Key ingredients to achieve effective I/O

Sébastien Ponce
sebastien.ponce@cern.ch

CERN

Thematic CERN School of Computing 2019

In the previous episodes...

- We've found out how to store data efficiently
- And how to distribute it
- We've learnt how to detect data corruptions
- And made sure our data was safe and consistent

In the previous episodes...

- We've found out how to store data efficiently
- And how to distribute it
- We've learnt how to detect data corruptions
- And made sure our data was safe and consistent

Today

How do we use them ? Efficiently

Outline

Asynchronous I/O

Sources of I/O latency

Physical constraints

- preparing an SSD ($\sim$ 100 µs)
- moving disk's arm ($\sim$ 10 ms)
- mounting and rotating tapes ($\sim$ 1 min)

Sources of I/O latency

Physical constraints

- preparing an SSD ($\sim$ 100 µs)
- moving disk's arm ($\sim$ 10 ms)
- mounting and rotating tapes ($\sim$ 1 min)

Network infrastructure

- number of switches routers on the way
- 200 µs delay per switch/router

CERN School of Computing

Sources of I/O latency

Physical constraints

- preparing an SSD ($\sim$ 100 µs)
- moving disk's arm ($\sim$ 10 ms)
- mounting and rotating tapes ($\sim$ 1 min)

Network infrastructure

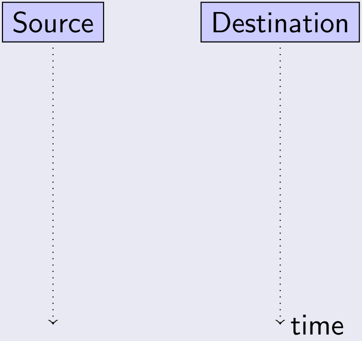
- number of switches routers on the way
- 200 µs delay per switch/router

"slow" speed of light

- "speed" of light is slower in a fiber (refractive index 1.47)
- that gives around 200 m µs^{-1} or 20 cm ns^{-1}
- Budapest's ping time from CERN is $\sim$21 ms for $\sim$2500 km

Impact of I/O latency

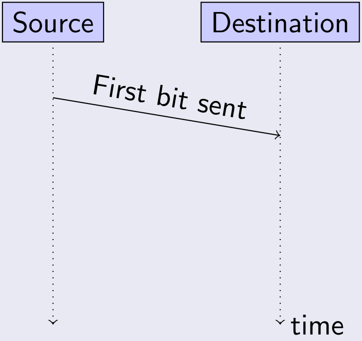
Typical I/O pattern



Impact of I/O latency

Typical I/O pattern

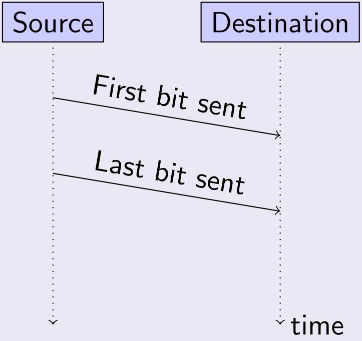
- send a packet to destination



Impact of I/O latency

Typical I/O pattern

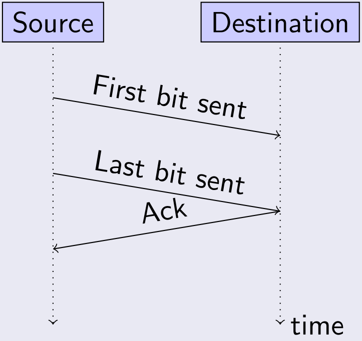
- send a packet to destination



Impact of I/O latency

Typical I/O pattern

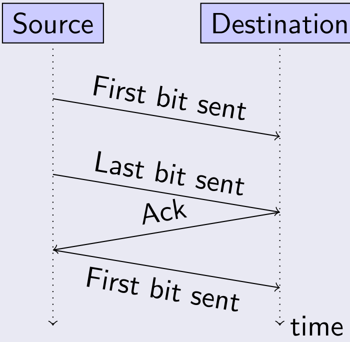
- send a packet to destination
- wait for ack



Impact of I/O latency

Typical I/O pattern

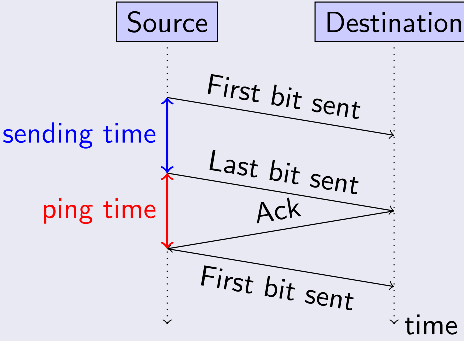
- send a packet to destination
- wait for ack
- go to next block



Impact of I/O latency

Typical I/O pattern

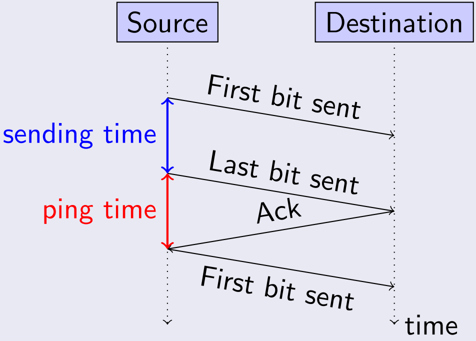
- send a packet to destination
- wait for ack
- go to next block



Impact of I/O latency

Typical I/O pattern

- send a packet to destination
- wait for ack
- go to next block



$$efficiency = \frac{sending\ time}{sending\ time + ping\ time}$$

Some mathematics

Definitions

$$efficiency = \frac{sending\ \ time}{sending\ \ time + ping\ \ time} \tag{1}$$

$$sending\ \ time = \frac{data\ size}{speed} \tag{2}$$

$$ping\ \ size = speed * ping\ \ time \tag{3}$$

Some mathematics

Definitions

$$efficiency = \frac{sending\ time}{sending\ time + ping\ time} \quad (1)$$

$$sending\ time = \frac{data\ size}{speed} \quad (2)$$

$$ping\ size = speed * ping\ time \quad (3)$$

Gives

$$efficiency = \frac{1}{1 + \frac{ping\ size}{data\ size}} \quad (4)$$

$$data\ size = \frac{efficiency}{1 - efficiency} * ping\ size \quad (5)$$

Some (bad) numbers

Consequences for 10KB blocks

Usage	Speed	Latency	Ping Size	Efficiency
CC	$1\,GB\,s^{-1}$	$100\,\mu s$	10 kB	50%
CC	$10\,GB\,s^{-1}$	$100\,\mu s$	100 kB	9%
WAN	$1\,GB\,s^{-1}$	10 ms	1 MB	1%
WAN	$10\,GB\,s^{-1}$	10 ms	10 MB	1‰
UK-JP	$10\,GB\,s^{-1}$	250 ms	250 MB	0.04‰

More bad numbers

Data size for decent efficiency

Usage	Speed	Latency	50% efficiency	91% efficiency
CC	$1\,GB\,s^{-1}$	100 μs	10 kB	100 kB
CC	$10\,GB\,s^{-1}$	100 μs	100 kB	1 MB
WAN	$1\,GB\,s^{-1}$	10 ms	1 MB	10 MB
WAN	$10\,GB\,s^{-1}$	10 ms	10 MB	100 MB
UK-JP	$10\,GB\,s^{-1}$	250 ms	250 MB	2.5 GB

More bad numbers

Data size for decent efficiency

Usage	Speed	Latency	50% efficiency	91% efficiency
CC	$1\,\mathrm{GB\,s^{-1}}$	100 μs	10 kB	100 kB
CC	$10\,\mathrm{GB\,s^{-1}}$	100 μs	100 kB	1 MB
WAN	$1\,\mathrm{GB\,s^{-1}}$	10 ms	1 MB	10 MB
WAN	$10\,\mathrm{GB\,s^{-1}}$	10 ms	10 MB	100 MB
UK-JP	$10\,\mathrm{GB\,s^{-1}}$	250 ms	250 MB	2.5 GB

Remember maximum TCP packet size is 64 KiB

The solution : asynchronous I/O

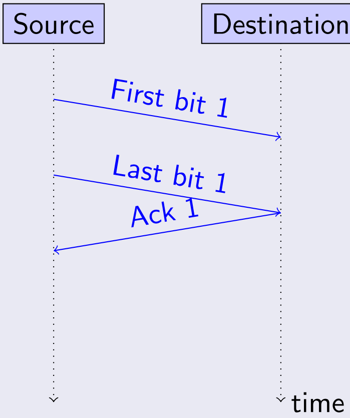
Do not wait the acknowledgment !

- call an API to express what should be transfered
- this immediately returns without doing much
- get called back when transfer has been done/has failed

The solution : asynchronous I/O

Do not wait the acknowledgment !

- call an API to express what should be transfered
- this immediately returns without doing much
- get called back when transfer has been done/has failed

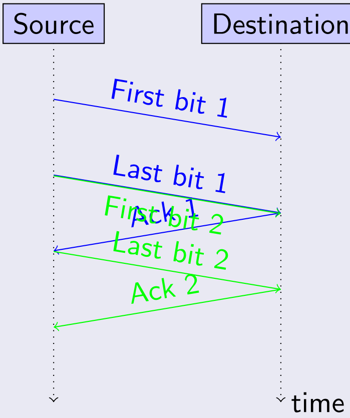


CERN School of Computing

The solution : asynchronous I/O

Do not wait the acknowledgment !

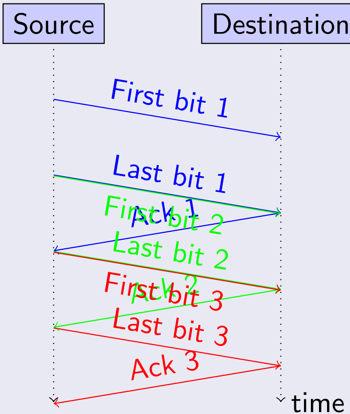
- call an API to express what should be transfered
- this immediately returns without doing much
- get called back when transfer has been done/has failed



The solution : asynchronous I/O

Do not wait the acknowledgment !

- call an API to express what should be transfered
- this immediately returns without doing much
- get called back when transfer has been done/has failed

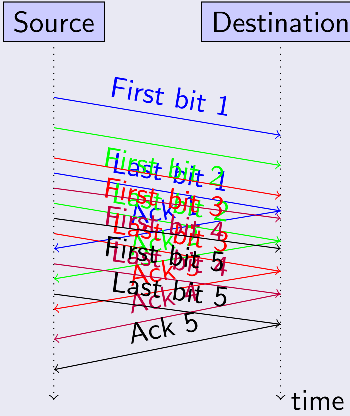


CERN School of Computing

The solution : asynchronous I/O

Do not wait the acknowledgment !

- call an API to express what should be transfered
- this immediately returns without doing much
- get called back when transfer has been done/has failed



The complexity of asynchronous I/O

Some consequences

- involves multi-threaded / multi-process approach
  - with a background task for the actual transfer.
- involves out of order transfers
  - no garantee on the order in which transfers will arrive to destination
  - especially in case of failures
- involves queue management on the caller side
  - to not send too much in parallel
  - to adapt to network efficiency

The complexity of asynchronous I/O

Some consequences

- involves multi-threaded / multi-process approach
  - with a background task for the actual transfer.
- involves out of order transfers
  - no garantee on the order in which transfers will arrive to destination
  - especially in case of failures
- involves queue management on the caller side
  - to not send too much in parallel
  - to adapt to network efficiency

Libraries

Do not try to do all that by hand
Use a library that does (most of) it for you

CERN School of Computing

POSIX aynchronous I/O

Old interface based on signals

```
struct aiocb {
  int                aio_fildes;     /* fd */
  off_t              aio_offset;     /* offset */
  volatile void     *aio_buf;        /* buffer */
  size_t             aio_nbytes;     /* length */
  int                aio_reqprio;    /* priority */
  struct sigevent    aio_sigevent;   /* cb method */
  int                aio_lio_opcode; /* operation */
};
int aio_read (struct aiocb *aiocbp);
int aio_write (struct aiocb *aiocbp)
```

The application can elect to be notified of completion of the I/O operation in a variety of ways: by delivery of a signal, by instantiation of a thread, or no notification at all.

POSIX aynchronous I/O

Internal tweaks

- The proc file system contains two virtual files that can be tuned for asynchronous I/O performance:
  - The /proc/sys/fs/aio-nr file provides the current number of system-wide asynchronous I/O requests.
  - The /proc/sys/fs/aio-max-nr file is the maximum number of allowable concurrent requests. The maximum is commonly 64KB, which is adequate for most applications.

CERN School of Computing

Ceph aynchronous I/O

More modern interface with callbacks

```
typedef void (*rados_callback_t)
  (rados_completion_t cb, void *arg);
int rados_aio_create_completion
  (void *cb_arg, rados_callback_t cb_complete,
   rados_callback_t cb_safe,
   rados_completion_t *pc);
int rados_aio_write
  (rados_ioctx_t io, const char * oid,
   rados_completion_t completion,
   const char * buf, size_t len, uint64_t off);
int rados_aio_wait_for_safe(rados_completion_t c);
```

Handling of threading is done for you

CERN School of Computing

Ceph aynchronous I/O

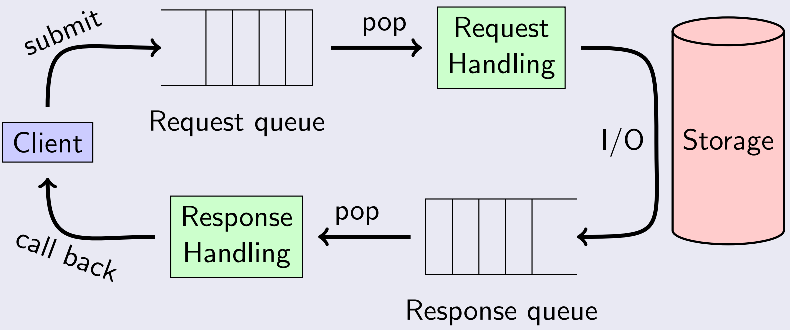
example code

```
void commit_cb(rados_completion_t comp, void *arg) {
  struct timeval *t = (struct timeval *) arg;
  gettimeofday(&t, NULL);
}
int main() {
  ... // connect to ceph
  struct timeval commit_end;
  rados_completion_t comp;
  rados_aio_create_completion
    ((void*) &commit_end, NULL, commit_cb, &comp);
  rados_aio_append(ioctx, obj_name, comp, data, len);
  rados_aio_flush(io); // wait for call back
  // print commit_end
}
```

Modern aynchronous I/O internals

Architecture

- based on message queuing
- both for requests and reponses



Advantages of message queues

Architecture

- disentangles client API calls from actual processing
- allows optimizations on both sides
  - asynchronous I/O on client side
  - reordering of requests on server side
- allows easy scalability, parallelizing the handlers
  - using thread safe or even distributed queues

I/O optimizations

Optimizing network I/O

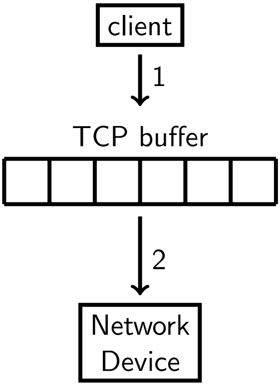
Golden rules

- async I/O is a must
- setsockopt is often needed
  - as default parameters may not be optimal for your case
  - e.g. TCP_NODELAY or SND_BUF

The TCP buffer

Actual steps writing to a TCP socket

1. write to the TCP buffer
2. create a paquet and actually send it to the network device



The Nagle algorithm of TCP

Reason of being

- avoid sending too many small paquets when write to TCP buffer are very small
  - as there is 40 bytes overhead (20 for TCP, 20 for IPv4)
- initially for key pressing in telnet sessions over ARPANET (1984 !)

Algorithm

- if enough data ($\geq$ maximum segment size), send a paquet
- else if there is unconfirmed data still in the pipe
  - enqueue data in the buffer until an acknowledge is received
- else send data immediately

Potential bad consequences

TCP delayed acknowledgment

- allows to combine several acknowledgments into a single response to reduce protocol overhead
- ACK may wait up to 500 ms

Collision of features

- suppose you use a socket in "write-read-read" schema
  - e.g. send control paquet "give me more" and read until nothing left
- a delay of 500 ms will be introduced

CERN School of Computing

Bypass Nagle via TCP_NODELAY

Code example

```
int flag = 1;
int result =
  setsockopt(sock,            /* socket */
             IPPROTO_TCP,     /* TCP level */
             TCP_NODELAY,     /* option */
             (char *) &flag,  /* value */
             sizeof(int));    /* length */
if (result < 0) { ... error handling ... }
```

A note on the size of the TCP buffer

Standard case

- default size is used (16 kB)
- allows to create paquets up to 16 kB
- already not able to use the maximum of 64 kB

A note on the size of the TCP buffer

Standard case

- default size is used (16 kB)
- allows to create paquets up to 16 kB
- already not able to use the maximum of 64 kB

Bad case

- via setsockopt and SND_BUF option, the TCP buffer is shortened
  - a bug strikes, size is set to 0, which will be corrected into 1448
- many small paquets will be sent
- latency will kill efficiency
  - e.g. to $\leq 5\,\mathrm{MB\,s^{-1}}$

A note on the size of the TCP buffer

Standard case

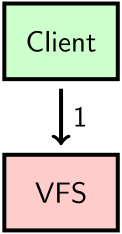
- default size is used (16 kB)
- allows to create paquets up to 16 kB
- already not able to use the maximum of 64 kB

Bad case

- via setsockopt and SND_BUF option, the TCP buffer is shortened
  - a bug strikes, size is set to 0, which will be corrected into 1448
- many small paquets will be sent
- latency will kill efficiency
  - e.g. to $\leq 5\,\mathrm{MB\,s^{-1}}$
- and CMS may call you for complaining....

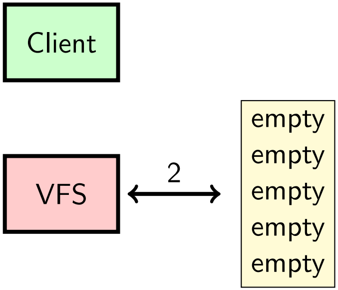
Anatomy of local I/O in linux

1. client requests bytes from VFS
   Virtual File System



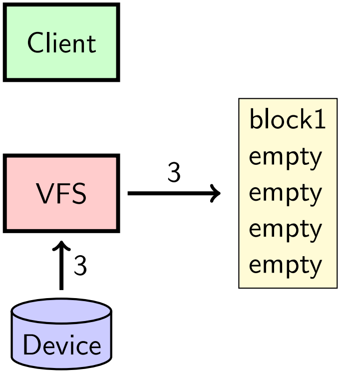
Anatomy of local I/O in linux

1. client requests bytes from VFS
   Virtual File System
2. kernel checks whether they are
   available in cache



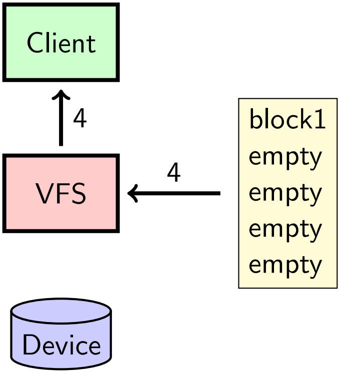
Anatomy of local I/O in linux

1. client requests bytes from VFS
   Virtual File System
2. kernel checks whether they are available in cache
3. if not, it reads a 4K block from device and updates cache



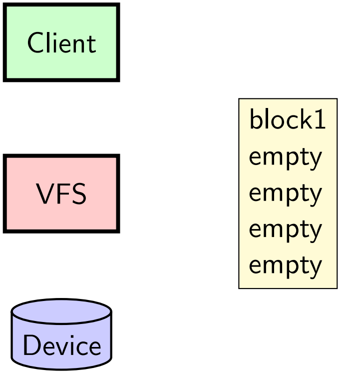
Anatomy of local I/O in linux

1. client requests bytes from VFS Virtual File System
2. kernel checks whether they are available in cache
3. if not, it reads a 4K block from device and updates cache
4. bytes are read from cache and given to application



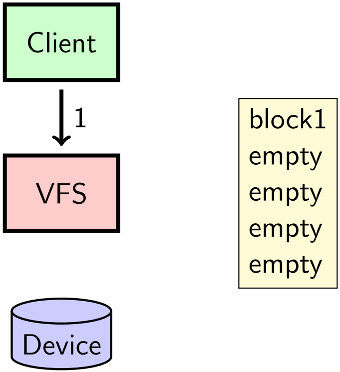
Anatomy of local I/O in linux

1. client requests bytes from VFS
   Virtual File System
2. kernel checks whether they are available in cache
3. if not, it reads a 4K block from device and updates cache
4. bytes are read from cache and given to application
5. subsequent reads skip step 3



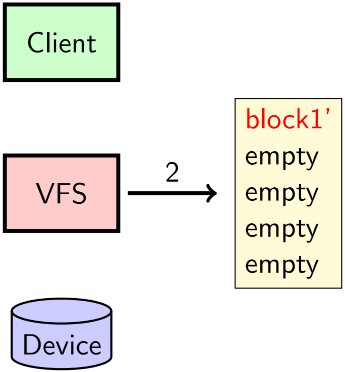
The write case

1. client sends bytes to VFS



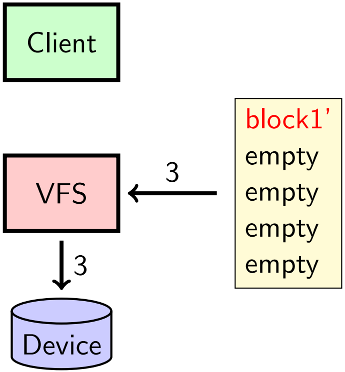
The write case

1. client sends bytes to VFS
2. kernel overwrite block(s) in cache and marks the cache dirty

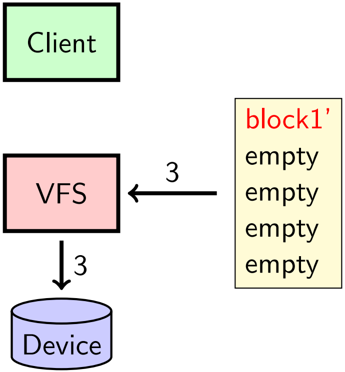


The write case

1. client sends bytes to VFS
2. kernel overwrite block(s) in cache and marks the cache dirty
3. "later", the cache is synced to disk



The write case



1. client sends bytes to VFS
2. kernel overwrite block(s) in cache and marks the cache dirty
3. "later", the cache is synced to disk

Possible data loss

- between step 2 and 3, data are only in cache
- you need to use fsync to sync to the device

More data loss...

fsync

- only forces the cache to be synced to the device
- that is the content of the cache to be sent to the device

But...

- there may be many other caches involved
  - in the controller
  - in the hardware, e.g. inside the disk itself
- so after syncing, data may still be at risk

More data loss...

fsync

- only forces the cache to be synced to the device
- that is the content of the cache to be sent to the device

But...

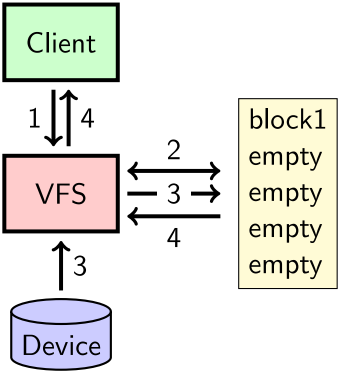
- there may be many other caches involved
  - in the controller
  - in the hardware, e.g. inside the disk itself
- so after syncing, data may still be at risk

In practice, massive reboots ⇔ loss of data !

Block size matters

Back to our read case :

1. client requests bytes from VFS
   Virtual File System
2. kernel checks whether they are available in cache
3. if not, it reads **a 4K block** from device and updates cache
4. bytes are read from cache and given to application

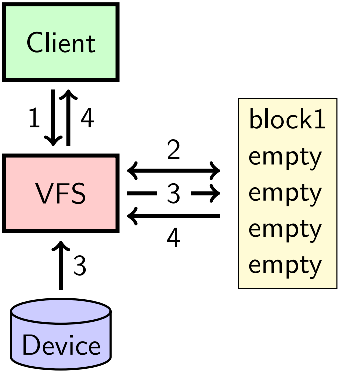


Block size matters

Back to our read case :

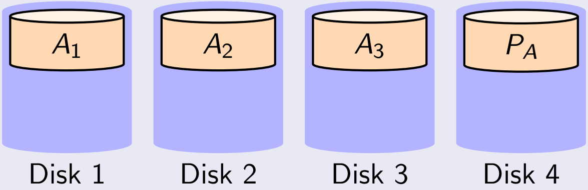
1. client requests bytes from VFS
   Virtual File System
2. kernel checks whether they are available in cache
3. if not, it reads **a 4K block** from device and updates cache
4. bytes are read from cache and given to application

- scattered tiny reads will transform into 4K blocks anyway
- so scanning events, reading only the 4 bytes of their type may not be very efficient !



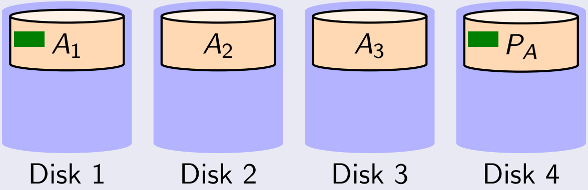
Block size : the real killer

Remember RAID 5 - aka Parity



Block size : the real killer

Remember RAID 5 - aka Parity



4 bytes write into into A_1

- the parity will have to be recomputed
  - either A_1 and P_A or all A_i will be read
  - and A_1 and P_A will written back
- that's a minimum of 8K reads + 8K writes for 4 bytes !
- can be much worse for RAID6 or erasure coding

Combined RAID levels

RAID 60

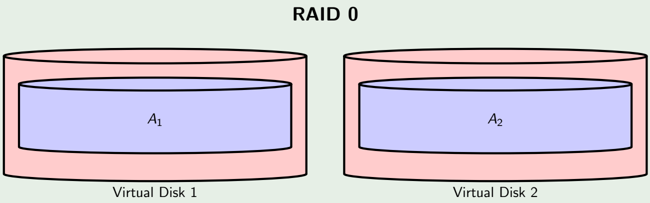
- RAID levels can be put ontop each other
- 60 means a RAID 0 on top of a RAID 6
- Let's see how a file A is stored on such a configuration

Combined RAID levels

RAID 60

- RAID levels can be put ontop each other
- 60 means a RAID 0 on top of a RAID 6
- Let's see how a file A is stored on such a configuration

Layout of file A

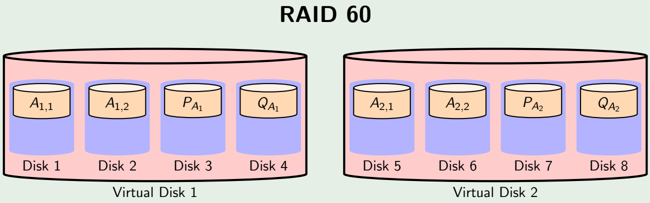


Combined RAID levels

RAID 60

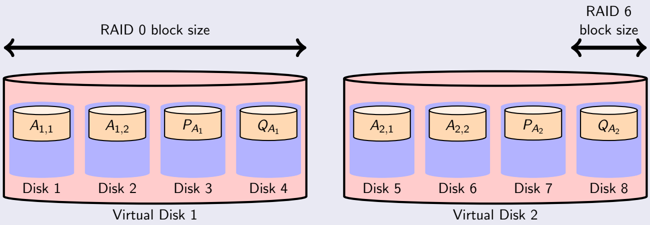
- RAID levels can be put ontop each other
- 60 means a RAID 0 on top of a RAID 6
- Let's see how a file A is stored on such a configuration

Layout of file A



Block size and combined RAID levels

2 levels - 2 chunk sizes, ideally like this



Incoherent chunk sizes

Consequence of block size incoherence

- suppose you have the same chunk size for both levels
- writing a chunk to the RAID 0 will only fill one of the chunks in the RAID 6 stripe
- it will trigger the reading of the rest to recompute the parities
- leading to 3 reads and 3 writes
- to fill a stripe of the RAID 6, this will be repeated twice

Incoherent chunk sizes

Consequence of block size incoherence

- suppose you have the same chunk size for both levels
- writing a chunk to the RAID 0 will only fill one of the chunks in the RAID 6 stripe
- it will trigger the reading of the rest to recompute the parities
- leading to 3 reads and 3 writes
- to fill a stripe of the RAID 6, this will be repeated twice

Bottom line

- 6 reads and 6 writes instead of 4 writes !
- a slow down of a factor 3 minimum

Incoherent chunk sizes

Consequence of block size incoherence

- suppose you have the same chunk size for both levels
- writing a chunk to the RAID 0 will only fill one of the chunks in the RAID 6 stripe
- it will trigger the reading of the rest to recompute the parities
- leading to 3 reads and 3 writes
- to fill a stripe of the RAID 6, this will be repeated twice

Bottom line

- 6 reads and 6 writes instead of 4 writes !
- a slow down of a factor 3 minimum
- in case of 8+2 disks in the RAID6,
  the slow down is of a factor 5 !

O_DIRECT : bypass the kernel cache

When / Why ?

- when reading large amounts of data sequentially
  - so that you avoid wiping the cache
- if you're handling caching by hand
  - typical use case is databases

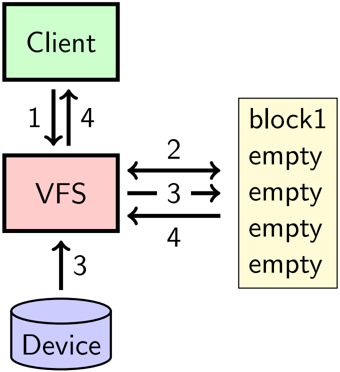
How

```
int fd = open("myfile", O_RDONLY|O_DIRECT, ...);
```

Data duplication

Back again to our read case :

1. client requests bytes from VFS
2. kernel checks the cache
3. kernel transfers a 4K block from device to cache
4. bytes are read from cache and **given to application**

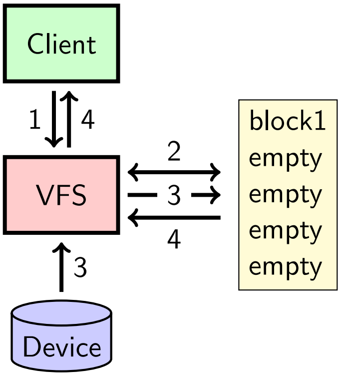


Data duplication

Back again to our read case :

1. client requests bytes from VFS
2. kernel checks the cache
3. kernel transfers a 4K block from device to cache
4. bytes are read from cache and **given to application**

- "given to application" ⇔ "copied to user buffer"
    - it takes CPU time
    - it hurts the CPU cache
    - only to duplicate the data into memory !



Memory-mapped files

Idea

- allows direct use of kernel cache, mapped to user buffer
- for up to 30% gain compared to regular file read

CERN School of Computing

Memory-mapped files

Idea

- allows direct use of kernel cache, mapped to user buffer
- for up to 30% gain compared to regular file read

Usage

```
int *map;  /* mmapped array of int's */
int fd = open(FILEPATH, O_RDONLY);
if (fd == -1) { ... }
map = mmap(0, SIZE, PROT_READ, MAP_SHARED, fd, 0);
if (map == MAP_FAILED) { ... }
/* access file through map array */
if (munmap(map, FILESIZE) == -1) { ... }
close(fd);
```

slightly more complicated for writing but still easy

Read-ahead mechanism

Principle

- when you read a block of a file, there is high chance you will read the next few ones
- cache misses in the kernel cache are expensive (disk seeks)
- so why no read the next few blocks from the start ?

practical usage

- just do nothing, this is on by default
- give advices to kernel for special cases :
  - use posix_fadvise, posix_madvise or madvice (non POSIX)
  - advices include POSIX_FADV_SEQUENTIAL, POSIX_FADV_RANDOM, POSIX_FADV_NOREUSE

Influence of data structures on I/O

Algorithm efficiency

Standard efficiency metric

- speed, that is time spent per data
- mostly CPU time
- mapping to number of operations, e.g. $O(n^2)$

Algorithm efficiency

Standard efficiency metric

- speed, that is time spent per data
- mostly CPU time
- mapping to number of operations, e.g. $O(n^2)$

Well... what about I/O ?

- e.g. scan 1GB of logs and count nb Errors
- is CPU dominant ?

Algorithm efficiency

Standard efficiency metric

- speed, that is time spent per data
- mostly CPU time
- mapping to number of operations, e.g. $O(n^2)$

Well... what about I/O ?

- e.g. scan 1GB of logs and count nb Errors
- is CPU dominant ?
- depends whether data is in RAM or on disk...

Data access latencies

Remember the numbers of Danilo

Device	Latency (time)	Latency (cycles)
Reg	0.4 ns	1
L1	1.5 ns	4
L2	4 ns	10
L3	14 ns	35
Mem	80 ns	200
SSD	12 µs	30 000
HDD	4 ms	10 000 000

Data access latencies

Remember the numbers of Danilo

Device	Latency (time)	Latency (cycles)
Reg	0.4 ns	1
L1	1.5 ns	4
L2	4 ns	10
L3	14 ns	35
Mem	80 ns	200
SSD	12 µs	30 000
HDD	4 ms	10 000 000

Bottom line :

- never, ever go to disk if not absolutely needed !
- care about cache hits in general

Algorithm I/O efficiency

- we use mem and disk here
- but it applies to any two levels

Notations

N	number of items in the problem
M	number of items fitting in memory
B	number of items fitting in a disk block

Algorithm I/O complexity

Fundamental Bounds of I/O complexity

- scanning : $O(\frac{N}{B})$
- searching : $O(log_B N)$
- sorting : $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$

CERN School of Computing

Algorithm I/O complexity

Fundamental Bounds of I/O complexity

- scanning : $O(\frac{N}{B})$
- searching : $O(log_B N)$
- sorting : $O(\frac{N}{B} log_{\frac{M}{B}} \frac{N}{B})$

Importance of B

- N = 256 10^6, B = 8000, 1ms disk access time
- N I/Os take 256 10^3 sec = 71 h
- $\frac{N}{B}$ I/Os take 32 s

Real life example : linked list

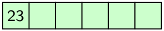
standard implementation

- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks

Real life example : linked list

standard implementation

- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks

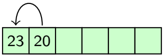


CERN School of Computing

Real life example : linked list

standard implementation

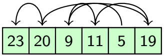
- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks



Real life example : linked list

standard implementation

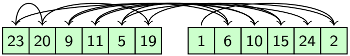
- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks



Real life example : linked list

standard implementation

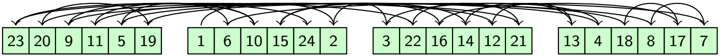
- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks



Real life example : linked list

standard implementation

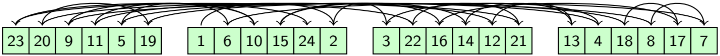
- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks



Real life example : linked list

standard implementation

- nodes are stored in an array and contain pointer to next
- new nodes are inserted at the end of the array
- but in the middle of the linked list
- so most of links are across disk blocks



Complexity

insertion $O(1)$

scan $O(N)$

Improved linked list

Ordered linked list

- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block

Improved linked list

Ordered linked list

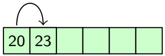
- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in "list order", blocks contain consecutive elements
- most of the links point to same block

23					

Improved linked list

Ordered linked list

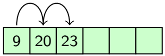
- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in "list order", blocks contain consecutive elements
- most of the links point to same block



Improved linked list

Ordered linked list

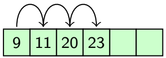
- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in "list order", blocks contain consecutive elements
- most of the links point to same block



Improved linked list

Ordered linked list

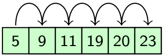
- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in "list order", blocks contain consecutive elements
- most of the links point to same block



Improved linked list

Ordered linked list

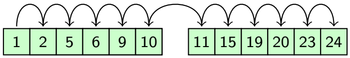
- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in "list order", blocks contain consecutive elements
- most of the links point to same block



Improved linked list

Ordered linked list

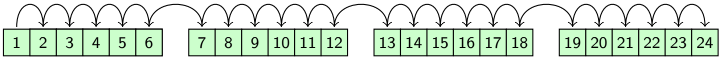
- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in “list order”, blocks contain consecutive elements
- most of the links point to same block



Improved linked list

Ordered linked list

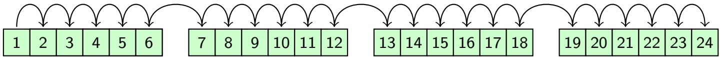
- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in "list order", blocks contain consecutive elements
- most of the links point to same block



Improved linked list

Ordered linked list

- nodes are stored in an array and contain pointer to next
- new nodes are inserted in the middle of the array
- nodes are kept in "list order", blocks contain consecutive elements
- most of the links point to same block



Complexity

insertion $O(\frac{N}{B})$

scan $O(\frac{N}{B})$

I/O Efficient linked list

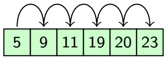
Ordered sparse linked list

- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse

I/O Efficient linked list

Ordered sparse linked list

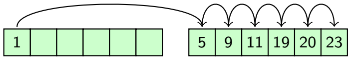
- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse



I/O Efficient linked list

Ordered sparse linked list

- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse

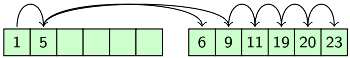


CERN School of Computing

I/O Efficient linked list

Ordered sparse linked list

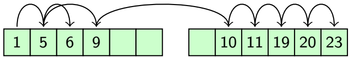
- keep same ideas
  - nodes put in "list order"
  - links point mainly to same block
- but allow blocks to be sparse



I/O Efficient linked list

Ordered sparse linked list

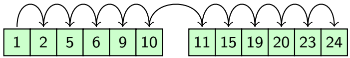
- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse



I/O Efficient linked list

Ordered sparse linked list

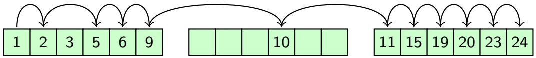
- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse



I/O Efficient linked list

Ordered sparse linked list

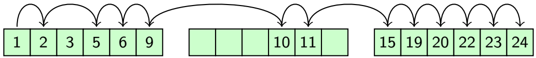
- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse



I/O Efficient linked list

Ordered sparse linked list

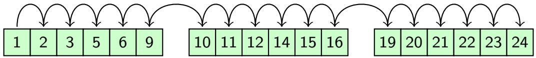
- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse



I/O Efficient linked list

Ordered sparse linked list

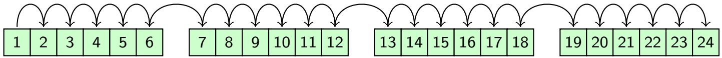
- keep same ideas
  - nodes put in "list order"
  - links point mainly to same block
- but allow blocks to be sparse



I/O Efficient linked list

Ordered sparse linked list

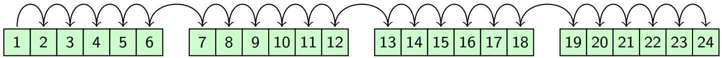
- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse



I/O Efficient linked list

Ordered sparse linked list

- keep same ideas
  - nodes put in “list order”
  - links point mainly to same block
- but allow blocks to be sparse



Complexity

insertion $O(1)$

scan $O(\frac{N}{B})$

I/O complexity in general

Applies to more complex structures / algorithms

- queues, stack, trees
- searching, sorting

Is always a trade off with CPU complexity

- ordered linked list means $O(n)$ CPU complexity at insertion
- a tree of nodes stored in an array may be a better alternative

Libraries exist

- In particular STXXL, the STL for XXL Data Sets
- `http://stxxl.sourceforge.net`

Caching

Caching principles

Why to use (custom) caches

- to optimize access to a costful resource (network, storage, ...)
- to take benefit of particular access patterns

Caching principles

Why to use (custom) caches

- to optimize access to a costful resource (network, storage, ...)
- to take benefit of particular access patterns

When to use custom caches

- when you have a particular, well defined access pattern
  - e.g. web browsing, with mostly static pages
  - e.g. tape/archive handling
  - e.g. proxy architecture for software distribution
- for specific data structures that would benefit from a cache
  - e.g. trees in the root framework

Caching usage

Where to use caching

- server side to benefit of cross client hits
- client side to avoid network traffic
  - think of your browser cache
- in a proxy to lower load on the server
- locally, in front of "expensive" storage
  - tape $\rightarrow$ disk cache
  - disk $\rightarrow$ ssd/memory cache

Anatomy of a cache

Architecture

- it has a limited size
- so must use some garbage collection algorithm

Policies

Replacement policy which item to discard when cache is full and new item is coming

Writing policy how to handle writes of new items to the system

Write-miss policy how to handle "write misses"

Replacement policies

FIFO - First In First Out

- most naive and easiest to implement (simple list)
- not very effective in practice

LRU - Least Recently Used

- still easy to implement, just needs some “age bits”
- one of the most common

Others

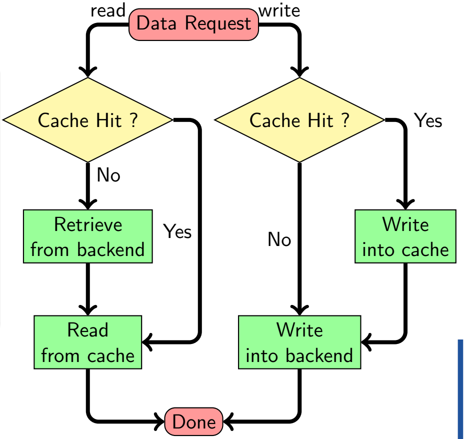
LFU Least Frequently Used
refinement on LRU where number of accesses is taken into account

LIRS Low Inter-reference Recency Set (LIRS)
using reuse distance as a metric for dynamically ranking accessed pages

Writing policies (1)

Write-through

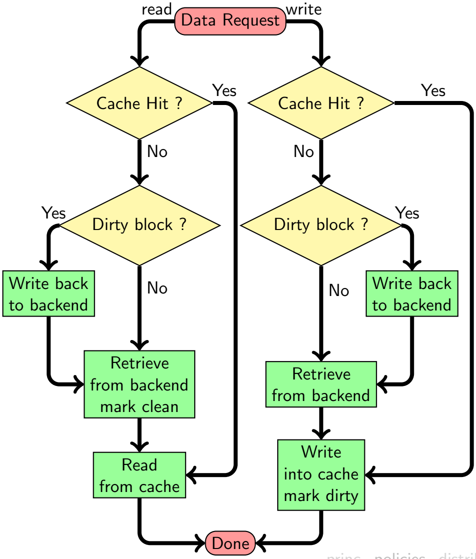
- write is done synchronously both to the cache and to the backend store
- no caching gain on writes
- backend store is always up to date



Writing policies (2)

Write-back

- writing is done only to the cache initially
- the write to the backing store is postponed until the cache blocks containing the data are about to be modified/replaced by new content
- writes are faster
- but backend store is not up to date
- data may be at risk



Write-miss policies

Write allocate

- datum at the missed-write location is loaded to cache, followed by a write-hit operation
- write misses are similar to read misses

No write allocate

- datum at the missed-write location is not loaded to cache
- it is written to the backend
- no cache for writes

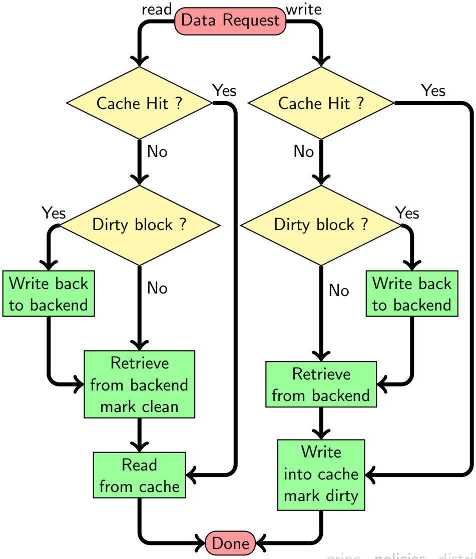
Write-miss policies

Write allocate

- datum at the missed-write location is loaded to cache, followed by a write-hit operation
- write misses are similar to read misses

No write allocate

- datum at the missed-write location is not loaded to cache
- it is written to the backend
- no cache for writes



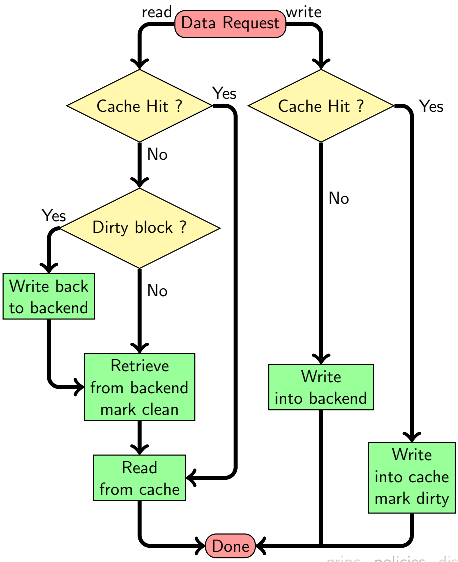
Write-miss policies

Write allocate

- datum at the missed-write location is loaded to cache, followed by a write-hit operation
- write misses are similar to read misses

No write allocate

- datum at the missed-write location is not loaded to cache
- it is written to the backend
- no cache for writes



Parallelization and caches

The problem

- you scale horizontally by adding more servers/processes
- each server has its own cache
- all caches will tend to have the same content
- so the cache size will not scale

Parallelization and caches

The problem

- you scale horizontally by adding more servers/processes
- each server has its own cache
- all caches will tend to have the same content
- so the cache size will not scale

Solutions

- use shared caches and tackle the synchonization issues
  - locally via shared memory
  - across servers via shared filesystem
- do nothing, you're fine with it
  - you have automatic replication of hot items
  - your caches are big enough

Cache consistency

The problem

- with distributed caches, you main have several copies of one data
- do you have the same version in all cases ?
  - e.g. in case it was just overwritten in one cache with write-back policy

Cache consistency

The problem

- with distributed caches, you main have several copies of one data
- do you have the same version in all cases ?
  - e.g. in case it was just overwritten in one cache with write-back policy

Solutions

- use write-through policy
- not caring
  - e.g. in case of web browsing for pretty static pages
- add cleverness
  - inform all caches when setting a dirty flag
  - syncing all caches for the given data when retrieving from backend

Conclusion

Conclusion

Key messages of the day

- Take care of latencies and use asynchronous I/O when needed
- In case of bad performance, optimize your I/O
- Make sure that you don't do more I/O than needed
- Caches may help at all levels