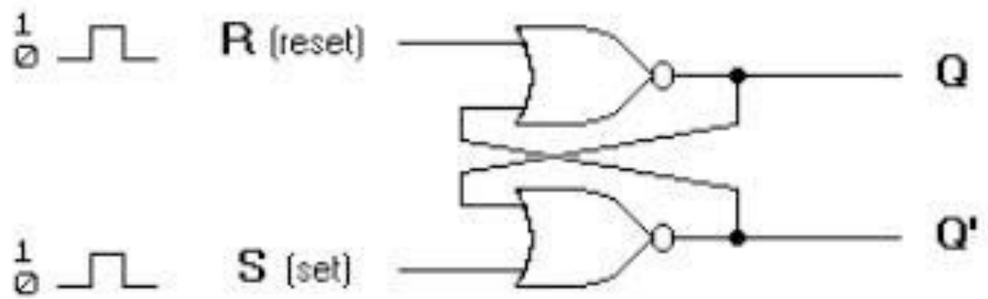
Hoofdstuk 14

Externe Opslag: Basisconcepten

Wat is data? Bits : 2 ≠ voltages



(a) Logic diagram

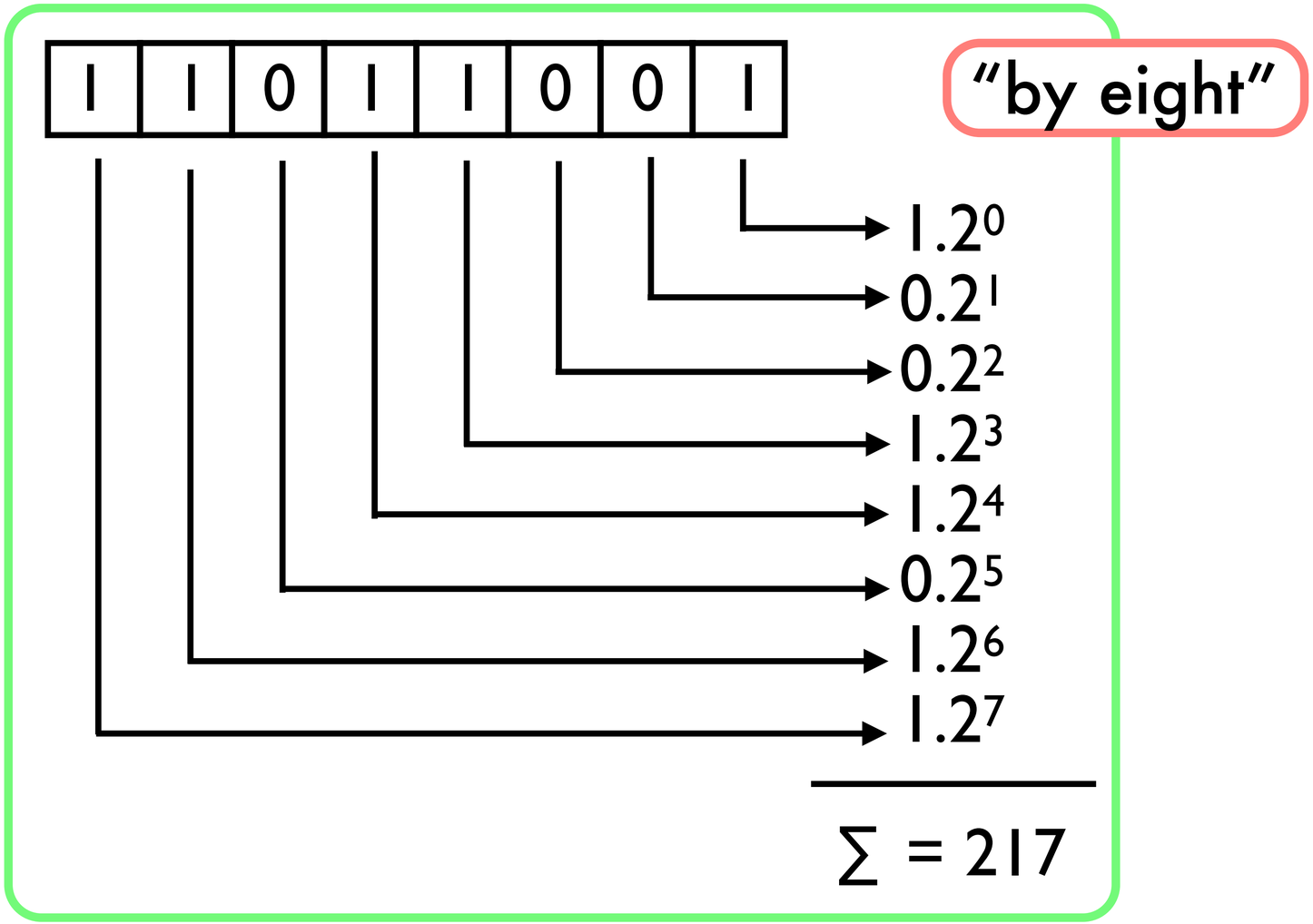
S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after S=1, R=0)
0	1	0	1	
0	0	0	1	(after S=0, R=1)
1	1	0	0	

(b) Truth table

Basic flip-flop circuit with NOR gates

"binary digit"

Wat is data? Bytes



Grootste byte: 255

256 ≠ waarden

Natuurlijke getallen als byterijen

```
> (define number 1234567)
> (modulo number 256)
135
> (set! number (quotient number 256))
> (modulo number 256)
214
> (set! number (quotient number 256))
> number
18
```

$$1234567 = 18 \cdot 256^2 + 214 \cdot 256^1 + 135 \cdot 256^0$$

18, 214, 135

big endian

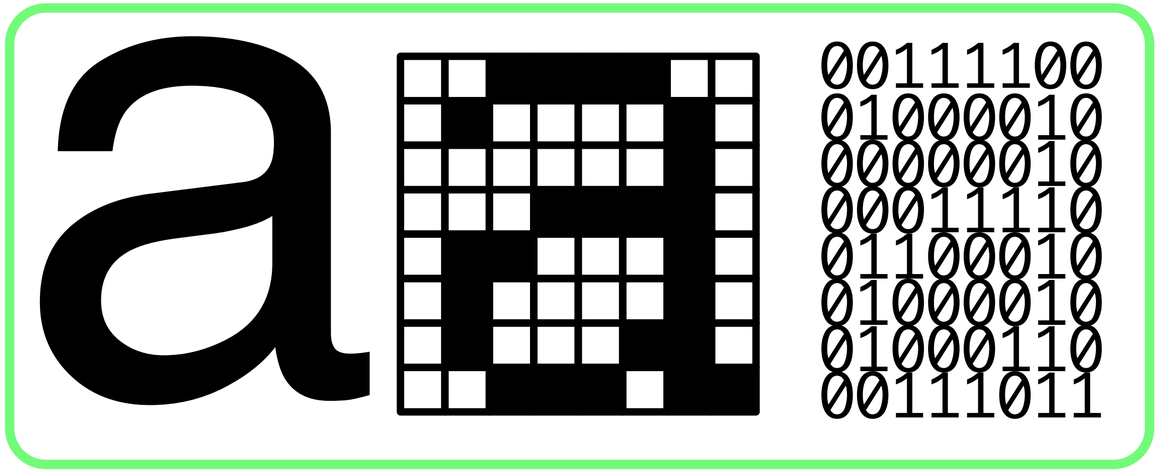
meest significante byte op eerste (laagste) geheugenadres

135, 214, 18

little endian

minst significante byte op eerste (laagste) geheugenadres

Grafische data als byterijen



60, 66, 2, 30, 98, 66, 70, 59

Terminologie

1024 bytes = 1 kilobyte
1024 kilobyte = 1 megabyte (1M)
1024 megabyte = 1 gigabyte (1G)
1024 gigabyte = 1 terabyte (1T)
1024 terabyte = 1 petabyte (1P)
1024 petabyte = 1 exabyte (1E)
1024 exabyte = 1 zettabyte (1Z)
1024 zettabyte = 1 yottabyte (1Y)

$1024 = 2^{10}$

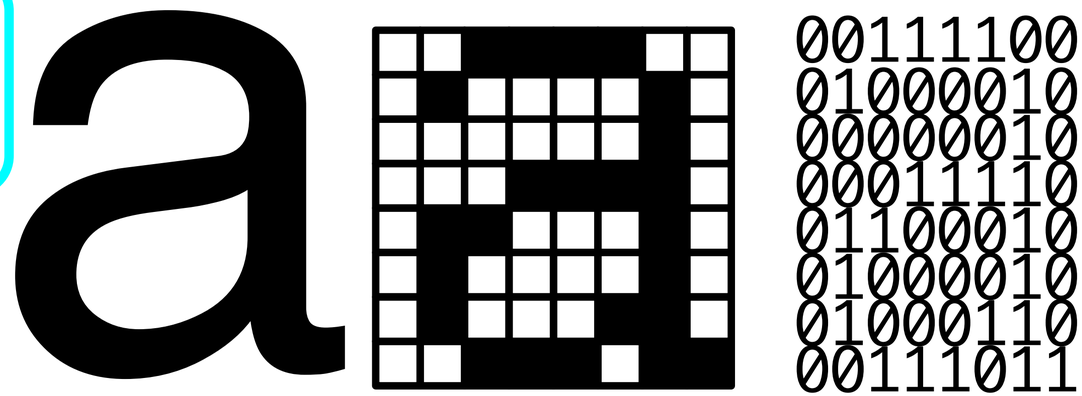
$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^{10} = 1024$
$2^{16} = 65536$

Computers
≈
gigabytes

Harddisks
≈
terabytes

Scheme's Bytevectoren

Nuttig om "rauwe" data voor te stellen




```
(define letter-a
  (let ((v (make-bytevector 8)))
    (bytevector-u8-set! v 0 60)
    (bytevector-u8-set! v 1 66)
    (bytevector-u8-set! v 2 2)
    (bytevector-u8-set! v 3 30)
    (bytevector-u8-set! v 4 98)
    (bytevector-u8-set! v 5 66)
    (bytevector-u8-set! v 6 70)
    (bytevector-u8-set! v 7 59)
    v))
```

Veel efficiënter dan gewone vectoren

values + type tag

```
> (bytevector-u8-ref letter-a 6)
70
```

make-bytevector, bytevector-u8-ref, bytevector-u8-set!

∈ R7RS

Getallen in Bytevectoren: plaats

```
(define (natural-bytes nmbr)
  (exact (ceiling (log (max (+ nmbr 1) 2) 256))))

(define (integer-bytes nmbr)
  (exact (ceiling (log (max (abs (+ (* 2 nmbr) 1)) 2) 256))))
```

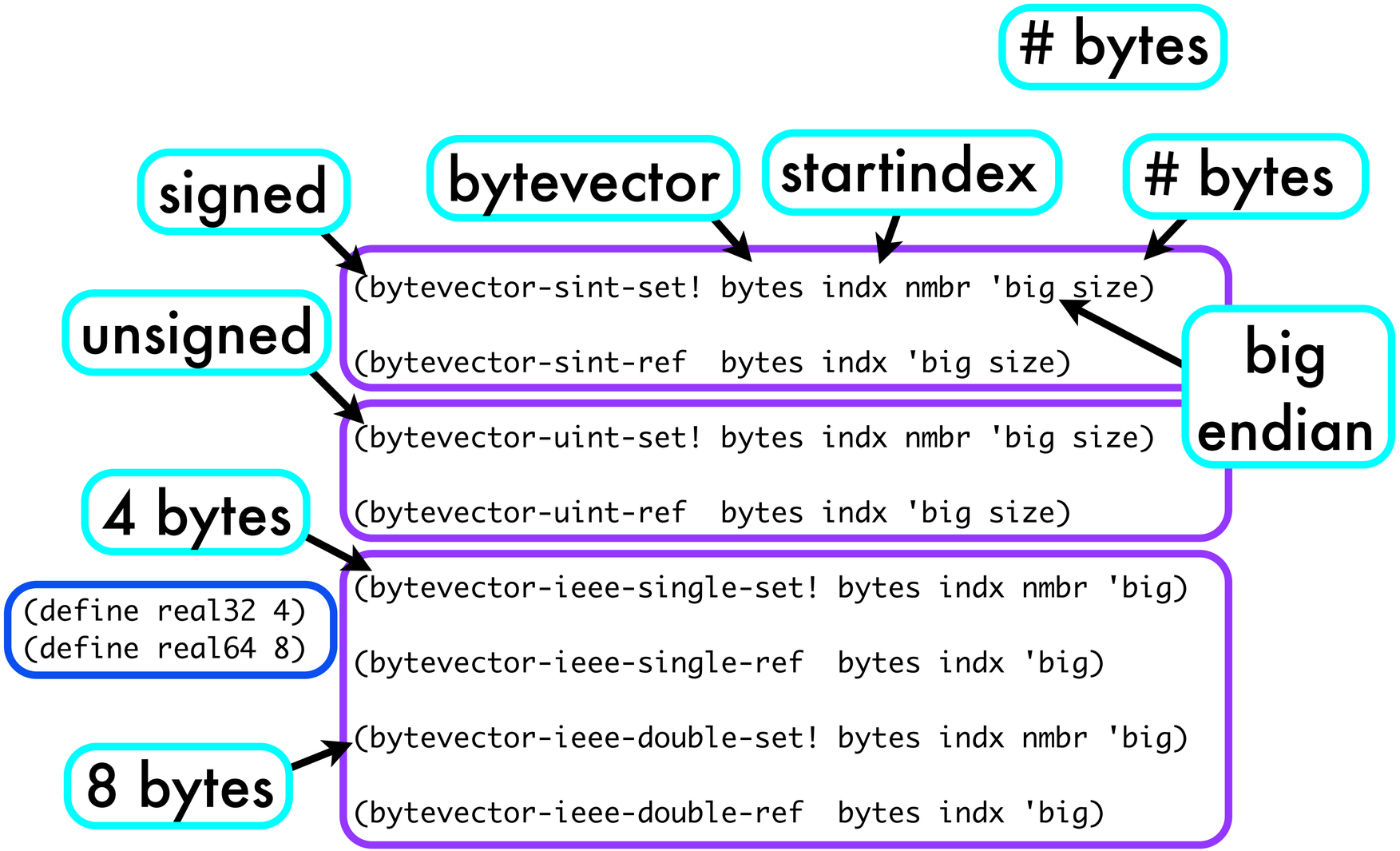
```
> (natural-bytes 0)
1
> (natural-bytes 1)
1
> (natural-bytes 77)
1
> (natural-bytes 255)
1
> (natural-bytes 256)
2
> (natural-bytes 1234567)
3
```

```
> (integer-bytes 1)
1
> (integer-bytes 127)
1
> (integer-bytes 128)
2
> (integer-bytes -127)
1
> (integer-bytes -128)
1
> (integer-bytes -129)
2
```

```
> (log 123 256)
0.867814313167405
> (ceiling (log 123 256))
1.0
> (ceiling (log 1234567 256))
3.0
> (ceiling (log 1 256))
0
> (ceiling (log 0 256))
log: division by zero
```

```
(import (scheme inexact))
```

Getallen in Bytevectoren 2/2



Strings in Bytevectoren

```
(string->utf8 str)

(utf8->string bytes)
```

```
> (bytevector-length (string->utf8 "hello"))
5
> (bytevector-length (string->utf8 "Здравствуйте!"))
25
> (bytevector-length (string->utf8 "你好"))
6
```

```
(define (utf8-sentinel-for nmbr-byts)
  (define byts (make-bytevector nmbr-byts))
  (define (fill! offset rem)
    (cond ((= rem 1)
           (bytevector-u8-set! byts offset 127))
          ((= rem 2)
           (bytevector-u8-set! byts offset 223)
           (bytevector-u8-set! byts (+ offset 1) 191))
          (else
           (bytevector-u8-set! byts offset 239)
           (bytevector-u8-set! byts (+ offset 1) 191)
           (bytevector-u8-set! byts (+ offset 2) 191)
           (if (> rem 3)
               (fill! (+ offset 3) (- rem 3))))))
  (fill! 0 nmbr-byts)
  (utf8->string byts))
```

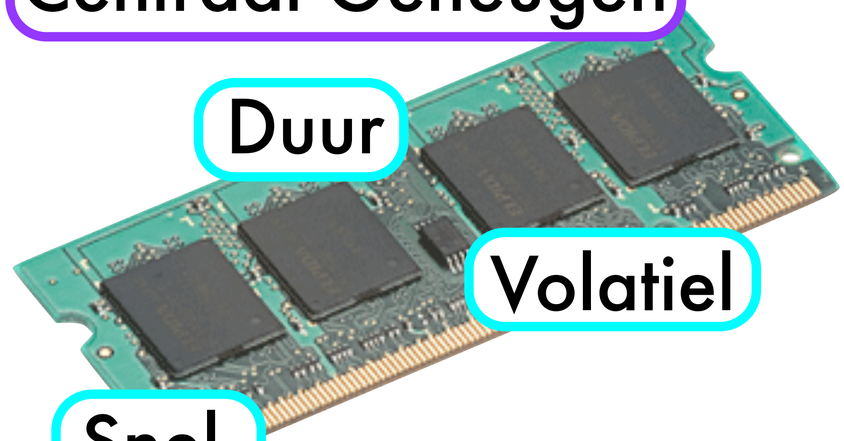
Later nodig...

Iedere lengte heeft zijn eigen +∞

```
> (string<? "д" (utf8-sentinel-for 2))
#t
> (string<? "д" (utf8-sentinel-for 1))
#f
```

Waar zitten die bytes?

Centraal Geheugen



Duur

Volatiel

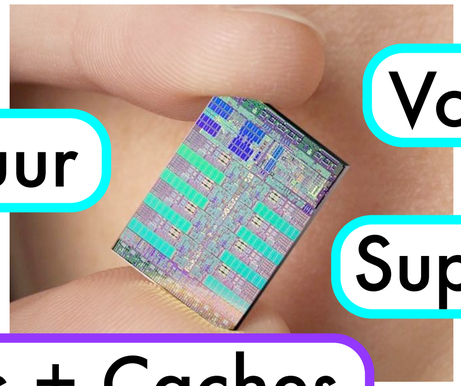
Snel

Periferisch Geheugen

Persistent

Traag

Supergoedkoop



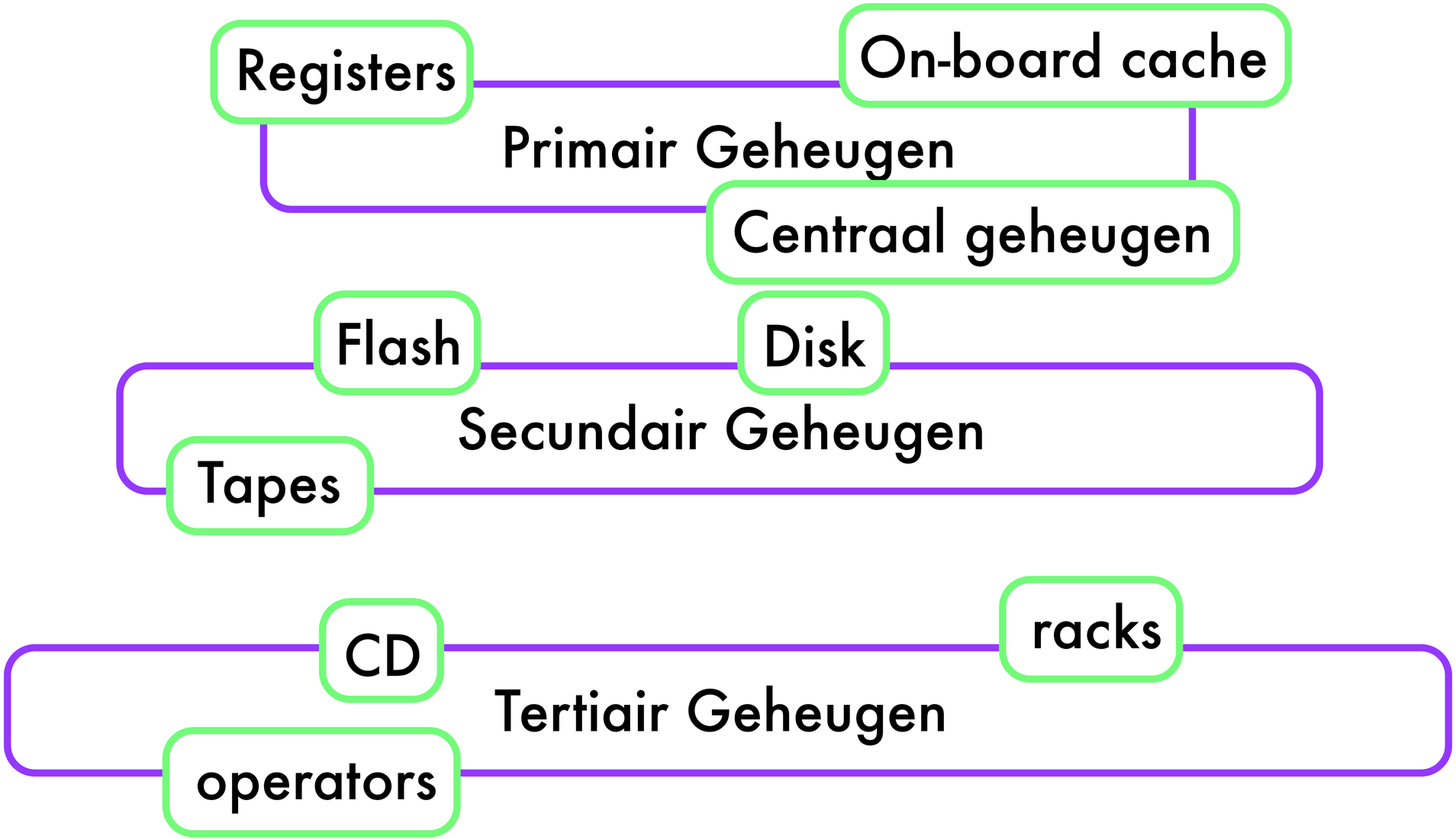
Zeer duur

Volatiel

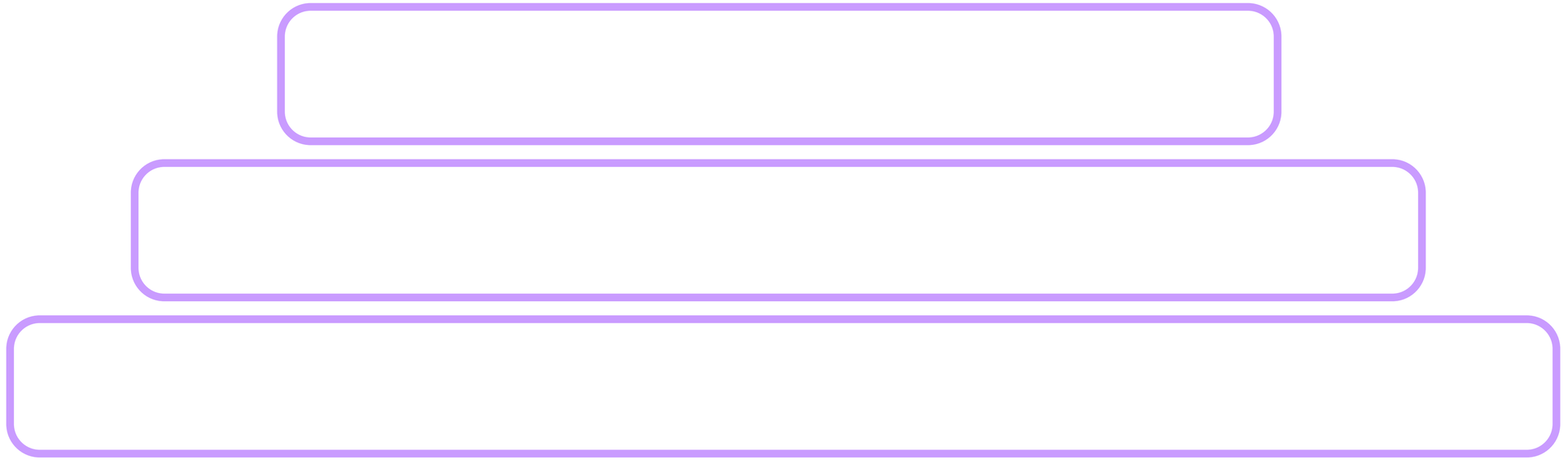
Supersnel

Registers + Caches

De Geheugenhiërarchie



De Geheugenhiërarchie



Verschil in opslagcapaciteit >3 grootteordes

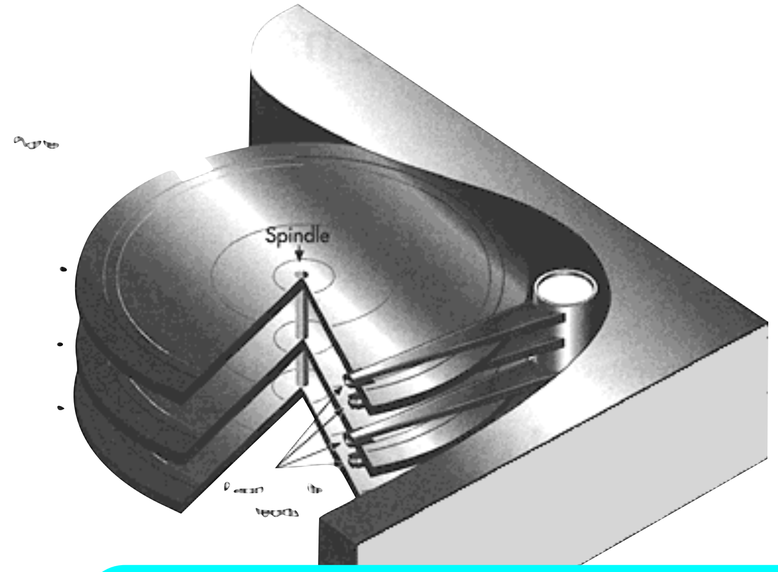
Verschil in toegangstijden >5 grootteordes

Verschil in cost/byte >3 grootteordes

Disks

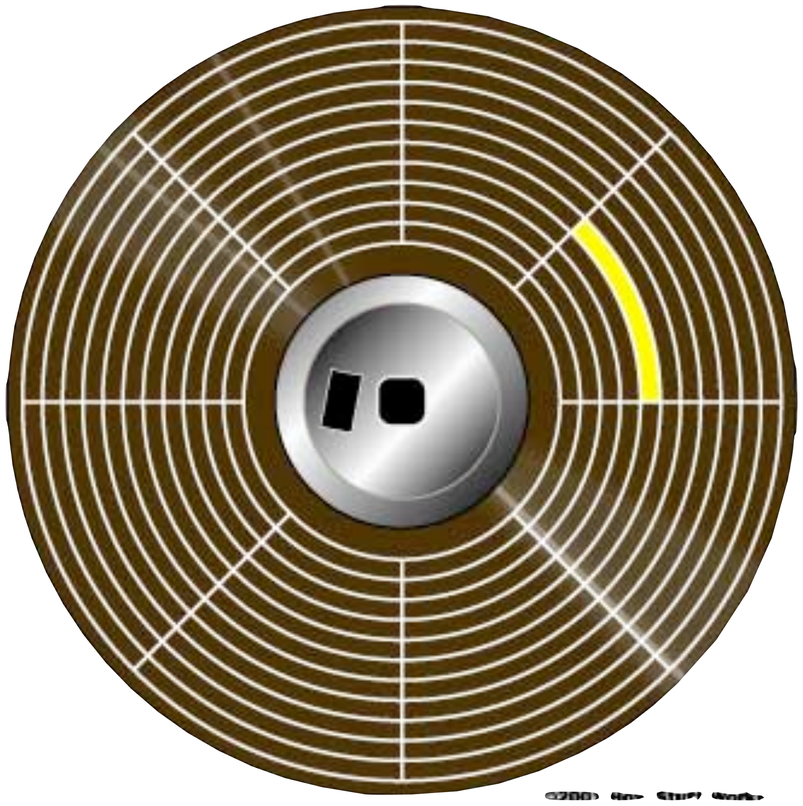
Een as met schijven

Een lees- en schrijfkop



Magnetiseerbaar materiaal + hysterese

Geformateerde Disk

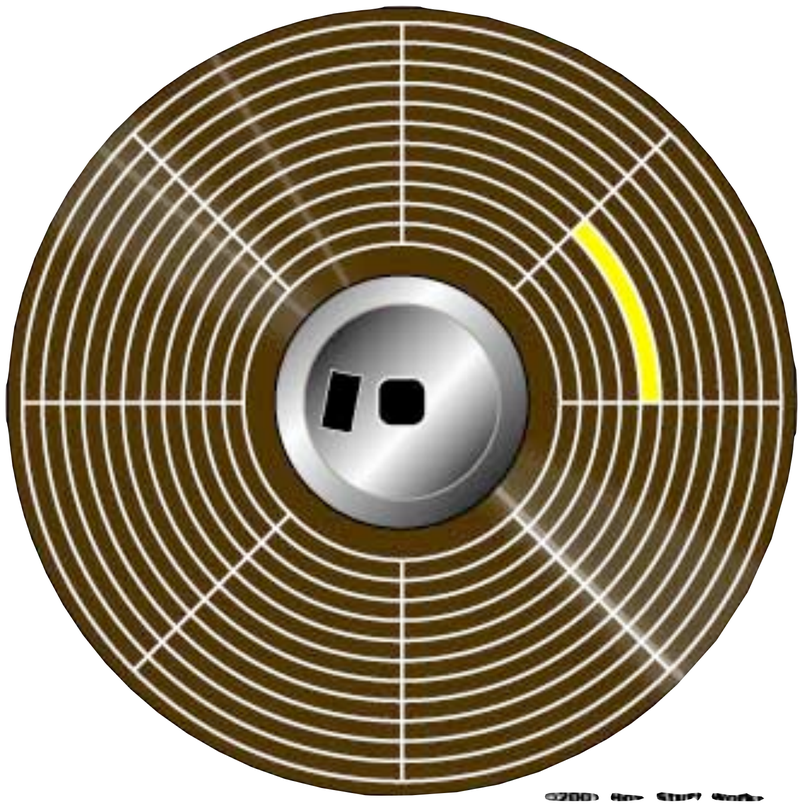


Georganiseerd in tracks en sectoren. Op elke track ∩ sector bevindt zich een blok.

Een blok bevat typisch tussen de 512 bytes en 4K data.

Een blok is de lees- en schrijfeenheid.

Een blok lezen of schrijven



Fase 1: De kop bewegen naar de juiste track.

Fase 2: Wachten tot de juiste sector passeert.

Fase 3: Het blok effectief lezen of schrijven.

$$T = T_{seek} + T_{latency} + T_{transfer}$$

Disk Abstractie

```
ADT disk

block-size
  number
disk-size
  number
block-ptr-size
  number
block-idx-size
  number
new
  ( string → disk )
mount
  ( string → disk )
unmount
  ( disk → ∅ )
disk?
  ( any → boolean )
name
  ( disk → string )
read-block
  ( disk number → block )
```

Een groep blokken met nummer ∈ [0 .. disksize-1]

Iedere blok bevat block-size bytes

```
(define block-size 50)
(define disk-size  200)
(define block-ptr-size (natural-bytes disk-size))
(define block-idx-size (natural-bytes block-size))
```

aantal blocks

size van pointer naar block binnen disk

size van pointer naar offset binnen block

Elk bloknummer past in block-ptr-size bytes

Elke blokindex past in block-idx-size bytes

Disk (image): Implementatie

```
(define-record-type disk
  (make-disk n)
  disk?
  (n name))
```

Bezet een stuk van de echte disk

```
(define (new name)
  (define port (open-output-file name #:exists 'truncate))
  (define zeroes (make-bytevector block-size 0))
  (let low-level-format
    ((block-nr 0))
    (write-bytevector zeroes port)
     (if (< (+ 1 block-nr) disk-size)
         (low-level-format (+ 1 block-nr))))
     (close-port port)
     (make-disk name))

(define (mount name)
  (make-disk name))

(define (unmount dsk)
  '())
```

Maakt de blokken aan

open-output-file, make-bytevector
write-bytevector, close-port

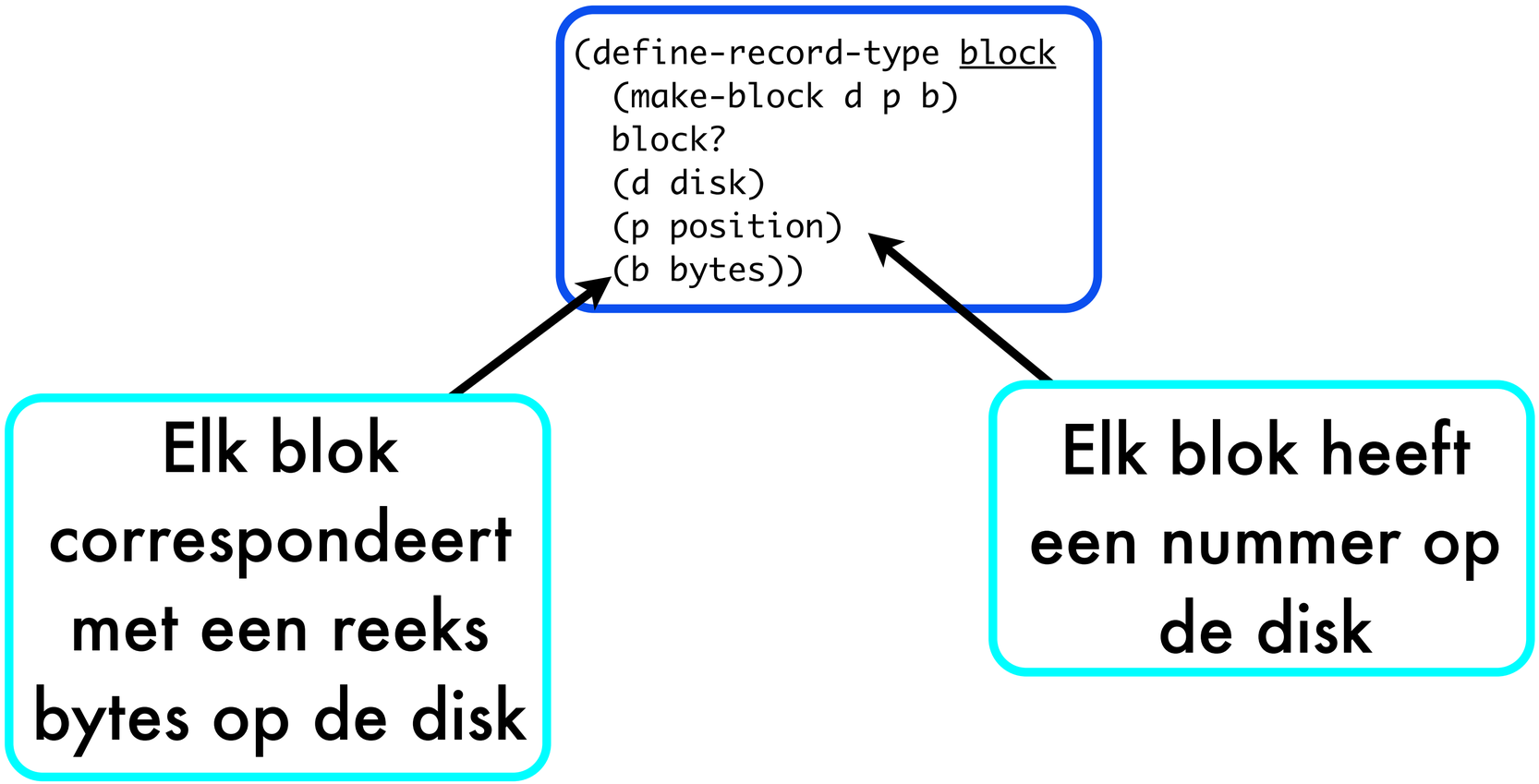
∈ R7RS

Block Abstractie

```
ADT block

write-block!
  ( block → ∅ )
block?
  ( any → boolean )
disk
  ( block → disk )
position
  ( block → number )
```

```
decode-byte
  ( block number → byte )
encode-byte!
  ( block number byte → ∅ )
decode-string
  ( block number number → string )
encode-string!
  ( block number number string → ∅ )
decode-fixed-natural
  ( block number number → natural )
encode-fixed-natural!
  ( block number number natural → ∅ )
decode-arbitrary-integer
  ( block number → integer × number )
encode-arbitrary-integer!
  ( block number integer → number )
decode-real
  ( block number number → real )
encode-real!
  ( block number number real  → number )
decode-bytes
  ( block bytevector number number number → ∅ )
encode-bytes!
  ( block bytevector number number number → ∅ )
```

Representatie van Blokken



Lezen en Schrijven van een Blok

```
(define (read-block dsk bptr)
  (define port (open-input-file (name dsk)))
  (set-port-position! port (* bptr block-size))
  (let ((byts (read-bytevector block-size port)))
    (close-port port)
    (make-block dsk bptr byts)))

 (define (write-block! blck)
   (define bptr (position blck))
   (define data-byts (bytes blck))
   (define port (open-output-file (name (disk blck)) #:exists 'update))
   (set-port-position! port (* bptr block-size))
   (write-bytevector port data-byts)
   (close-port port))
```

Zet de “current” van de output port op de juiste byte

Lees/schrijf de bytevector horend bij het blok

open-input-file, open-output-file, set-port-position!, read-bytevector, write-bytevector, close-port

∈ R7RS

En/De-coderen van data(1)

```
(define (encode-byte! blck offs byte)
  (bytevector-u8-set! (bytes blck) offs byte))

(define (decode-byte blck offs)
  (bytevector-u8-ref (bytes blck) offs))
```

`offs` = offset in het blok

```
(define (encode-fixed-natural! blck offs size nmbr)
  (bytevector-uint-set! (bytes blck) offs nmbr 'big size))

(define (decode-fixed-natural blck offs size)
  (bytevector-uint-ref (bytes blck) offs 'big size))
```

Lengte wordt ook opgeslagen

```
(define (encode-arbitrary-integer! blck offs nmbr)
  (define size (integer-bytes nmbr))
  (encode-byte! blck offs size)
  (bytevector-sint-set! (bytes blck) (+ offs 1) nmbr 'big size)
  (+ size 1))

(define (decode-arbitrary-integer blck offs)
  (define size (decode-byte blck offs))
  (define nmbr (bytevector-sint-ref (bytes blck) (+ offs 1) 'big size))
  (cons nmbr (+ offs size 1)))
```

En/De-coderen van data(2)

```
(define (encode-real! blck offs size nmbr)
  (cond ((= size real64)
         (bytevector-ieee-double-set! (bytes blck) offs nmbr 'big)
         size)
        ((= size real32)
         (bytevector-ieee-single-set! (bytes blck) offs nmbr 'big)
          size)
        (else
         (error "illegal real size" size))))

(define (decode-real blck offs size)
  (cond ((= size real64)
         (bytevector-ieee-double-ref (bytes blck) offs 'big))
        ((= size real32)
         (bytevector-ieee-single-ref (bytes blck) offs 'big))
        (else
         (error "illegal real size" size))))
```

```
(define real32 4)
(define real64 8)
```

En/De-coderen van data(3)

```
(define (encode-string! blck offs size strg)
  (set! strg (string->utf8 strg))
  (do ((indx 0 (+ indx 1)))
    ((= indx size))
    (let ((byte (if (< indx (bytevector-length strg))
                    (bytevector-u8-ref strg indx)
                    str-end-byte)))
      (encode-byte! blck (+ offs indx) byte))))

(define (decode-string blck offs size)
  (let ((bytevector-u8-set!
         (lambda (bv indx val)
           (bytevector-u8-set! bv indx val)
           bv)))
    (utf8->string
     (let loop
       ((indx 0))
       (if (< indx size)
           (let ((byte (decode-byte blck (+ offs indx))))
             (if (eq? byte str-end-byte)
                 (make-bytevector indx 0)
                 (bytevector-u8-set! (loop (+ indx 1)) indx byte)))
           (make-bytevector indx 0))))))
```

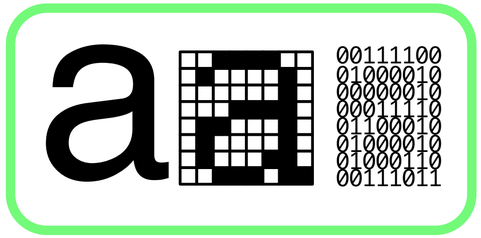
Kopieer alle bytes en vul aan met 0'en

Padding

```
(define str-end-byte 0)
```

Blijf uitlezen tot je een 0 ziet

En/De-coderen van rauwe data(4)




```
(define (encode-bytes! blck byts blck-offs u8-offs size)
  (do ((indx 0 (+ indx 1)))
    ((= indx size))
    (encode-byte! blck
                  (+ blck-offs indx)
                  (bytevector-u8-ref byts (+ u8-offs indx)))))

(define (decode-bytes blck byts blck-offs u8-offs size)
  (do ((indx 0 (+ indx 1)))
    ((= indx size))
    (bytevector-u8-set! byts
                        (+ u8-offs indx)
                        (decode-byte blck (+ blck-offs indx)))))
```

Kopieer byts in het blck

Inverse operatie

Het Filesysteem

Unix

Mac OS X

DOS

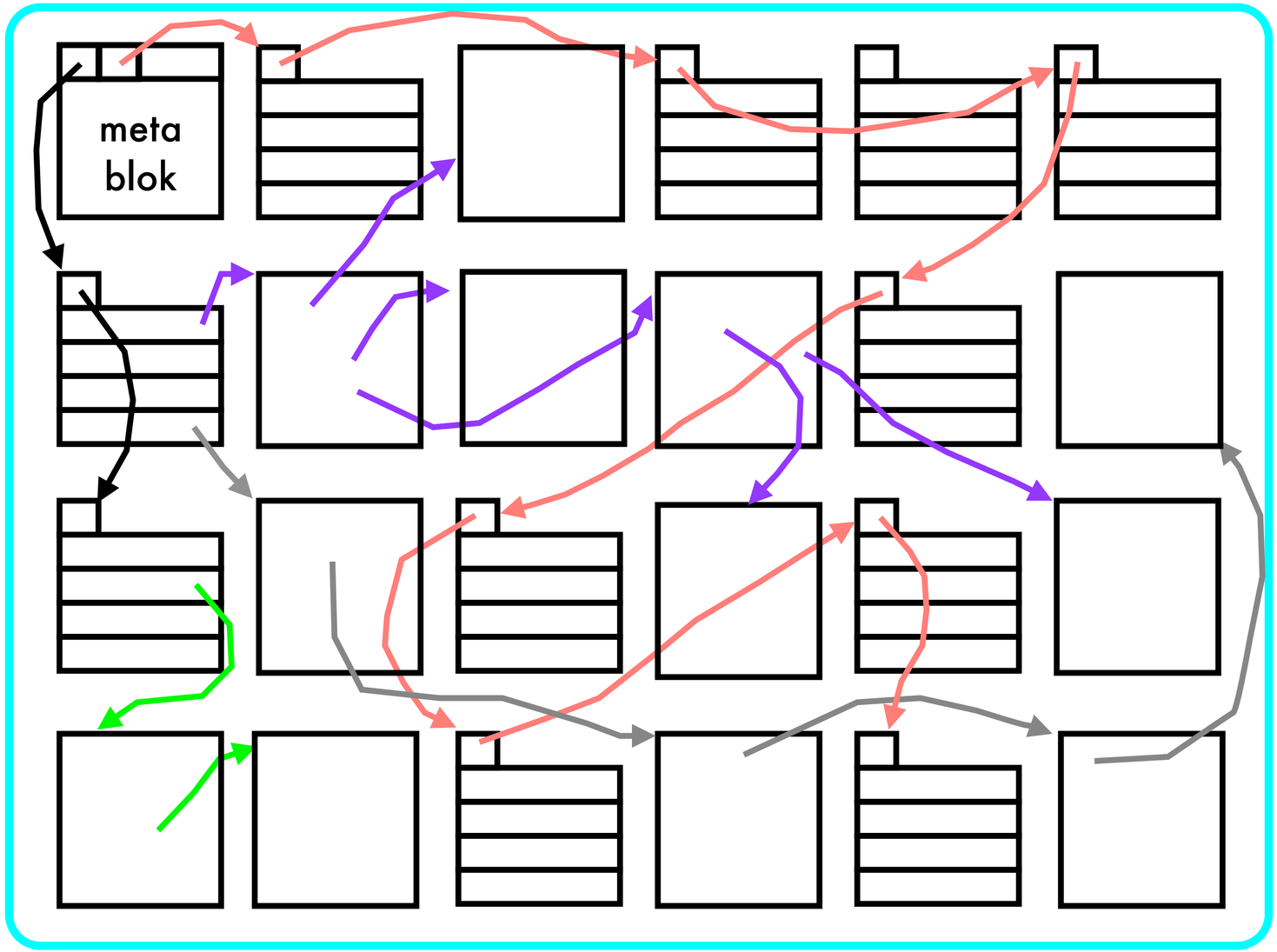
Data wordt georganiseerd in allerlei structuren die ontstaan door blokken aan elkaar te linken. Dit gebeurt door de blokpointer van het ene blok te encoderen als bytes in het andere blok. Een reeks aan elkaar gelinkte blokken noemen we een file.

Sommige blokken behoren tot één of andere file; andere zijn vrij. De vrije blokken worden bijgehouden in een freelist.

Alles begint bij het metablok

De files geven we een naam. De directory is een lijst van filenamen met bloknummer van het begin van de file

Organisatie van de Disk



Filesysteem: Constanten

```
(define filename-size 10)

(define (cap-name name)
  (if (> (string-length name) filename-size)
      (substring name 0 filename-size)
      name))

(define sentinel-filename (utf8-sentinel-for filename-size))
```

+∞

```
(define null-block 0)
(define meta-bptr  null-block)

(define (null-block? bptr)
  (= bptr null-block))
```

0 is het bloknummer van het metablok en dient dus ook als ()

mag geen onderdeel van file zijn

```
(define (read-meta-block disk)
  (disk:read-block disk meta-bptr))

(define (write-meta-block! meta)
  (disk:write-block! meta))
```

Enkel-Linken van Blokken

```
(define next-offset 0)

(define (next-bptr blck)
  (disk:decode-fixed-natural blck next-offset disk:block-ptr-size))

(define (next-bptr! blck bptr)
  (disk:encode-fixed-natural! blck next-offset disk:block-ptr-size bptr))
```

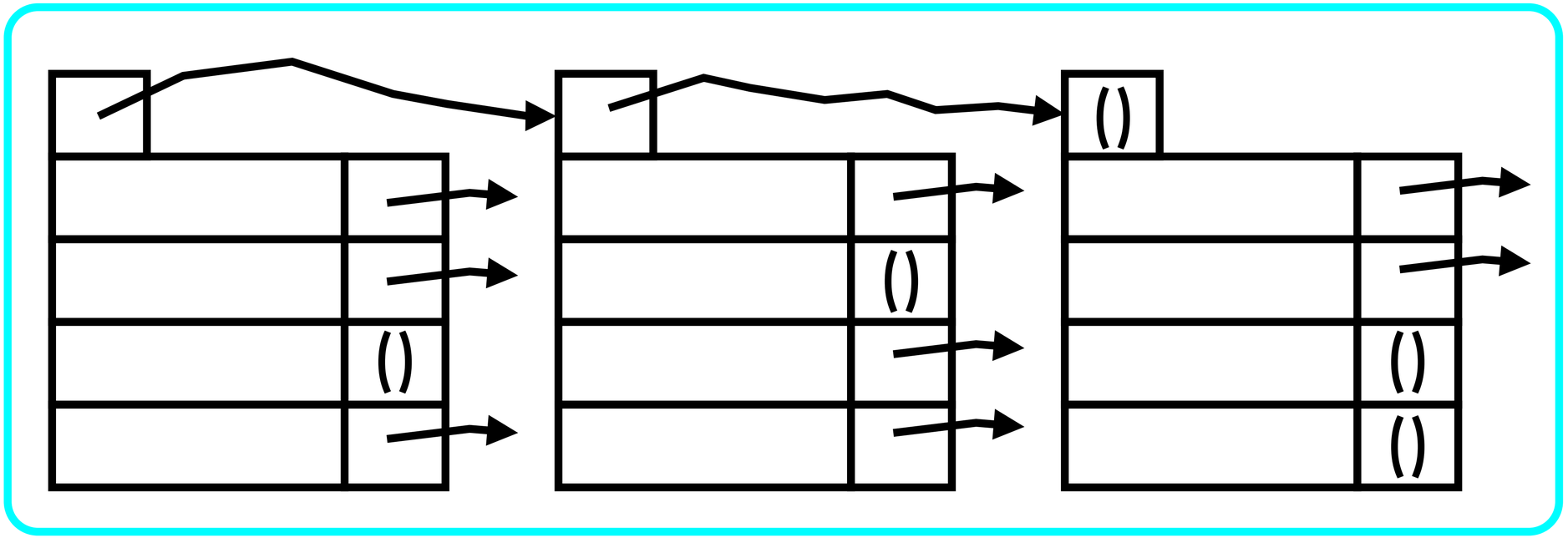
```
(define (has-next? blck)
  (not (null-block? (next-bptr blck))))
```

Layout van het Metablok

```
(define directory-offset 0)
(define freelist-offset  disk:block-ptr-size)
(define available-offset (* 2 disk:block-ptr-size))

(define (directory meta)
  (disk:decode-fixed-natural  meta directory-offset disk:block-ptr-size))
(define (directory! meta blck)
  (disk:encode-fixed-natural! meta directory-offset disk:block-ptr-size blck))

(define (freelist meta)
  (disk:decode-fixed-natural  meta freelist-offset  disk:block-ptr-size))
(define (freelist! meta flst)
  (disk:encode-fixed-natural! meta freelist-offset  disk:block-ptr-size flst))

(define (blocks-free meta)
  (disk:decode-fixed-natural  meta available-offset disk:block-ptr-size))
(define (blocks-free! meta free)
  (disk:encode-fixed-natural! meta available-offset disk:block-ptr-size free))
```

Structuur van de Directory

(name,bptr) - slots



Enkel volledig lege blokken worden opnieuw in de freelist gehangen

Layout van één Directoryblok

```
(define slot-size       (+ filename-size disk:block-ptr-size))
(define nr-of-dir-slots (quotient (- disk:block-size disk:block-ptr-size)
                                  slot-size))

(define (dir-name/bptr! blck slot name bptr)
  (define offn (+ disk:block-ptr-size (* slot slot-size)) )
  (define offp (+ disk:block-ptr-size (* slot slot-size) filename-size))
  (disk:encode-string! blck offn filename-size name)
  (disk:encode-fixed-natural! blck offp disk:block-ptr-size bptr))

(define (dir-name blck slot)
  (define offn (+ disk:block-ptr-size (* slot slot-size)))
  (disk:decode-string blck offn filename-size))

(define (dir-bptr blck slot)
  (define offp (+ disk:block-ptr-size (* slot slot-size) filename-size))
  (disk:decode-fixed-natural blck offp disk:block-ptr-size))
```

(name,bptr) - slots

```
(define (empty-slot? blck slot)
  (string=? sentinel-filename (dir-name blck slot)))

(define (at-end? blck slot)
  (= slot nr-of-dir-slots))
```

High-level Format van de Disk

Het metablok bevat een pointer naar de directory, naar de freelist en houdt het aantal vrije blokken bij

Het metablok zélf is niet vrij

Ieder blok in de freelist bevat het bloknummer van zijn opvolger

```
(define (format! disk)
  (define meta (read-meta-block disk))
  (directory!   meta null-block)
  (freelist!    meta (+ meta-bptr 1))
  (blocks-free! meta (- disk:disk-size 1))
  (write-meta-block! meta)
  (let high-level-format
    ((bptr (+ meta-bptr 1)))
    (let ((block (disk:read-block disk bptr)))
      (cond ((< (+ bptr 1) disk:disk-size)
             (next-bptr! block (+ bptr 1))
             (disk:write-block! block)
             (high-level-format (+ bptr 1)))
            (else
             (next-bptr! block null-block)
             (disk:write-block! block)))))
  disk)
```

Beheer van Vrije Blokken (1)

```
(define (new-block disk)
  (define meta (read-meta-block disk))
  (define flst (freelist meta))
  (if (null-block? flst)
      (error "disk full! (new-block)" disk)
      (let ((blck (disk:read-block disk flst)))
        (blocks-free! meta (- (blocks-free meta) 1))
        (freelist!    meta (next-bptr blck))
        (write-meta-block! meta)
        blck)))
```

Een blok uit de freelist “poppen”: O(1)

```
(define (fresh-block! disk next!)
  (define next (new-block disk))
  (next-bptr! next null-block)
  (do ((slot 0 (+ slot 1)))
    ((at-end? next slot)
     (next! next)
     next)
    (dir-name/bptr! next slot sentinel-filename null-block)))
```

Een blok alloceren, initialiseren en vastlinken

Beheer van Vrije Blokken (2)

Een blok in de freelist "pushen": O(1)

```
(define (delete-block blck)
  (define disk (disk:disk blck))
  (define meta (read-meta-block disk))
  (next-bptr! blck (freelist meta))
  (disk:write-block!  blck)
  (freelist!    meta (disk:position blck))
  (blocks-free! meta (+ (blocks-free meta) 1))
  (write-meta-block! meta))
```

```
(define (delete-chain! disk bptr)
  (unless (null-block? bptr)
    (let* ((blck (disk:read-block disk bptr))
           (next (next-bptr blck)))
      (delete-block blck)
      (delete-chain! disk next))))
```

Een hele lijst van blokken vrijgeven

Filesysteem Operatie #1: mk

```
(define (mk disk name bptr)
  (define meta (read-meta-block disk))
  (let loop-dir
    ((dptr (directory meta))
     (new! (lambda (newb)
             (let ((meta (read-meta-block disk)))
               (directory! meta (disk:position newb))
               (write-meta-block! meta)))))
    (let ((blck (if (null-block? dptr)
                  (fresh-block! disk new!)
                  (disk:read-block disk dptr))))
      (let loop-block
        ((slot 0))
        (cond ((at-end? blck slot)
               (loop-dir (next-bptr blck)
                         (lambda (newb)
                          (next-bptr! blck (disk:position newb))
                           (disk:write-block! blck))))
              ((empty-slot? blck slot)
               (dir-name/bptr! blck slot name bptr)
               (disk:write-block! blck))
              (else
               (loop-block (+ slot 1))))))))
```

Een slot aanmaken in de directory

outer loop: directory blokken

inner loop: slot zoeken in een blok

Filesysteem Operatie #2: rm

```
(define (rm disk name)
  (define meta (read-meta-block disk))
  (set! name (cap-name name))
  (let loop-dir
    ((bptr (directory meta))
     (nxt! (lambda (next)
             (directory! meta next)
             (write-meta-block! meta))))
    (let ((blck (if (null-block? bptr)
                    (error "file not found (rm)" name)
                    (disk:read-block disk bptr))))
      (let loop-block
        ((slot 0)
         (seen #f))
        (cond ((at-end? blck slot)
               (loop-dir (next-bptr blck) (lambda (next)
                                            (next-bptr! blck next)
                                            (disk:write-block! blck))))
              ((empty-slot? blck slot)
               (loop-block (+ slot 1) seen))
              ((string=? name (dir-name blck slot))
               (dir-name/bptr! blck slot sentinel-filename null-block)
               (disk:write-block! blck)
               (if (not seen)
                   (maybe-delete-block! blck slot nxt!)))
              (else
               (loop-block (+ slot 1) #t)))))))
```

Een slot verwijderen uit de directory

outer loop: directoryblokken

inner loop: slot zoeken in een blok

Filesysteem Operatie #2: rm (ctd)

```
(define (maybe-delete-block! blck slot next!)
  (cond
    ((at-end? blck slot)
     (next! (next-bptr blck))
     (delete-block blck))
    ((empty-slot? blck slot)
     (maybe-delete-block! blck (+ slot 1) next!))))
```

Operaties #3&4: whereis, ls

```
(define (ls disk)
  (define meta (read-meta-block disk))
  (define bptr (directory meta))
  (if (null-block? bptr)
      ()
      (let traverse-dir
```

De directory opsommen als lijst

```
(define (whereis disk name)
  (define meta (read-meta-block disk))
  (define bptr (directory meta))
  (set!   name (cap-name name))
  (if (null-block? bptr)
      0
      (let traverse-dir
        ((blck (disk:read-block disk bptr))
         (slot 0))
        (cond ((at-end? blck slot)
               (if (has-next? blck)
                   (traverse-dir (disk:read-block disk (next-bptr blck)) 0)
                   null-block))
              ((string=? name (dir-name blck slot))
               (dir-bptr blck slot))
              (else
               (traverse-dir blck (+ slot 1)))))))
```

Een file opzoeken in de directory

Sequentiële Files

Populairste soort file

Lange rijen van Scheme waarden op disk

```
ADT output-file

new
  ( disk string → output-file )
name
  ( output-file → string )
open-write!
  ( disk string → output-file )
close-write!
  ( output-file → ∅ )
reread!
  ( output-file → input-file )
write!
  ( output-file any → ∅ )
delete!
  ( output-file → ∅ )
```

```
ADT input-file

name
  ( input-file → string )
open-read!
  ( disk string → input-file )
rewrite!
   ( input-file → output-file )
close-read!
  ( input-file → ∅ )
has-more?
  ( input-file → boolean )
read
  ( input-file → any )
peek
  ( input-file → any )
delete!
  ( input-file → ∅ )
```

reread! en rewrite!
"spoelen de file terug"

Voorbeeld

```
(define d (disk:new "My Computer"))
(fs:format! d)

(define f (out:new d "TestFile"))

(out:write! f 3.14)
(out:write! f 42)
(out:write! f "Done!")

(out:close-write! f)

(set! f (in:open-read! d "TestFile"))
(display (in:read f))(newline)
(display (in:read f))(newline)
(display (in:read f))(newline)

(in:close-read! f)
```

```
Welcome to DrRacket, version 5.1.3 [3m].
Language: r6rs [custom].
3.14
42
Done!
>
```

Duale Representatie

```
(define-record-type sequential-file
  (make d n h b)
  sequential-file?
  (d disk)
  (n name)
  (h header header!)
  (b buffer buffer!))
```

De header bevat een referentie naar het 1ste blok v/d file. De buffer is een kopie in centraal geheugen van één blok van de file. Slechts 2 blokken in het geheugen!

Header

```
(define frst-offs 0)
(define curr-offs disk:block-ptr-size)

(define (first hder)
  (disk:decode-fixed-natural  hder frst-offs disk:block-ptr-size))
(define (first! hder bptr)
  (disk:encode-fixed-natural! hder frst-offs disk:block-ptr-size bptr))
(define (current hder)
  (disk:decode-fixed-natural  hder curr-offs disk:block-idx-size))
(define (current! hder offs)
  (disk:encode-fixed-natural! hder curr-offs disk:block-idx-size offs))
```

Creëren / Openen

```
(define (new disk name)
  (define hder (fs:new-block disk))
  (define bffr (fs:new-block disk))
  (define file (make disk name hder bffr))
  (fs:mk disk name (disk:position hder))
  (first!  hder (disk:position bffr))
  (fs:next-bptr! bffr fs:null-block)
  (current! hder disk:block-ptr-size)
  (disk:write-block! hder)
  file)
```

Header maken & registreren in de directory

Header opzoeken in de directory

```
(define (open-write! disk name)
  (define bptr (fs:whereis disk name))
  (define hder (disk:read-block disk bptr))
  (define fptr (first hder))
  (define bffr (disk:read-block disk fptr))
  (define file (make disk name hder bffr))
  (current! hder disk:block-ptr-size)
  file)
```

```
(define (open-read! disk name)
  (define bptr (fs:whereis disk name))
  (define hder (disk:read-block disk bptr))
  (define fptr (first hder))
  (define bffr (disk:read-block disk fptr))
  (define file (make disk name hder bffr))
  (current! hder disk:block-ptr-size)
  file)
```

Current: next pointer overslaan

Sluiten / Deleten

```
(define (close-read! file)
  ()); nothing to write
```

```
(define (block-bytes-free file)
  (define hder (header file))
  (define curr (current hder))
  (- disk:block-size curr))

(define (close-write! file)
  (define hder (header file))
  (define bffr (buffer file))
  (define fdsk (disk file))
  (disk:write-block! hder)
  (if (> (block-bytes-free file) 0)
      (disk:encode-byte! bffr (current hder) eof-tag))
  (let ((rest-list (fs:next-bptr bffr)))
    (fs:next-bptr! bffr fs:null-block)
    (fs:delete-chain! fdsk rest-list))
  (disk:write-block! bffr))
```

Sluit eventueel de buffer netjes af, schrijf weg en ruim ongebruikte blokken op

```
(define (delete! file)
  (define fnam (name file))
  (define hder (header file))
  (define fdsk (disk file))
  (fs:delete-chain! fdsk (first hder))
  (fs:delete-block hder)
  (fs:rm fdsk fnam))
```

Terugspoelen

```
(define (reread! file)
  (define dsk (disk file))
  (define hder (header file))
  (define fptr (first hder))
  (close-write! file)
  (buffer! file (disk:read-block dsk fptr))
  (current! hder disk:block-pointer-size)
  file)
```

Current: next pointer overslaan

```
(define (rewrite! file)
  (define fdsk (disk file))
  (define hder (header file))
  (define fptr (first hder))
  (close-read! file)
  (buffer!  file (disk:read-block fdsk fptr))
  (current! hder disk:block-ptr-size)
  file)
```

Schrijven

Encodeer het type van de waarde in 1 byte

Schrijf de waarde achter die byte

```
(define (write! file sval)
  (cond ((natural? sval)
         (write-type-tag file natural-tag)
         (write-natural file (exact sval)))
        ((integer? sval)
         (write-type-tag file integer-tag)
         (write-integer  file (exact sval)))
        ((real? sval)
         (write-type-tag file decimal-tag)
         (write-real     file sval))
        ((string? sval)
         (write-type-tag file string-tag)
         (write-string   file sval))
        (error "unsupported type (write!)" sval)))
```

```
(define natural-tag 0)
(define integer-tag 1)
(define decimal-tag 2)
(define string-tag  3)
(define eof-tag 255)
(define eob-tag 254)
```

Schrijven van ≠ datatypes

```
(define (write-type-tag file ttag)
  (claim-bytes! file 1)
  (let* ((bffr (buffer file))
         (hder (header file))
         (curr (current hder)))
    (disk:encode-byte! bffr curr ttag)
    (current! hder (+ curr 1))))
```

Het type

```
(define (write-natural file nmbr)
 (claim-bytes! file (+ (disk:natural-bytes nmbr) 1))
 (let* ((hder (header file))
```

```
(define (write-integer file nmbr)
  (claim-bytes! file (+ (disk:integer-bytes nmbr) 1))
  (let* ((
```

```
(define (write-real file nmbr)
  (claim-bytes! file disk:real64)
  (let* ((h
```

```
(define (write-string file strg)
  (claim-bytes! file 1)
  (let* ((hder (header file))
         (curr (current hder))
         (bffr (buffer file))
         (byts (string->utf8 strg)))
    (disk:encode-byte! bffr curr (bytevector-length byts))
    (current! hder (+ curr 1))
    (rollout-bytes file byts 0)))
```

1 procedure per type

Garanderen van Plaats

```
(define (claim-bytes! file nmbr)
  (define bffr (buffer file))
  (define hder (header file))
  (define curr (current hder))
  (when (< (block-bytes-free file) nmbr)
    (if (not (= curr disk:block-size))
        (disk:encode-byte! bffr curr eob-tag))
    (provide-next-block! file)))
```

```
(define (provide-next-block! file)
  (define fdsk (disk file))
  (define hder (header file))
  (define bffr (buffer file))
  (define next (cond ((fs:null-block? (fs:next-bptr bffr))
                      (let ((newb (fs:new-block fdsk)))
                        (fs:next-bptr! newb fs:null-block)
                        (fs:next-bptr! bffr (disk:position newb))
                        (disk:write-block! bffr)
                        newb))
                     (else
                      (disk:write-block! bffr)
                      (disk:read-block fdsk (fs:next-bptr bffr)))))
  (buffer!  file next)
  (current! hder disk:block-ptr-size))
```

Ofwel een nieuw blok; Ofwel een oud blok

Lezen

```
(define (read file)
  (define hder (header file))
  (define curr (current hder))
  (let* ((ttag (read-type-tag file))
         (cur
         (car
```

```
(define (peek file)
  (define hder (header file))
  (let* ((bffr (buffer file))
         (curr-offs (current hder))
         (ttag (read-type-tag file))
         (res (car (cond ((= ttag natural-tag)
                          (peek-natural file))
                         ((= ttag integer-tag)
                          (peek-integer file))
                         ((= ttag decimal-tag)
                          (peek-real file))
                         ((= ttag string-tag)
                          (peek-string file))
                         ((= ttag eof-tag)
                          (cons () curr-offs))
                         (else
                          (error "unsupported type on file (peek)" ttag))))))
    (current! hder curr-offs)
    (buffer!  file bffr)
    res))
```

Lees de waarde na het type gelezen te hebben

`peek` reset de toestand

Garanderen van Bytes

```
(define (supply-bytes! file)
  (define bffr (buffer file))
  (define hder (header file))
  (define curr (current hder))
  (if (not (more-on-buffer? file))
    (read-next-block! file)))
```

```
(define (more-on-buffer? file)
  (define bffr (buffer file))
  (define hder (header file))
  (define offs (current hder))
  (and (< offs disk:block-size)
       (not (or (= (disk:decode-byte bffr offs) eob-tag)
                (= (disk:decode-byte bffr offs) eof-tag)))))
```

```
(define (read-next-block! file)
  (define fdsk (disk file))
  (define hder (header file))
  (define bffr (buffer file))
  (define next-bptr (fs:next-bptr bffr))
  (define next-blck (disk:read-block fdsk next-bptr))
  (buffer!  file next-blck)
  (current! hder disk:block-ptr-size))
```

H9-H11: Hét Centrale Thema

Minimaliseren van het aantal bloktransferten

Caching

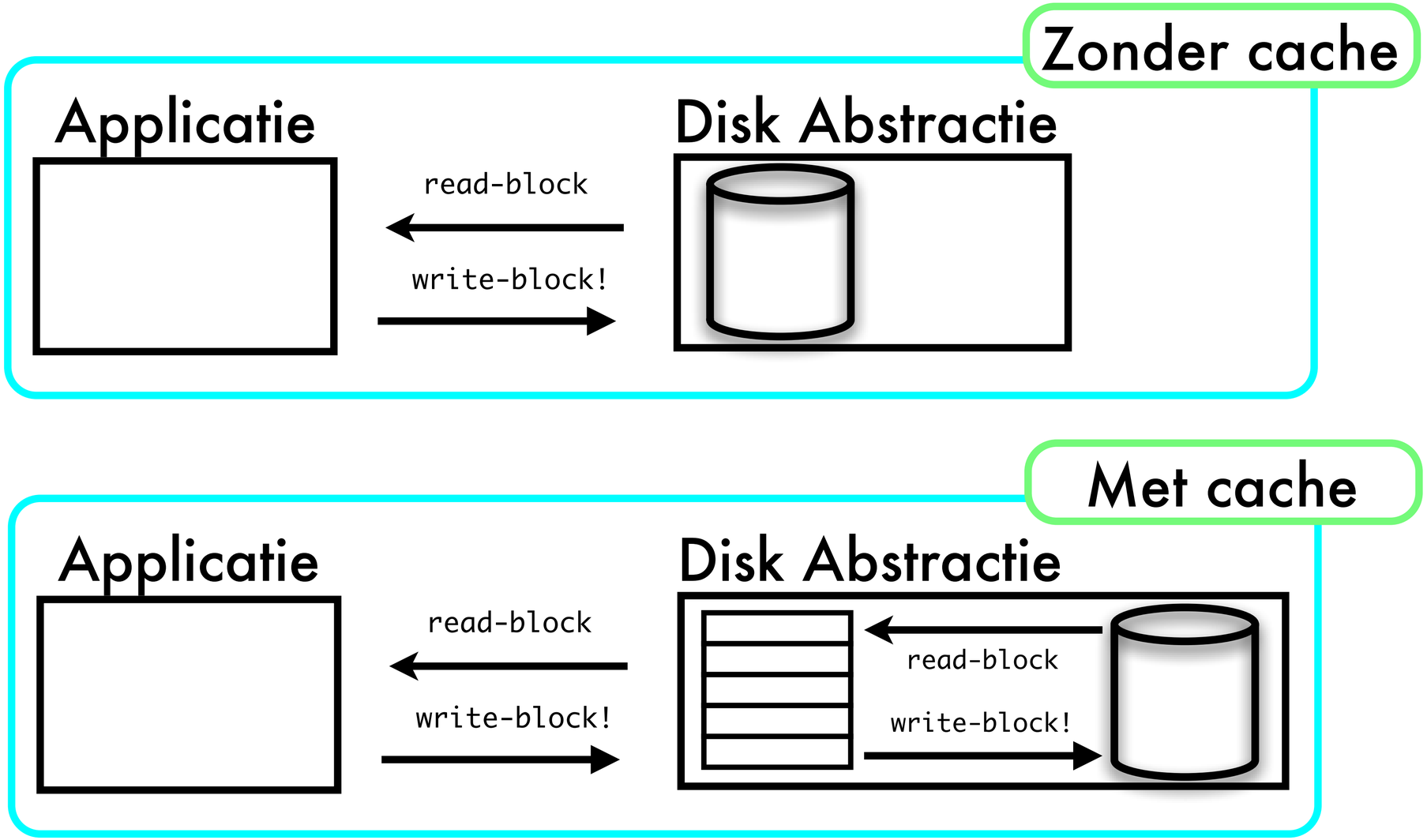
Een cache is een mechanisme dat twee verschillende geheugens doorverbindt en het verschil in performantiegedrag van die twee geheugens opvangt via statistische technieken.

Een stuk gereserveerd geheugen in het snelle geheugen dat een kopie van een deel van het trage geheugen bevat.

Enkele bytevectoren in de processor

Enkele blokken in centraal geheugen

Disk Caches


Zonder cache
Applicatie
Disk Abstractie
read-block
write-block!
Met cache
Applicatie
Disk Abstractie
read-block
write-block!
read-block
write-block!

Basis voor Caching: Locality

Temporal Locality: De meest recent gebruikte geheugenlocaties hebben de grootste kans om in de nabije toekomst opnieuw gebruikt te worden.

Spatial Locality: Geheugenlocaties dichtbij de meest recent gebruikte geheugenlocaties hebben de grootste kans om in de nabije toekomst gebruikt te worden.

Selection Sort

Quicksort

Caching Terminologie

Als een benodigde datawaarde tijdens het lezen van de cache gevonden wordt, spreken we van een cache hit. Anders van een cache miss en is een evict nodig alvorens een nieuwe datawaarde in de cache kan geladen worden.

Als een datawaarde in de cache geschreven wordt moet het trage geheugen ook ooit geupdate worden. In een write-through cache gebeurt dat meteen. In een write-back cache gebeurt dat bij een evict. Een dirty-bit is dan nodig voor boekhouding.

Welke Datawaarden Evicten?

Random: Gooi een willekeurige datawaarde uit de cache.

O(1)

First-In, First-Out (FIFO): De oudste datawaarde wordt eruit gegooid.

Least Frequently Used (LFU): Gooi de minst gebruikte datawaarde eruit.

O(size)

Least Recently Used (LRU): Gooi de datawaarde eruit die het langst niet meer gebruikt is.

Gecachete Disk Blokken

`write-block!` maakt het blok ongelocked

`read-block` maakt het blok gelocked

Elk gebruikt (encode of decode) actualiseert de time-stamp

Encoders maken het blok “dirty”

```
dirty?
  ( block → boolean )
dirty!
  ( block boolean→ ∅ )
locked?
  ( block → boolean )
locked!
  ( block boolean→ ∅ )
time-stamp
  ( block → time )
time-stamp!
  ( block time → ∅ )
invalidate!
  ( block → ∅ )
valid?
  ( block → boolean )
```

Eens ge-evict uit de cache, is het blok niet meer valid

Gecachete Blokken

```
(define-record-type cblock
  (make d l t i b)
  block?
  (d dirty? dirty-set!)
  (l locked? locked!)
  (t time-stamp time-stamp!)
  (i disk)
  (b block block!))

(define (make-block cdsk blck)
  (make #f #t (current-time) cdsk blck))
```

dirty: werd het blok door encoding aangeraakt?

locked: werd het block door de applicatie nog niet logisch weggeschreven?

```
(define (dirty! blck)
  (dirty-set! blck #t))

(define (invalidate! blck)
  (block! blck ()))

(define (valid? blck)
  (not (null? (block blck))))
```

Gecachete Encoders

```
(define (make-cached-encoder proc)
  (lambda args
    (define blck (car args))
    (if (not (valid? blck))
        (error "invalidated cblock(cached version of encoder)" blck))
    (time-stamp! blck (current-time))
    (dirty! blck)
    (apply proc (cons (block blck)  (cdr args)))))

(define encode-byte!
   (make-cached-encoder disk:encode-byte!))
(define encode-fixed-natural!
   (make-cached-encoder disk:encode-fixed-natural!))
(define encode-arbitrary-integer!
   (make-cached-encoder disk:encode-arbitrary-integer!))
(define encode-real!
   (make-cached-encoder disk:encode-real!))
(define encode-string!
   (make-cache
(define encode
   (make-cache
```

De dirty flag wordt gezet

We maken het blok “jonger”

Oorspronkelijke operatie wordt opgeroepen

Operaties op invalid blokken worden tegengehouden

Gecachete Decoders

```
(define (make-cached-decoder proc)
  (lambda args
    (define blck (car args))
    (if (not (valid? blck))
        (error "invalidated cblock(cached version of decoder)" blck))
    (time-stamp! blck (current-time))
    (apply proc (cons (block blck)  (cdr args)))))

(define decode-byte
   (make-cached-decoder disk:decode-byte))
(define decode-fixed-natural
   (make-cached-decoder disk:decode-fixed-natural))
(define decode-arbitrary-integer
   (make-cached-decoder disk:decode-arbitrary-integer))
(define decode-real
   (make-cached-decoder disk:decode-real))
(define decode-string
   (make-cached-decoder disk:decode-string))
(define decode-bytes
   (make-cached-decoder disk:decode-bytes))
```

Zelfde idee

Gecachete Disk

```
ADT disk

block-size
  number
disk-size
  number
block-ptr-size
  number
block-idx-size
  number
new
  ( string → disk )
mount
  ( string → disk )
unmount
  ( disk → ∅ )
disk?
  ( any → boolean )
name
  ( file → string )
size
  ( file → number )
read-block
  ( disk number → block )
```

ADT blijft onveranderd

Implementatie bevat echter een cache in het centraal geheugen

De Eigenlijke Cache

```
(define cache-size 10)

(define (cache:new)
  (make-vector cache-size ()))

(define (cache:get cche indx)
  (vector-ref cche indx))

(define (cache:put! cche indx blck)
  (vector-set! cche indx blck))
```

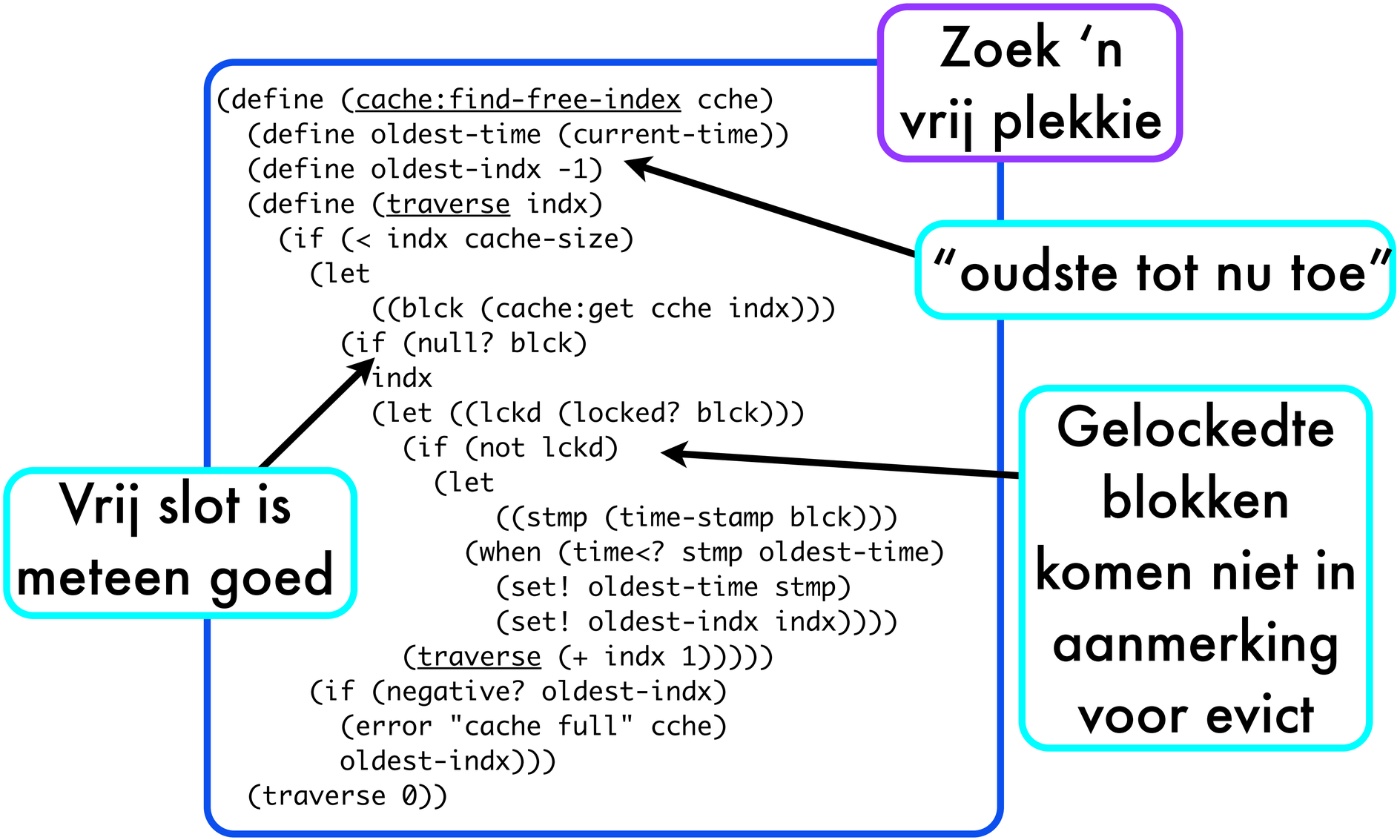
Gewoon een vector van blokken

Zoek een gegeven blok nummer op in de cache

```
(define (cache:find-block cche bptr)
  (define (position-matches? blck)
    (and (not (null? blck))
         (= (position blck) bptr)))
  (let traverse
    ((indx 0)
     (blck (cache:get cche 0)))
    (cond ((position-matches? blck)
           blck)
          ((< (+ indx 1) cache-size)
           (traverse (+ indx 1)
                     (cache:get cche (+ indx 1))))
          (else
           ()))))
```

Deze cache is fully associative: ieder blok kan gelijk waar terecht komen in de cache.

De eigenlijke Cache (ctd)



Creëren, Mounten & Unmounten

```
(define-record-type cdisk
  (make-cdisk v d)
  disk?
  (v disk-cache)
  (d real-disk))
```

Gecachete disk
= disk + cache

```
(define (new name)
  (make-cdisk (cache:new) (disk:new name)))

(define (mount name)
  (make-cdisk (cache:new) (disk:mount name)))

(define (name cdsk)
  (disk:name (real-disk cdsk)))
```

Alle dirty blokken moeten geschreven worden

Wandel de volledige cache af

```
(define (unmount disk)
  (define vctr (cache:vector disk))
  (define (traverse indx)
    (if (< indx cache-size)
        (let ((blck (cache:get vctr indx)))
          (cond
            ((not (null? blck))
             (if (dirty? blck)
                 (disk:write-block! (block blck)))
             (invalidate! blck)))
          (traverse (+ indx 1)))))
  (traverse 0)
  (disk:unmount disk))
```

Lezen en Schrijven van Blokken

```
(define (read-block cdsk bptr)
  (define cche (disk-cache cdsk))
  (define blck (cache:find-block cche bptr))
  (if (null? blck)
    (let*
        ((indx (cache:find-free-index cche))
         (blck (cache:get cche indx)))
      (when (not (null? blck))
        (if (dirty? blck)
          (disk:write-block! (block blck)))
        (invalidate! blck))
      (set! blck (make-block cdsk (disk:read-block (real-disk cdsk) bptr)))
      (cache:put! cche indx blck)))
  (locked! blck #t)
  blck)
```

Indien niet in cache, lees van de disk, eventueel na een evict

Gelezen blokken worden gelocket

```
(define (write-block! blck)
  (if (not (valid? blck))
    (error "invalidated block(write-block!)" blck))
  (locked! blck #f))
```

en worden door schrijven weer unlocked

write schrijft het block niet meteen weg!

Hoofdstuk 14

14.1 Wat is Data? Bits, bytes, bytevectoren

14.2 De disk abstractie

14.3 Het filesysteem

14.4 Sequentiële Files

14.5 Caching