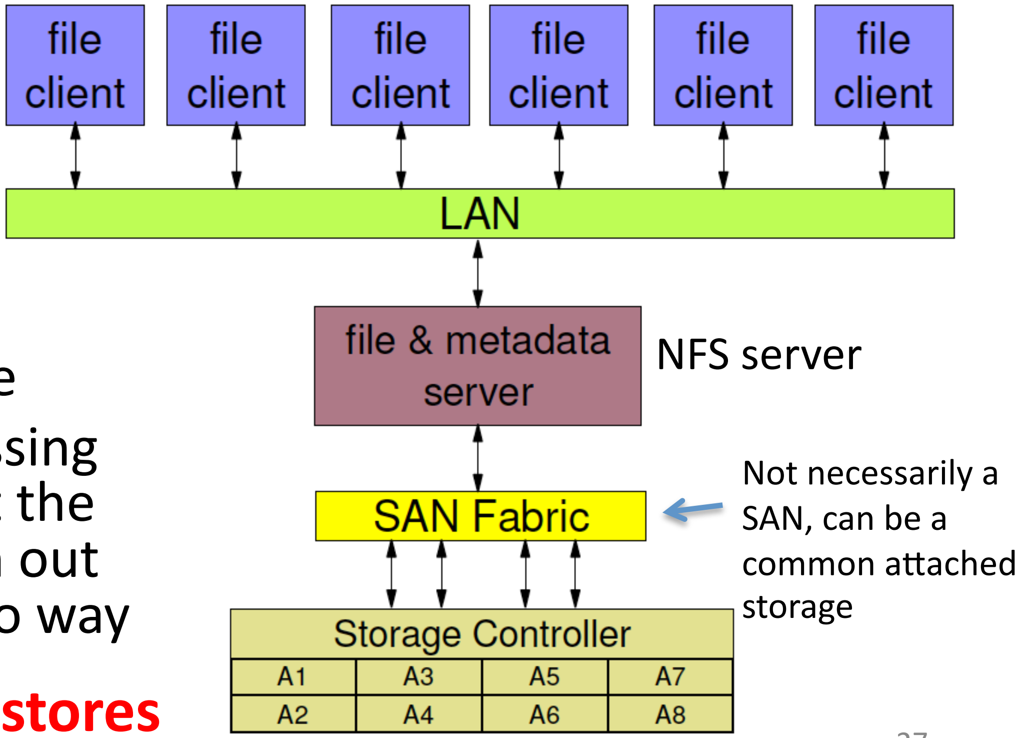
- Disk access from remote nodes via network generally based on TCP/IP over Ethernet
- NFS is a client/server application developed by Sun Microsystems, very common in Unix/Linux environments
- It lets a user view, store and update files on a remote computer as though the files were on the user's local machine.
- The basic function of the NFS server is to allow its file systems to be accessed by any computer on an IP network
- NFS clients access the server files by mounting the servers exported file systems



Limitations of NFS

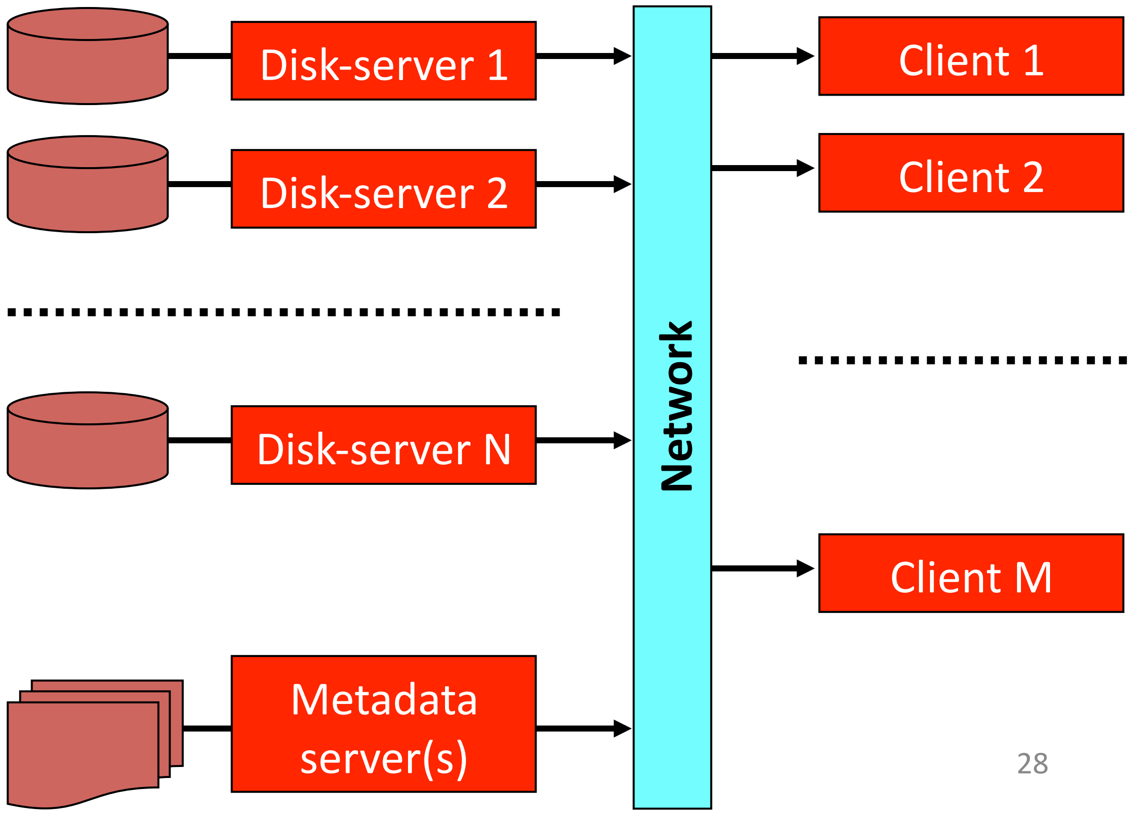
- Traditional NFS configurations limited by "single server” bottleneck
  - a single NFS file server manages both user data and metadata operations
  - this limits performance scaling and presents a single point of failure
  - if NFS server goes down all the processing nodes have to wait until the server comes back into life.
- When the number of computing nodes exceeds the performance capacity of the NFS server, the file system becomes slow and at some point even unusable
- Could add more memory, processing power and network interfaces at the NFS server, but you will soon run out of CPU, memory and PCI slots, no way

Cannot be used for large data stores



Solution: parallel filesystems

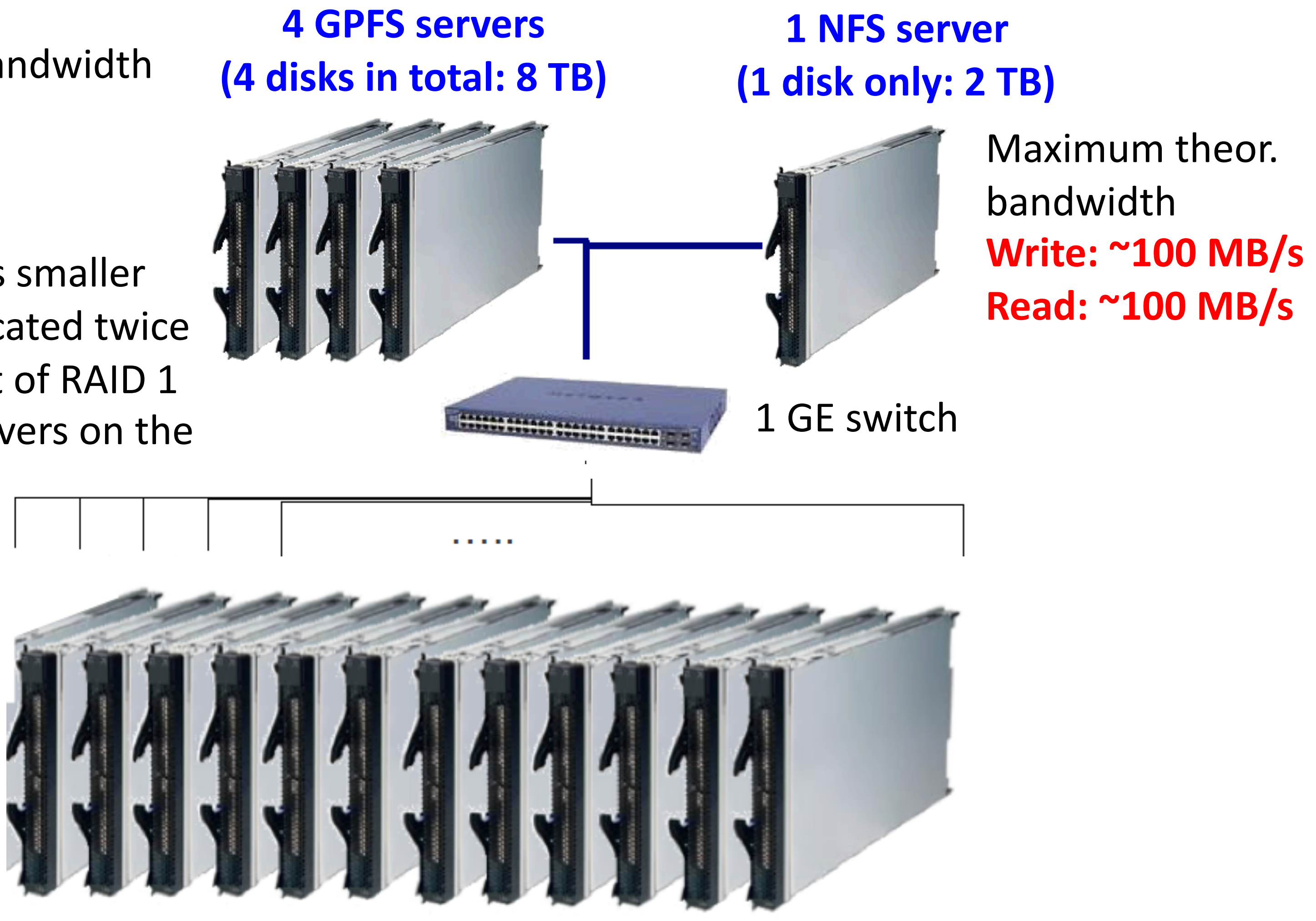
- Instead of having one server exporting a file via one single NFS server, you may have N servers exporting portions of a file to tasks running on multiple processing nodes over the network
  - The aggregate bandwidth exceeds that of a single machine exporting the same file to all processing nodes
- This works much the same way as RAID 0 – file data is striped across all I/O nodes
  - this is how e.g. GPFS (IBM) and Lustre (Oracle) work
- A simpler variant of this model consists in storing the files in their entirety on the various disk-servers in a balanced way, i.e. without striping individual files
  - this is how Castor, dCache, xrootd, and DPM work



Infrastructure at this school

Maximum theor. bandwidth
Write: ~200 MB/s
Read: ~400 MB/s

Write throughput is smaller since files are replicated twice on the servers (sort of RAID 1 across different servers on the network)



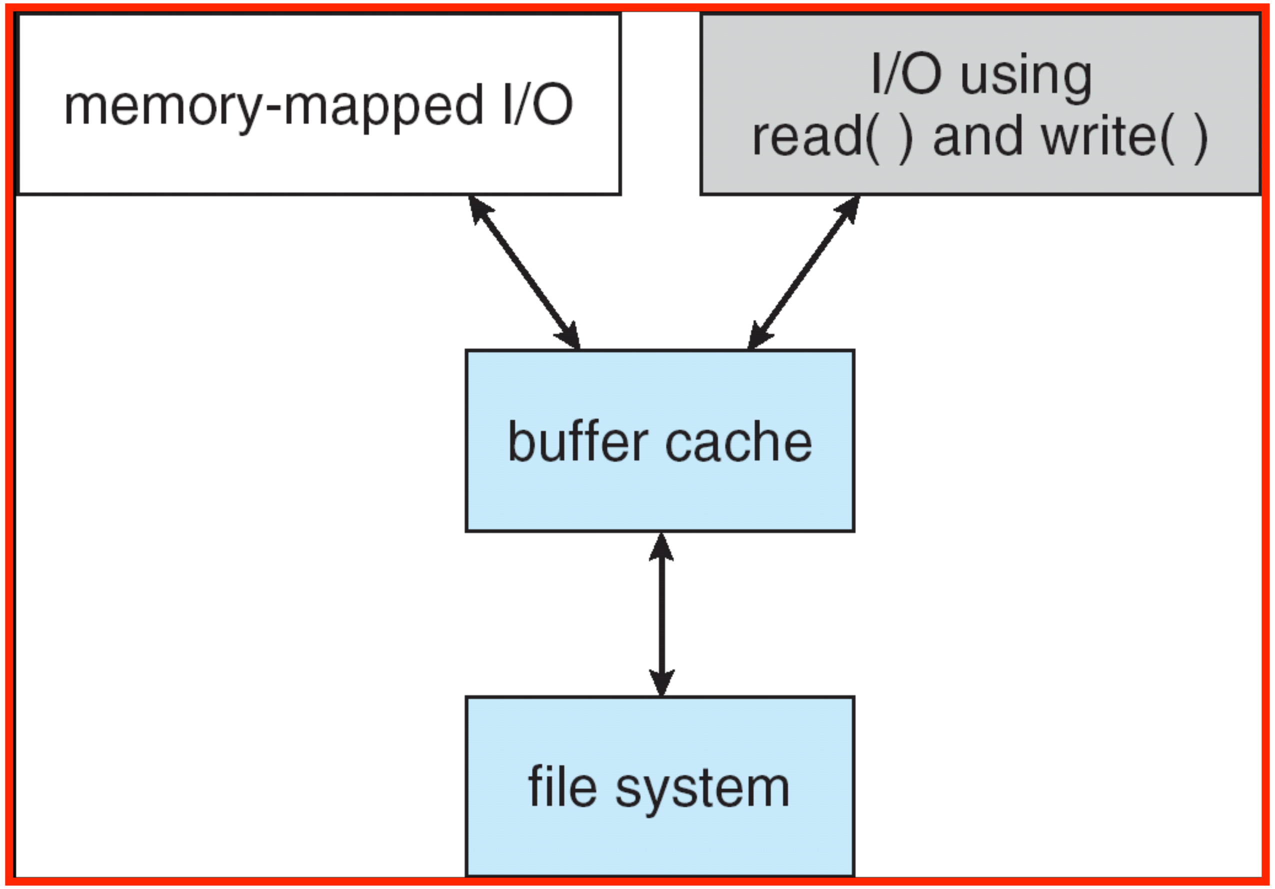
Maximum theor. bandwidth
Write: ~100 MB/s
Read: ~100 MB/s

Bertinoro-student-01…12

Page caching

Linux page cache

- The page cache is a collection of memory pages that normally belong to files
- Every I/O operation on filesystems, either using memory mapped I/O or simple read()/write() calls, proceeds via the cache



Memory pages

- The Linux kernel breaks disk I/O into pages
  - the default page size on most Linux systems is 4 KB
  - the kernel reads and writes disk blocks in and out of memory in 4 KB page sizes
  - you can check the page size of your system by e.g. using the time command in verbose mode and searching for the page size:

```
# /usr/bin/time -v date 2>&1 | grep “Page size”
Page size (bytes): 4096
```

- When an application tries to access a file, the kernel checks if the requested page is cached, and if not it issues a page fault which triggers real I/O from disk
- The pages are kept in memory as long as is possible for performance reasons and after a certain amount of time may take up the majority of the available memory

Lifetime of memory pages

- At some point the free memory gets short and the kernel needs to "free" the pages which have not been accessed recently
- When one sees a low amount of free memory but a large amount of cached memory, this simply means that the system is making good use of its caches
- This is an output taken from the /proc/meminfo where most of the memory is used by the page cache

```
# cat /proc/meminfo
MemTotal: 2075672 kB
MemFree: 52528 kB
Cached: 1766844 kB
 ...
```

- The system here has a total of 2 GB of RAM available on it, with only 52 MB of free RAM and 1.7 GB of pages in the cache
- The page cache can be entirely emptied by issuing with root privileges the following command

```
# echo 3 > /proc/sys/vm/drop_caches
```

Types of memory pages

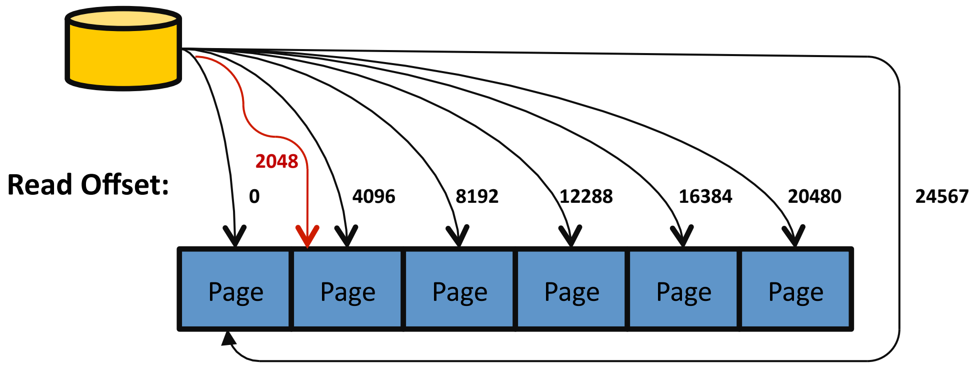
- Read Pages
  - These are read-only pages read from disk and include static files, binaries, and libraries that do not change
- Dirty Pages
  - These are pages of data that have been modified while in memory and must be synced to disk at some point by the pdflush kernel daemon
    - Of course they cannot be removed from memory until the changes are on disk
  - Applications themselves may choose to write dirty pages to disk immediately on request using the fsync() or sync() system calls
- Anonymous Pages
  - These are pages of data that belong to a process, but do not have any file or backing store associated with them, hence they cannot be synchronized back to disk
  - In the event of a memory shortage, the kswapd kernel process writes these to the swap device as temporary storage until more RAM is free ("swapped" pages).

How to remove the pages of a file from the cache

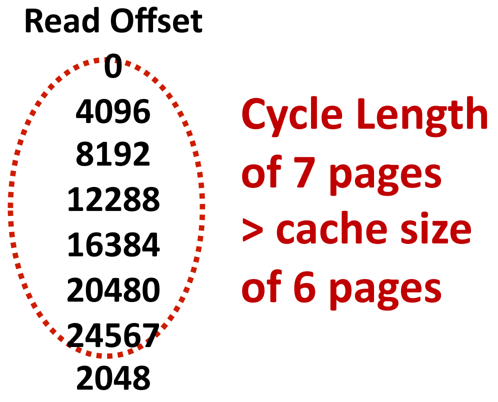
```
int fd;
fd = open(argv[1], O_RDONLY);
if (fd < 0) {
  perror("open");
  return 1;
}
// flushes all data buffers of the file to disk
if (fdatasync(fd)<0) {
  perror("fdatasync");
  return 1;
}
// tell the kernel that we do not need this file using posix_fadvise()
// the kernel will immediately remove the associated pages from the cache
if (posix_fadvise(fd, 0,0,POSIX_FADV_DONTNEED) < 0){
  perror("posix_fadvice");
  return 1;
}
if(close(fd)<0) {
  perror(”close");
  return 1;
}
```

Sequential and random access

- Sequential access is the easiest to handle for filesystem and disk
  - Large disk devices are inherently sequential
  - Kernel can read ahead into the page cache
- However, your application is not running alone on a node
  - application interleaving produces quasi-random I/O
- Each sequential request should be large enough for optimal performance
  - 1-4 MB per request usually OK
- Random I/O is a performance killer for mechanical disks
  - cycle length is important for exploiting the page cache efficiently



Page containing offset 0 replaced
Reading offset 2048 requires re-read!



NB: extreme example, real life caches are much larger than this

Practical demonstration #1

- In your home directory you can find a directory io_exercise

```
# cd ~/io_exercise/demo1
```

- Open with your favorite editor the file README.txt and follow the instructions therein

I/O monitoring tools

I/O monitoring tools: `top` ☺

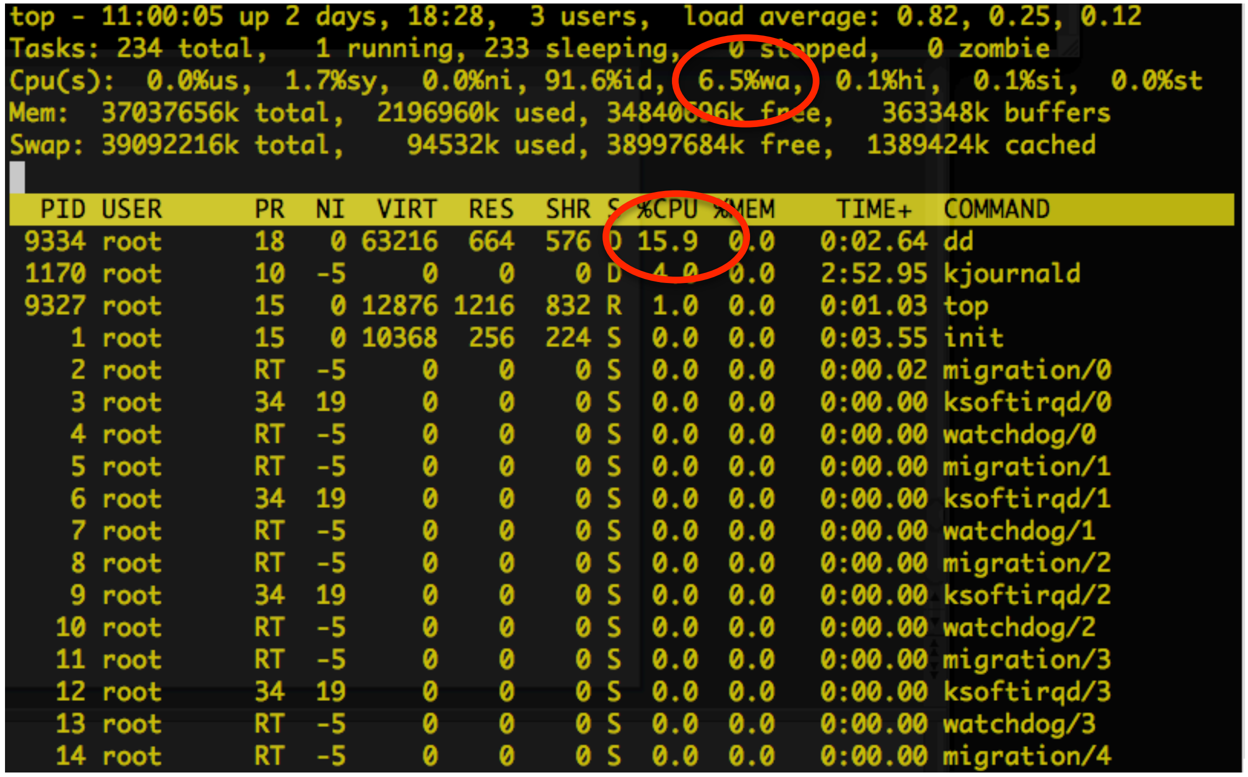
- A tool for very first screening is the old uncle `top`

```
# dd if=/dev/zero of=test.1.dat bs=64k count=163840 oflag=sync
```

This `dd` command writes a zero'ed file of 10 GB, automatically sync'ing to disk (to avoid caching effects)

`top` shows that the CPU load of the dd process is <<100% (15.9% in this snapshot) → the disk subsystem is saturating, i.e. there is an I/O bottleneck

It is also apparent that the fraction of I/O wait is not negligible (6.5%)



```
top - 11:00:05 up 2 days, 18:28,  3 users,  load average: 0.82, 0.25, 0.12
Tasks: 234 total,   1 running, 233 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  1.7%sy,  0.0%ni, 91.6%id,  6.5%wa,  0.1%hi,  0.1%si,  0.0%st
Mem:  37037656k total,  2196960k used, 34840696k free,   363348k buffers
Swap: 39092216k total,    94532k used, 38997684k free,  1389424k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 9334 root      18   0 63216  664  576 D 15.9  0.0   0:02.64 dd
 1170 root      10  -5     0    0    0 D  4.0  0.0   2:52.95 kjournald
 9327 root      15   0 12876 1216  832 R  1.0  0.0   0:01.03 top
    1 root      15   0 10368  256  224 S  0.0  0.0   0:03.55 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:00.02 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    4 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/1
    6 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/1
    7 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/1
    8 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/2
    9 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/2
   10 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/2
   11 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/3
   12 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/3
   13 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/3
   14 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 migration/4
```

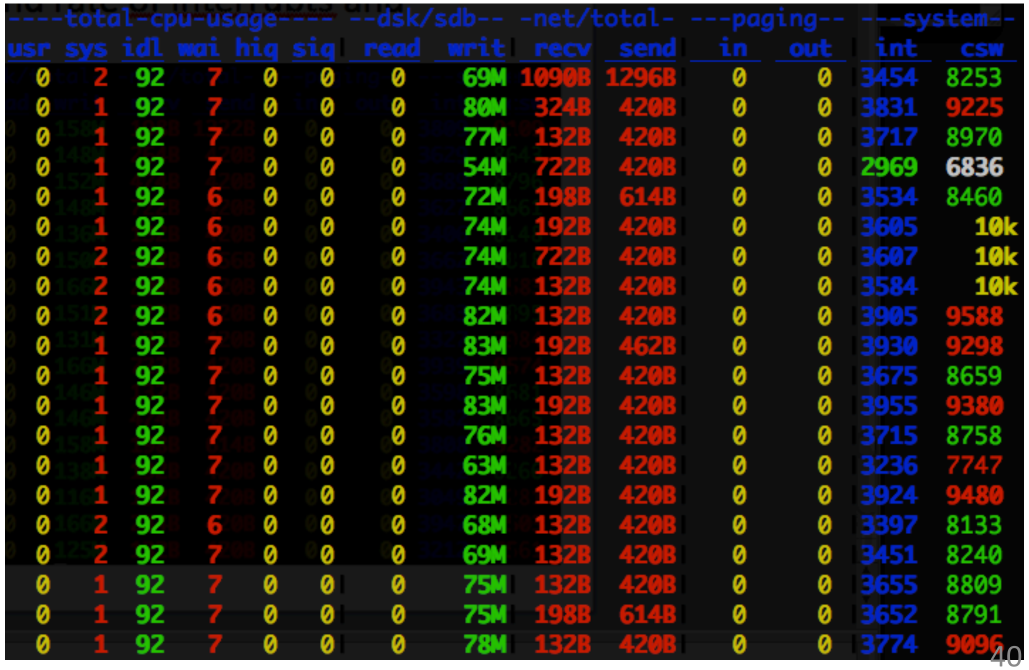
Already from such a first screening you can understand what is going on

I/O monitoring tools: `dstat`

- `dstat` is a widely used utility, similar to `vmstat` but more complete and versatile
- It allows to monitor the time evolution of the disk and network I/O with given time granularity
  - It also dumps other useful information, such as CPU cycles spent in user, system, wait, hard and soft IRQ's, and rate of interrupts and context switches

This snapshot is taken during the same `dd` comand of the previous slide (each row is 1s stat)

Note that the machine has 12 cores, hence 7% of global I/O wait time means almost 100% of one core!



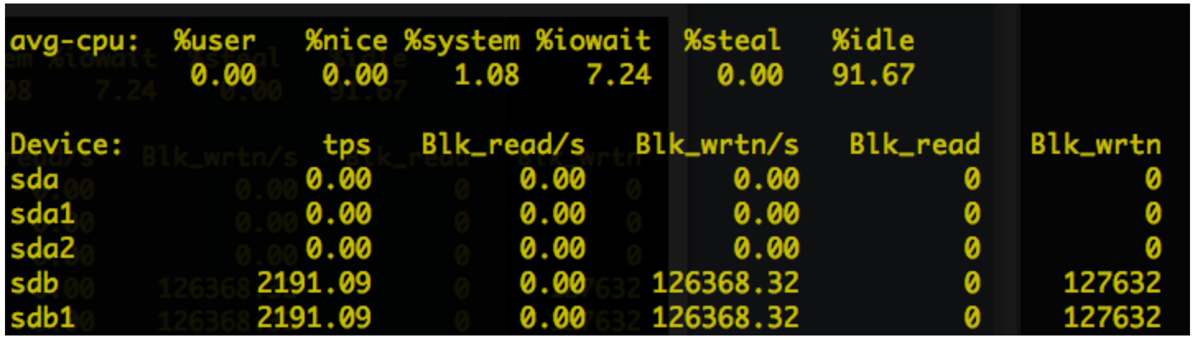
```
----total-cpu-usage---- --dsk/sdb-- -net/total- ---paging-- ---system--
usr sys idl wai hiq siq| read  writ| recv  send|  in   out | int   csw
  0   2  92   7   0   0|   0    69M|1090B 1296B|   0     0 |3454  8253
  0   1  92   7   0   0|   0    80M| 324B  420B|   0     0 |3831  9225
  0   1  92   7   0   0|   0    77M| 132B  420B|   0     0 |3717  8970
  0   1  92   7   0   0|   0    54M| 722B  420B|   0     0 |2969  6836
  0   1  92   6   0   0|   0    72M| 198B  614B|   0     0 |3534  8460
  0   1  92   6   0   0|   0    74M| 192B  420B|   0     0 |3605   10k
  0   2  92   6   0   0|   0    74M| 722B  420B|   0     0 |3607   10k
  0   2  92   6   0   0|   0    74M| 132B  420B|   0     0 |3584   10k
  0   2  92   6   0   0|   0    82M| 132B  420B|   0     0 |3905  9588
  0   1  92   7   0   0|   0    83M| 192B  462B|   0     0 |3930  9298
  0   1  92   7   0   0|   0    75M| 132B  420B|   0     0 |3675  8659
  0   1  92   7   0   0|   0    83M| 192B  420B|   0     0 |3955  9380
  0   1  92   7   0   0|   0    76M| 132B  420B|   0     0 |3715  8758
  0   1  92   7   0   0|   0    63M| 132B  420B|   0     0 |3236  7747
  0   1  92   7   0   0|   0    82M| 192B  420B|   0     0 |3924  9480
  0   2  92   6   0   0|   0    68M| 132B  420B|   0     0 |3397  8133
  0   2  92   7   0   0|   0    69M| 132B  420B|   0     0 |3451  8240
  0   1  92   7   0   0|   0    75M| 132B  420B|   0     0 |3655  8809
  0   1  92   7   0   0|   0    75M| 198B  614B|   0     0 |3652  8791
  0   1  92   7   0   0|   0    78M| 132B  420B|   0     0 |3774  9096
```

I/O monitoring tools: `iostat`

- `iostat` gives the possibility to disentangle traffic of the various disks
- It is useful e.g. on systems with more than one disk attached
  - on large systems with with storage area networks there might be hundreds of disks accessible by a single node

Again, the snapshot is taken during the same `dd` command of the previous slides

In this example the I/O is expressed in terms of blocks written per second (a block is defined as 512 bytes)



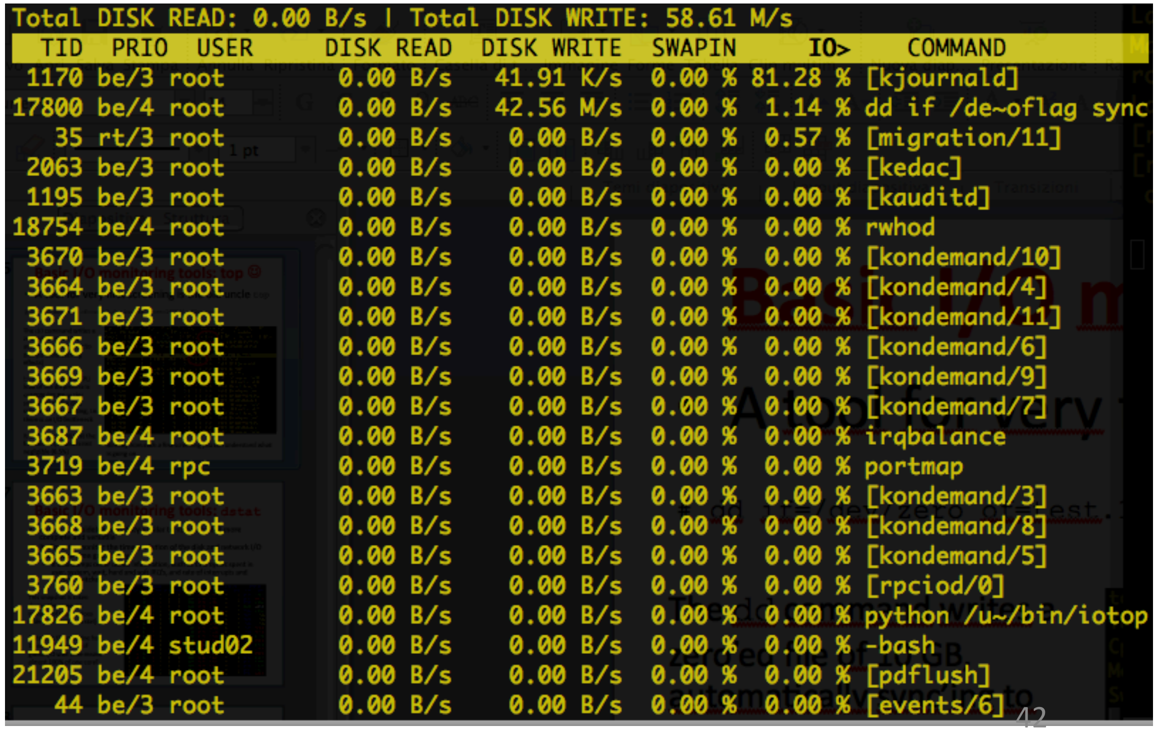
```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           0.00    0.00    1.08    7.24    0.00   91.67

Device:            tps   Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
sda               0.00         0.00         0.00          0          0
sda1              0.00         0.00         0.00          0          0
sda2              0.00         0.00         0.00          0          0
sdb            2191.09         0.00    126368.32          0     127632
sdb1           2191.09         0.00    126368.32          0     127632
```

The I/O writes are being issued on device /dev/sdb, in particular on partition sdb1

I/O monitoring tools: `iotop`

- Tools like dstat and iostat are able to monitor machine wide disk I/O performance
  - You cannot distinguish which process is actually loading your machine
- In certain situations, it is very useful to have single process I/O monitoring granularity
  - e.g. the machine where your processes run are not of your exclusive use, hence there might be other processes which complicate the understanding of what your processes are doing
- iotop provides various disk I/O information on a process-by-process basis



```
Total DISK READ: 0.00 B/s | Total DISK WRITE: 58.61 M/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
 1170 be/3 root      0.00 B/s   41.91 K/s  0.00 % 81.28 % [kjournald]
17800 be/4 root      0.00 B/s   42.56 M/s  0.00 %  1.14 % dd if /de~oflag sync
   35 rt/3 root      0.00 B/s    0.00 B/s  0.00 %  0.57 % [migration/11]
 2063 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kedac]
 1195 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kauditd]
18754 be/4 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % rwhod
 3670 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kondemand/10]
 3664 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kondemand/4]
 3671 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kondemand/11]
 3666 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kondemand/6]
 3669 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kondemand/9]
 3667 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kondemand/7]
 3687 be/4 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % irqbalance
 3719 be/4 rpc       0.00 B/s    0.00 B/s  0.00 %  0.00 % portmap
 3663 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kondemand/3]
 3668 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kondemand/8]
 3665 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [kondemand/5]
 3760 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [rpciod/0]
17826 be/4 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % python /u~/bin/iotop
11949 be/4 stud02    0.00 B/s    0.00 B/s  0.00 %  0.00 % -bash
21205 be/4 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [pdflush]
   44 be/3 root      0.00 B/s    0.00 B/s  0.00 %  0.00 % [events/6]
```

Snapshot is taken during our usual `dd` command

A simple (but powerful) profiler: `strace`

- strace is a powerful utility that can be used to profile any application

it is able to trace every single system call of a process

In this example it is used to profile the time spent by each type of system call

You can also attach to an already running process, and detach by hitting CTRL-C

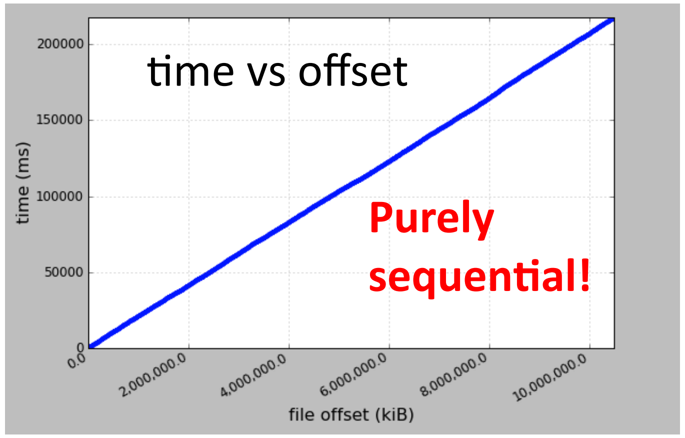
A similar tracer, ltrace, can be used for library calls instead of system calls

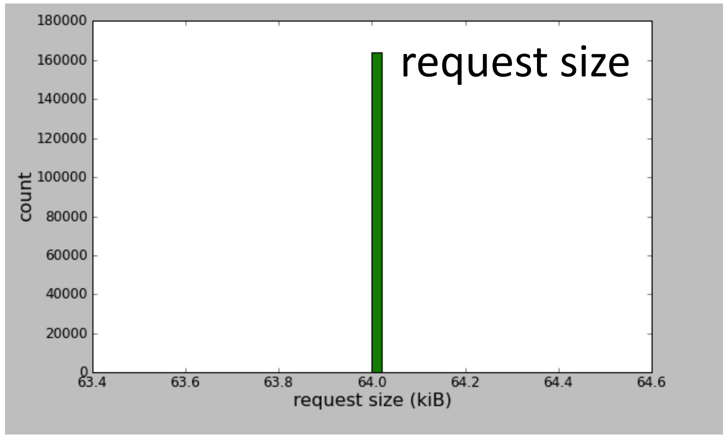
```
# strace -c dd if=/dev/zero of=test.1.dat bs=64k count=163840 oflag=sync
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.80    4.635656         130     35594           write
  0.19    0.009053           0     35598           read
  0.00    0.000013           1        14           mmap
  0.00    0.000000           0         7           open
  0.00    0.000000           0         7           close
  0.00    0.000000           0         5           fstat
  0.00    0.000000           0         1           lseek
  0.00    0.000000           0         7           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         6           rt_sigaction
  0.00    0.000000           0         1           rt_sigprocmask
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           getrlimit
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         1           futex
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         1           clock_gettime
  0.00    0.000000           0         1           set_robust_list
------ ----------- ----------- --------- --------- ----------------
100.00    4.644722                 71252         1 total
```

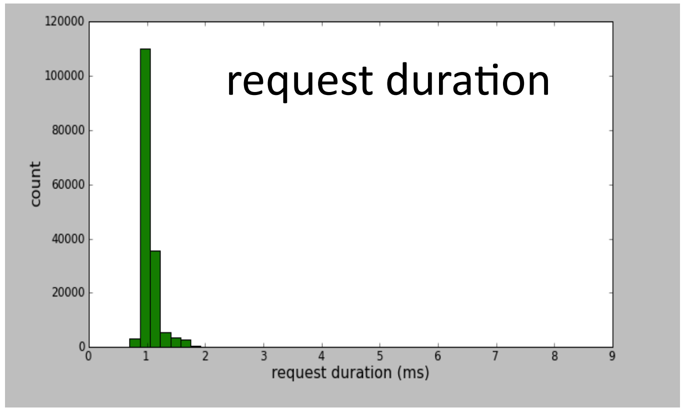
ioprofiler: a graphical tool on top of `strace`

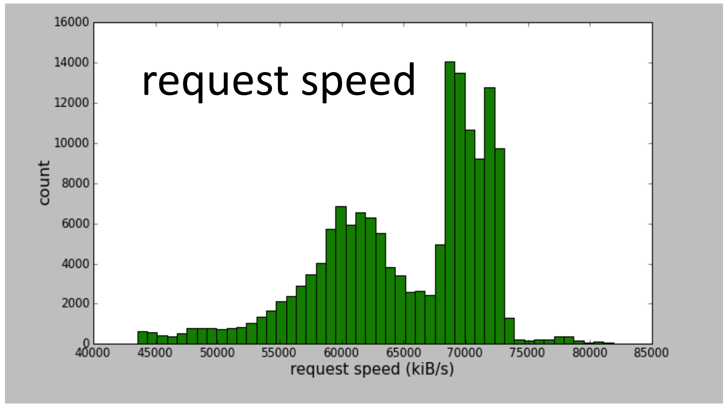
- ioprofiler is a simple graphical tool that takes as input the strace dump of I/O system calls and it is able to represent graphically some useful quantities

```
# strace -q -a1 -s0 -f -tttT –o strace.out –e trace=file,desc,process,socket \
   dd if=/dev/zero of=test.1.dat bs=64k count=163840 oflag=sync
# ioprofiler.py
```





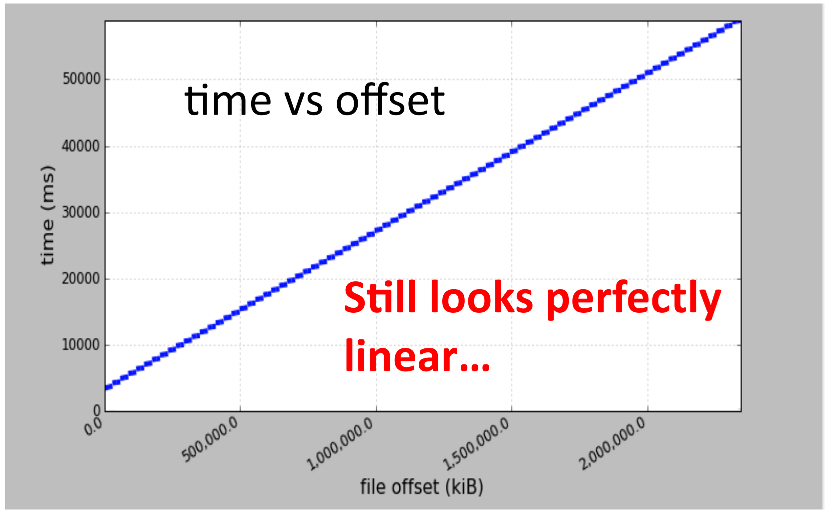


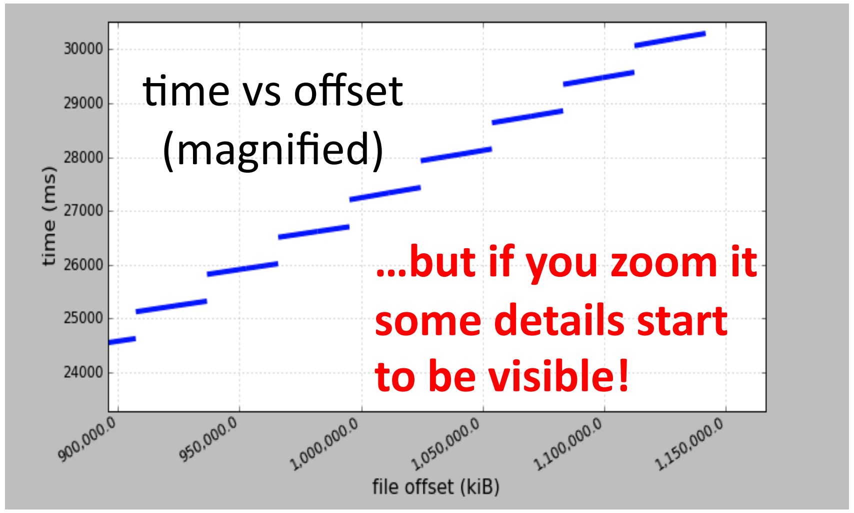


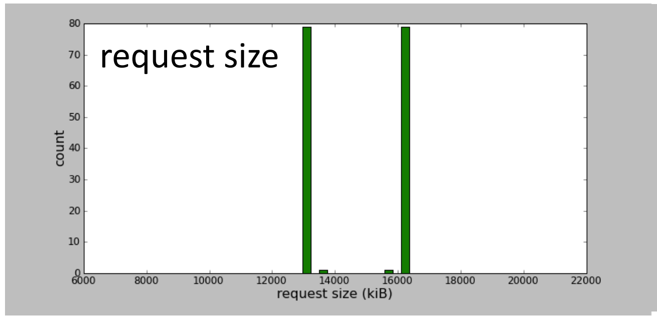
A slightly different pattern

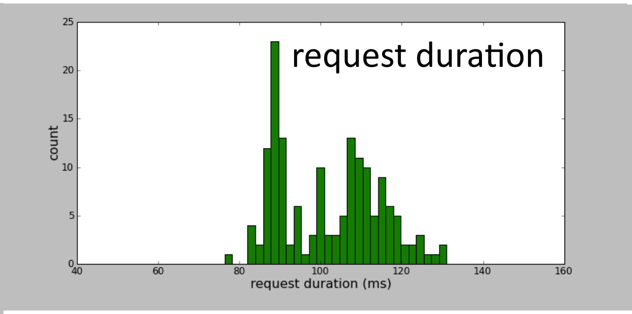
- Obtained using a simple ROOT benchmarking macro reading a TTree from the local disk

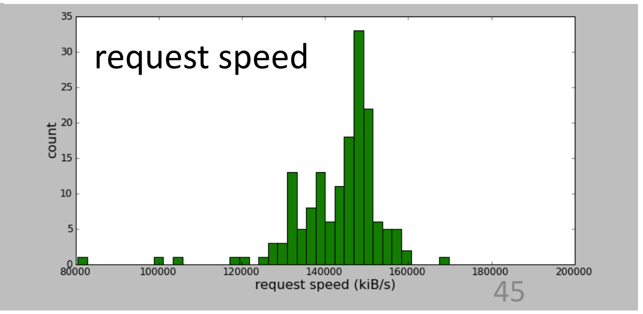
```
# strace -q -a1 -s0 -f -tttT –o strace.out –e trace=file,desc,process,socket \
   root -l BenchTree.C
# ioprofiler.py
```











Practical demonstration #2

```
# cd ~/io_exercise/demo2
```

- Have a look into README.txt

Advanced I/O API

Basic I/O API

- We assume that you are already familiar with basic I/O calls such as
  - open(), close(), read(), write()
- Maybe you are less familiar with
  - lseek()
    - used to move the offset of a given file descriptor, in order to access any arbitrary point within the file
      - e.g. used in random access patterns
  - pread()
    - like a read(), but allows to specify the offset as an argument
      - atomic lseek+read(), useful in multi-threaded programs which access the same file simultaneously from many threads
  - pwrite()
    - analogue to pread(), but for writing
- In the following, we will focus on more advanced I/O techniques, specifically
  - Scatter/Gather I/O, Memory mapped I/O, Asynchronous I/O

The basic I/O API calls are simple, but not always things are as simple as they seem

```
int fd = open(”1GBfile.dat", O_RDONLY);
if (fd < 0) {
  perror("open");
  return 1;
}

struct stat st;
if (fstat(fd, &st)) {
  perror("fstat");
  return 1;
}

long long offset = 0;
long long filesize = st.st_size;
printf("File length: %lld\n", filesize);
char buf[1];
while (offset < filesize) {
  int n = pread(fd, buf, 1, offset);
  if (n <= 0) {
    perror("pread");
    return 1;
  }
  offset += 8192;
}
```

In this extreme example we read 1 byte every 8192 bytes from the local disk

It looks like the OS should read from disk only 131072 bytes out of 1073741824 bytes, but indeed the OS retrieve from disk half of the file

Even if we try to read 1 byte at a time, the OS reads 4096 bytes as a minimum block size

This block size is not universal, but depends on the filesystem in use

To achieve optimal I/O performance you also need to know how the underlying backend (either local or network filesystem) works in conjunction with your workload

Scatter/gather I/O

- Scatter/gather I/O (a.k.a. vectored I/O) is a method of input and output where a single system call can read/write to a vector of buffers from a single data stream in place of a single buffer
  - the standard read and write system calls provide instead linear I/O
- Scatter/gather I/O provides several advantages over linear I/O methods
  - first of all if your data is naturally segmented, vectored I/O allows for intuitive and immediate manipulation
  - a single vectored I/O operation can replace multiple linear I/O operations, drastically reducing the rate of kernel context switches
- In addition to the reduction in the number of system calls, a vectored I/O is internally instrumented at the kernel level with a series of optimizations which improve performance and would be impossible to be implemented in user land
  - you may e.g. think that a process could concatenate the disjoint vectors into a single buffer before writing or decompose the returned buffer into multiple vectors after reading
  - this is a user-space implementation of S/G I/O, but for several reasons it would be not as efficient as what the kernel can do by itself

readv() and writev()

- The readv() and writev() functions behave exactly the same as read() and write() except that multiple buffers are involved

```
// writev() writes at most count segments from the iov array
// into the file descriptor fd
ssize_t readv (int fd, const struct iovec *iov, int count);

// readv() does the same but for reading
ssize_t writev (int fd, const struct iovec *iov, int count);
```

- Each iovec structure describes an independent disjoint buffer, which is called a segment

```
struct iovec {
  void *iov_base;   // pointer to start of buffer
  size_t iov_len;   // size of buffer in bytes
};
```

writev() example

```
struct iovec iov[3];
ssize_t nr;
int fd, i;
char *buf[] = { “This is the third edition of the INFN ESC11 school. The goal of the ”,
                “school is to increase the awareness of the new generations of ”,
                “scientists that will face future challenges in scientific computing.” };
fd = open (”esc11.txt", O_WRONLY | O_CREAT, 0666);
if (fd == -1) {
  perror ("open");
  return 1;
}
// prepare the 3 iovec structures...
for (i = 0; i < 3; i++) {
  iov[i].iov_base = buf[i];
  iov[i].iov_len = strlen (buf[i]);
}
// ...and with a single call, write them out!
nr = writev (fd, iov, 3);
if (nr == -1) {
  perror ("writev");
  return 1;
}
printf ("wrote %d bytes\n", nr);
if (close (fd)) {
  perror ("close");
  return 1;
}
```

Poor man's implementation of writev()

```
ssize_t ret=0;
int i;
for (i = 0; i < count; i++) {
  ssize_t nr;
  nr = write (fd, iov[i].iov_base, io[i].iov_len);
   if (nr == -1) {
     ret = -1;
     break;
   }
   ret += nr;
}
return ret;
```

Logically equivalent implementation by using linear I/O, but by far less efficient

Even more, Linux kernel internally implements all I/O as vectored, and linear I/O is treated as a limit case ov vectored I/O with only one segment

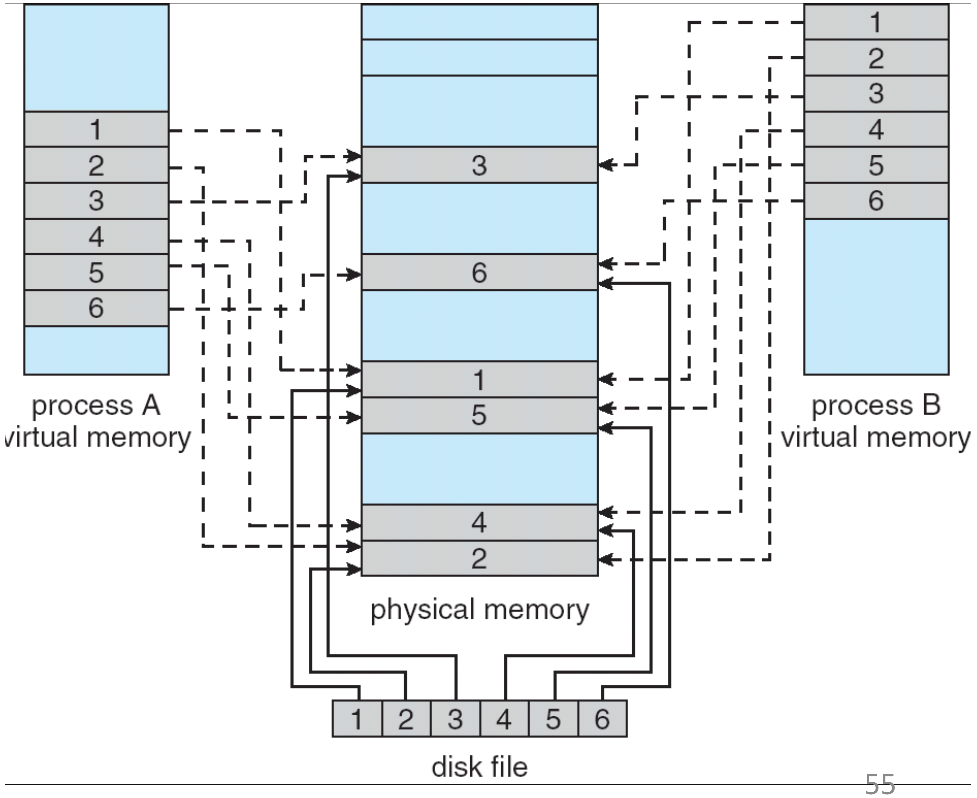
readv() example

Reads back the file written before

```
char buf_1[69], buf_2[63], buf_3[69];
struct iovec iov[3];
ssize_t nr;
int fd, i;
fd = open (”escl1.txt", O_RDONLY);
if (fd == -1) {
  perror ("open");
  return 1;
}
// prepare the iovec structures
iov[0].iov_base = buf_1; iov[0].iov_len = sizeof(buf_1)-1;
iov[1].iov_base = buf_2; iov[1].iov_len = sizeof(buf_2)-1;
iov[2].iov_base = buf_3; iov[2].iov_len = sizeof(buf_3)-1;
// read into the structures with a single call
nr = readv (fd, iov, 3);
if (nr == -1) {
  perror ("readv");
  return 1;
}
for (i = 0; i < 3; i++) printf ("%d: %s", i, (char *) iov[i].iov_base);
if (close (fd)) {
  perror ("close”);
  return 1;
}
```

Memory mapped I/O

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory
- A file is initially read using demand paging
- A page-sized portion of the file is read from the file system into a physical page
- Subsequent reads/writes to/from the file are treated as ordinary memory accesses

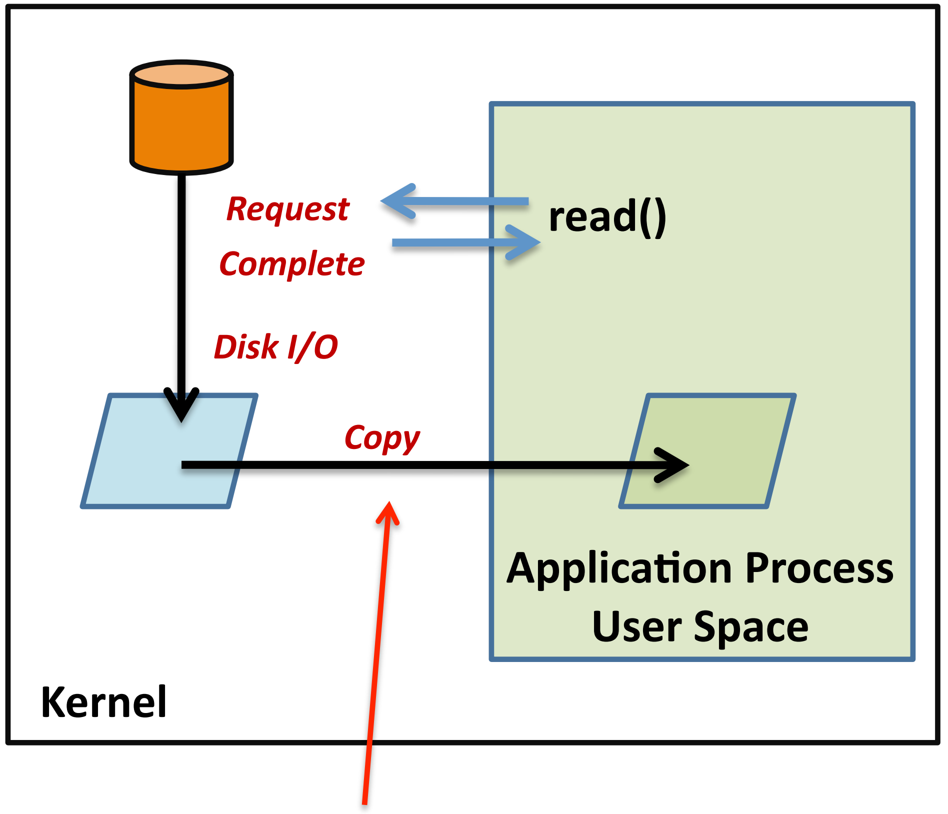


Advantages of memory mapped I/O

- Manipulating files via mmap() has a handful of advantages over the standard read() and write() system calls
  - reading from and writing to a memory-mapped file avoids the data movement from kernel space to user space, and viceversa, that occurs when using the read() or write() system calls (see next slide)
  - reading from and writing to a memory-mapped file does not incur any system call or context switch overhead
    - data access is as simple as accessing memory, no need of calls like lseek()
  - when multiple processes map the same object into memory, the data can be shared amongst all the processes

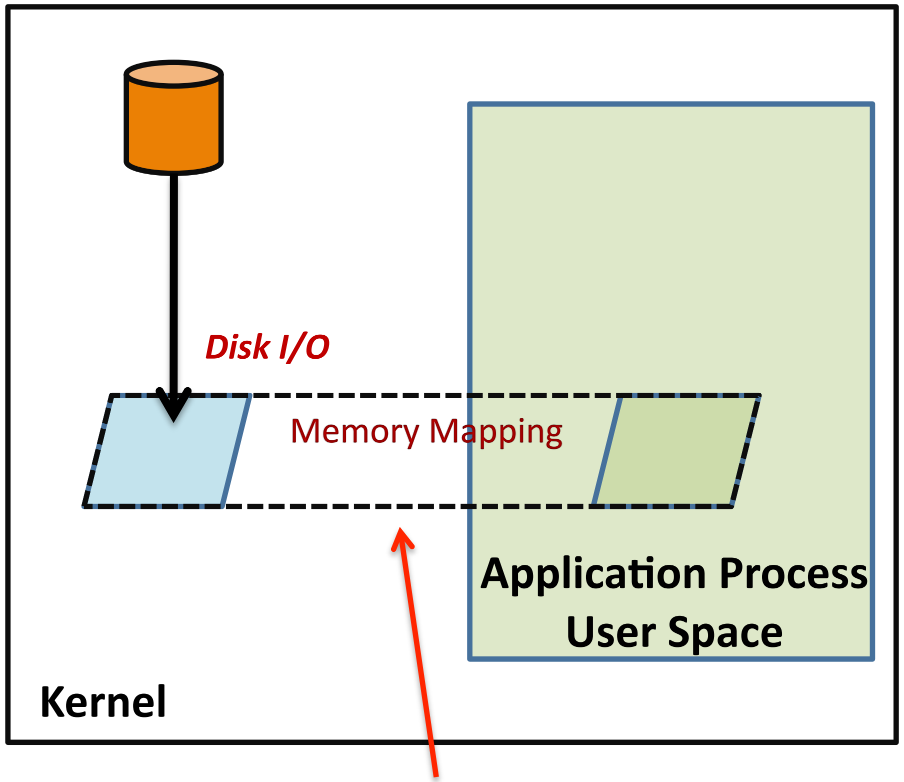
Memory mapping avoids data copy

Without mmap



With usual read() a copy of the data from the kernel space to the user space is required

With mmap



By mapping the physical memory to the virtual memory living in user space avoids the copy. Physical I/O is transparently triggered by page faults, i.e. when pages not yet in memory are accessed

Memory mapping interface

- **mmap()**
  - establishes a memory mapping
- **munmap()**
  - unmaps an established memory mapping
- **msync()**
  - forces modified pages to be written to disk. Depending on the flags passed can be blocking or unblocking
- **mlock()** and **munlock()**
  - locks and unlocks pages in memory. If locked, the pages cannot be swapped out by the kernel when the machine runs out of free memory
- **madvise()**
  - tell kernel how memory mapped pages should be handled, e.g. if you plan to access the memory page-by-page in a sequential or random order

Memory mapped I/O example

```
struct stat sb;
off_t len;
char *p;
int fd;
fd = open (“myfile.dat”, O_RDONLY);
if (fd == -1) {
   perror ("open");
   return 1;
}
if (fstat (fd, &sb) == -1) {
   perror ("fstat");
   return 1;
}
// map the file and attach to the pointer p
// PROT_READ: the map is read only. MAP_SHARE: the map can be shared with other processes
p = mmap (0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
if (p == MAP_FAILED) {
   perror ("mmap");
   return 1;
}
if (close (fd) == -1) {
   perror ("close");
   return 1;
}
// dump the file to stdout byte by byte
for (len = 0; len < sb.st_size; len++) putchar(p[len]);
// unmap the file
if (munmap (p, sb.st_size) == -1) {
   perror ("munmap");
   return 1;
}
```

Memory mapped I/O example with two files

```
int fd1, fd2;
char *buff1, *buff2;
struct stat stat;

// Input file is read-only. The mapping will be private
// open() flags and memory mapping must be coherent
if ((fd1 = open( "file1", O_RDONLY )) < 0) { perror(“open”); return 1; }
if ((fstat(fd1, &stat)) { perror(“fstat”); return 1; }

// map the input file. The file cannot be written and the mapping is not shared
buff1 = (char *)mmap(0, stat.st_size, PROT_READ, MAP_PRIVATE, fd1, 0);
if (buff1 == MAP_FAILED) { perror(“mmap”); return 1; }

// Open the output file
if ((fd2 = open( "file2", O_RDWR )) < 0) { perror(“open”); return 1; }
if ((fstat(fd2, &stat)) { perror(“fstat”); return 1; }

// map the output file
buff2 = (char *)mmap(0,stat.st_size,PROT_READ|PROT_WRITE,MAP_SHARED, fd2,0);
if (buff2 == MAP_FAILED) { perror(“mmap”); return 1; }

// now copy 1024 bytes from input file to output file. It is a simple memory copy
memcpy(buff2, buff1, 1024);

// call msync() to start sync to disk immediately
// specify in this example that this has to be non-blocking. For blocking call use MS_SYNC
if (msync (buff2, stat.st_size, MS_ASYNC) == -1) { perror ("msync"); return 1; }
```

AIO in linux

- The introduction of AIO support into the linux kernel is relatively recent
- As we have already said, the basic idea behind AIO is to allow a process to initiate a number of I/O operations **without having to block or wait** for any to complete
- At some later time or upon a notification from the kernel that the I/O has been completed, the application can retrieve the results of the I/O
- In the meantime, waiting for I/O, the application can actively perform computation

The AIO Interface

- The AIO interface API is quite simple, but it provides all the necessary functions to transfer data with two different notification models
  - **aio_read()**
    - Request an asynchronous read operation
  - **aio_write()**
    - Request an asynchronous write operation
  - **aio_cancel()**
    - Cancel an asynchronous I/O request, i.e. a previous read or write
  - **aio_error()**
    - Check the status of an asynchronous request
  - **aio_return()**
    - Get final status of completed request
  - **aio_suspend()**
    - Suspend the calling process until one or more asynchronous requests have completed (or failed)
  - **lio_listio()**
    - Initiate list of I/O operation

aiocb structure

- Each API function uses the aiocb structure for initiating or checking the status of an I/O request
  – This structure has many elements, here we only show those that one needs to use

```
struct aiocb {
  int aio_fildes;                 // File descriptor
  int aio_lio_opcode;             // valid for lio_listio operation
  volatile void *aio_buf;         // Data buffer
  size_t aio_nbytes;              // Length of buffer in bytes
  struct sigevent aio_sigevent; // Structure used for notification
  off_t aio_offset;               // Offset of the first byte to read

  // Other fields
  ...
};
```

- The sigevent structure tells AIO what to do when the I/O completes
  – We will see how it works in the AIO demonstration later

AIO read example

```
#include <aio.h>

int fd, rc, ret;
struct aiocb my_aiocb;

fd = open( "my_file", O_RDONLY ); // Open the file as usual
if (fd < 0) perror(“open”);

memset((char *)&my_aiocb, 0, sizeof(my_aiocb)); // Initialize to zero: recommended

// Allocate a data buffer for the aiocb request
my_aiocb.aio_buf = malloc(BUFSIZE);
if(!my_aiocb.aio_buf) perror(“malloc”);

// Initialize aiocb structure
my_aiocb.aio_fildes = fd;
my_aiocb.aio_nbytes = BUFSIZE;
my_aiocb.aio_nbytes = 0; // set the offset to the first byte of the file
                         // this is pleonastic here, but in any call you must specify
                         // the correct offset for every call
// submit the read
rc = aio_read( &my_aiocb );
if(rc < 0) perror(“aio_read”);

while( aio_error( &my_aiocb ) == EINPROGRESS ); // Loop checking if I/O is over

if ((ret = aio_return( &my_aiocb )) >= 0) { // data is ready, call aio_return and get it
  // got ret bytes from the read, can consume what is in the buffer
} else {
  perror(“aio_return”);
}
```

Nota Bene: this example is for illustrative purposes only, it does not really make sense as it is implementing a blocking behaviour!

Other common AIO functions

- aio_write() is used to submit a write, in an analogue way to aio_read()
- aio_suspend() can be used to block the calling process until an asynchronous I/O request has completed, a signal is raised, or a timeout occurs
  - A list of aiocb references is provided: if any of them complete, the call returns

```
struct aiocb *my_list[MAX_LIST];
memset( (char *)my_list, 0, sizeof(mylist) ); // Clear the list

// Load one or more references into the list
my_list[0] = &my_aiocb_0;
my_list[1] = &my_aiocb_1;
...

ret = aio_read( &my_aiocb_0 );
ret = aio_read( &my_aiocb_1 );

// block with aio_suspend, NULL elements in my_list are ignored
ret = aio_suspend( my_list, MAX_LIST, NULL );
```

- aio_cancel() is used to cancel one or all requests previously submitted to a given file descriptor

lio_listio()

- AIO provides a way to initiate multiple transfers at the same time
  - lio_listio() is useful because it allows you to start lots of I/Os within one single kernel context switch

```
struct aiocb aiocb1, aiocb2;
struct aiocb *list[MAX_LIST];
...
// Prepare the first aiocb
aiocb1.aio_fildes = fd;
aiocb1.aio_buf = malloc(BUFSIZE);
aiocb1.aio_nbytes = BUFSIZE;
aiocb1.aio_offset = next_offset;
aiocb1.aio_lio_opcode = LIO_READ; // the operation must be indicated in the aiocb
...
memset( (char *)list, 0, sizeof(list) );
list[0] = &aiocb1;
list[1] = &aiocb2;
// issue the lio_listio with LIO_WAIT (means blocking)
ret = lio_listio( LIO_WAIT, list, MAX_LIST, NULL ); // ignores NULL elements
```

- The read operation is noted in the aio_lio_opcode field with LIO_READ (LIO_WRITE for a write operation)
- The LIO_WAIT means the call will block
  - Better way to use it is with LIO_NOWAIT, i.e. non-blocking exploiting asynchronous notification, as we will see in a while

AIO notification

- The AIO notification relies on two different techniques
  - signal and callback notifications
- The use of signals is a common mechanism in unix systems and is also supported by AIO
  - The application can define a special function (a so-called signal handler) that is invoked when a specified signal occurs
  - We can configure AIO via the aiocb structure such that an asynchronous request will raise a signal when the I/O request has completed
- Using callbacks, instead of raising a signal for notification, AIO calls a function in user-space upon I/O completion
  - In this case sigevent structure into the aiocb is used specify the pointer to the function to be called back

Signal-based AIO notification example

```
  int fd;
  struct sigaction sig_act;
  struct aiocb my_aiocb;
  ...
  // Set up the signal handler
  sigemptyset(&sig_act.sa_mask);
  sig_act.sa_flags = SA_SIGINFO;
  sig_act.sa_sigaction = aio_completion_handler; // function to call when signal fires
  // Set up the aio request
  memset( (char *)&my_aiocb, 0, sizeof(struct aiocb) );
  my_aiocb.aio_fildes = fd;
  my_aiocb.aio_buf = malloc(BUFSIZE);
  my_aiocb.aio_nbytes = BUFSIZE;
  my_aiocb.aio_offset = next_offset;
  // Link the aio request to the signal
  my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
  my_aiocb.aio_sigevent.sigev_signo = SIGIO; // in this example use SIGIO signal
  my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
  // Map the signal to the signal handler
  ret = sigaction( SIGIO, &sig_act, NULL );
  ...
  ret = aio_read( &my_aiocb ); // submit the read request and return immediately
  ... // perform some computation while I/O is being carried out

void aio_completion_handler( int signo, siginfo_t *info, void *context ) {
  struct aiocb *req;
  req = (struct aiocb *) info->si_value.sival_ptr;
  if (aio_error( req ) == 0) {  // Did the request complete succesfully?
    // Request completed successfully, get the return status
    ret = aio_return( req );
  }
  return;
}
```

Not done in this example, but to increase performance the handler could continue the I/O by requesting the next asynchronous transfer. In this way, when one transfer has completed, the next is immediately started.

Callback-based AIO notification example

```
int fd;
struct aiocb my_aiocb;
...
// Set up the aio request
memset( (char *)&my_aiocb, 0, sizeof(struct aiocb) );
my_aiocb.aio_fildes = fd;
my_aiocb.aio_buf = malloc(BUFSIZE);
my_aiocb.aio_nbytes = BUFSIZE;
my_aiocb.aio_offset = next_offset;
// Link the aio request with a thread callback
my_aiocb.aio_sigevent.sigev_notify = SIGEV_THREAD;
my_aiocb.aio_sigevent.notify_function = aio_completion_handler;
my_aiocb.aio_sigevent.notify_attributes = NULL;
my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;
...
ret = aio_read( &my_aiocb );

void aio_completion_handler( sigval_t sigval ) {
  struct aiocb *req;
  req = (struct aiocb *)sigval.sival_ptr; // cast from sigval to get the aiocb pointer
  // Check if the request completed successfully
  if (aio_error( req ) == 0) {
     // Request completed successfully, get the return status
     ret = aio_return( req );
  }
  return;
}
```

AIO kernel tuning and conclusions

- The proc file system contains two virtual files that can be tuned for asynchronous I/O performance
  - `/proc/sys/fs/aio-nr` provides the current number of system-wide asynchronous I/O requests
  - `/proc/sys/fs/aio-max-nr` is the maximum number of allowable concurrent requests
    - This value is commonly 65535 which is adequate for most applications
- Asynchronous I/O is a tool which allows to design efficient I/O applications
  - This is of interest when the application I/O and data processing can be overlapped and executed in parallel
  - This I/O model differs from the traditional simple minded blocking patterns, by the asynchronous notification model is conceptually simple, although technically more advanced

Practical demonstration #3

```
# cd ~/io_exercise/demo3
```

- Have a look into README.txt

Practical demonstration #4

```
# cd ~/io_exercise/demo4
```

- Have a look into README.txt

Free demonstration

- If in the past days you have run some realistic job which reads files in input or writes files to output, and you remember how to rerun it, try to launch, monitor and profile the application by your own

Fin

Good luck guys!