Linux, Locking and Lots of Processors

Peter Chubb

`peter.chubb@nicta.com.au`

A LITTLE BIT OF HISTORY

NICTA

- Multix in the '60s

A LITTLE BIT OF HISTORY

- Multix in the '60s
- Ken Thompson and Dennis Ritchie in 1967–70

A LITTLE BIT OF HISTORY

- Multix in the ’60s
- Ken Thompson and Dennis Ritchie in 1967–70
- USG and BSD

A LITTLE BIT OF HISTORY

- Multix in the '60s
- Ken Thompson and Dennis Ritchie in 1967–70
- USG and BSD
- John Lions 1976–95

A LITTLE BIT OF HISTORY

NICTA

- Multix in the '60s
- Ken Thompson and Dennis Ritchie in 1967–70
- USG and BSD
- John Lions 1976–95
- Andrew Tanenbaum 1987

A LITTLE BIT OF HISTORY

- Multix in the ’60s
- Ken Thompson and Dennis Ritchie in 1967–70
- USG and BSD
- John Lions 1976–95
- Andrew Tanenbaum 1987
- Linux Torvalds 1991

A LITTLE BIT OF HISTORY

- Basic concepts well established
  - Process model
  - File system model
  - IPC

A LITTLE BIT OF HISTORY

- Basic concepts well established
  - Process model
  - File system model
  - IPC
- Additions:
  - Paged virtual memory (3BSD, 1979)

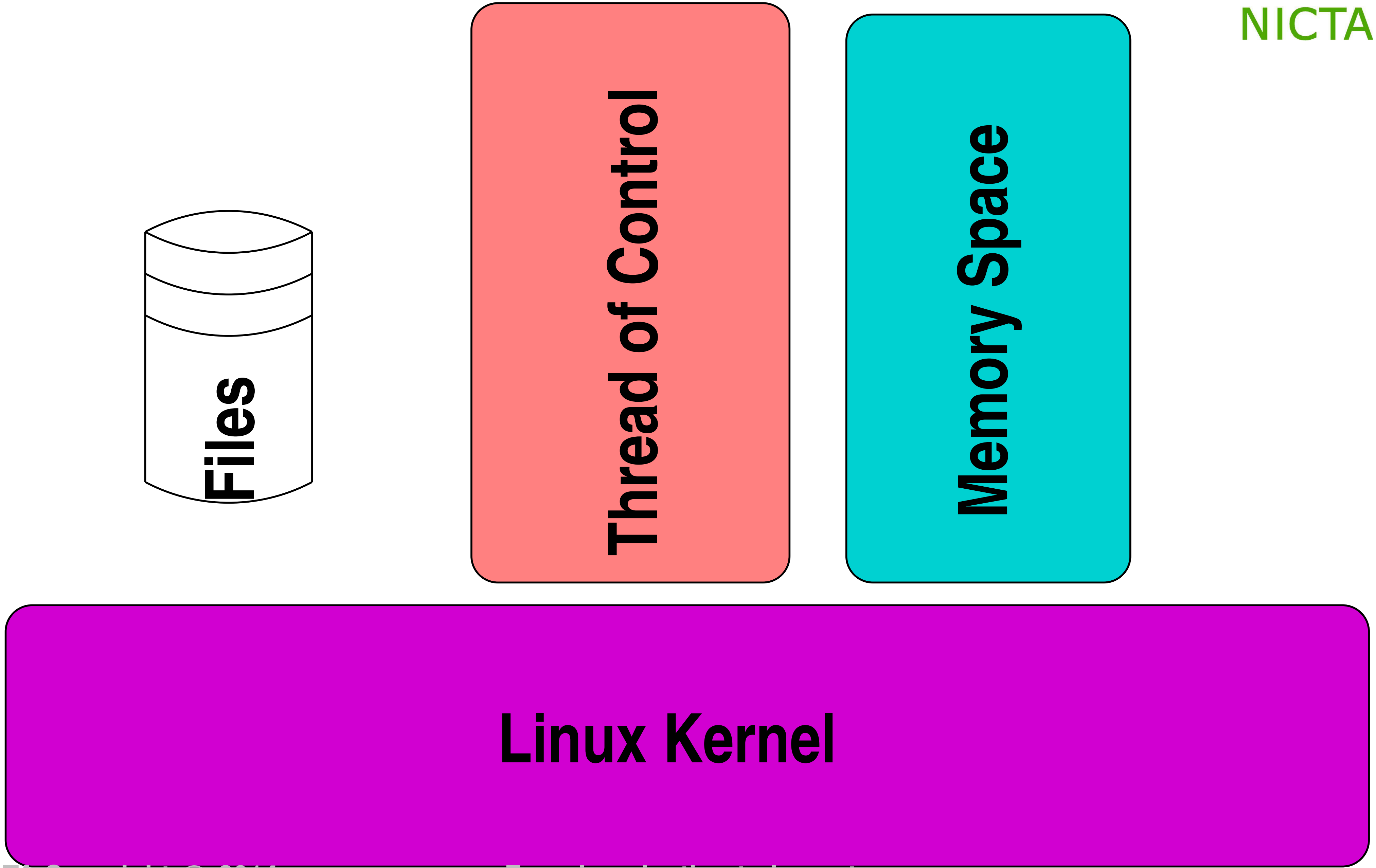
A LITTLE BIT OF HISTORY

- Basic concepts well established
  - Process model
  - File system model
  - IPC
- Additions:
  - Paged virtual memory (3BSD, 1979)
  - TCP/IP Networking (BSD 4.1, 1983)

A LITTLE BIT OF HISTORY

- Basic concepts well established
  - Process model
  - File system model
  - IPC
- Additions:
  - Paged virtual memory (3BSD, 1979)
  - TCP/IP Networking (BSD 4.1, 1983)
  - Multiprocessing (Vendor Unices such as Sequent's 'Balance', 1984)

ABSTRACTIONS



PROCESS MODEL

- Root process (`init`)
- `fork()` creates (almost) exact copy
  - Much is shared with parent — Copy-On-Write avoids overmuch copying
- `exec()` overwrites memory image from a file

PROCESS MODEL

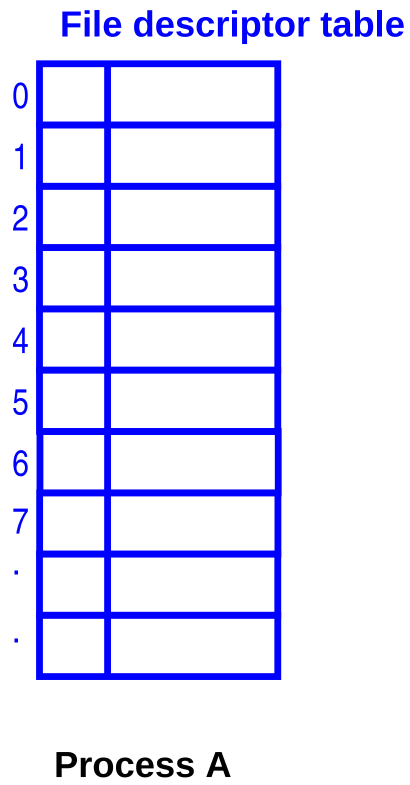
- Root process (`init`)
- `fork()` creates (almost) exact copy
  - Much is shared with parent — Copy-On-Write avoids overmuch copying
- `exec()` overwrites memory image from a file
- Allows a process to control what is shared

FORK() AND EXEC()

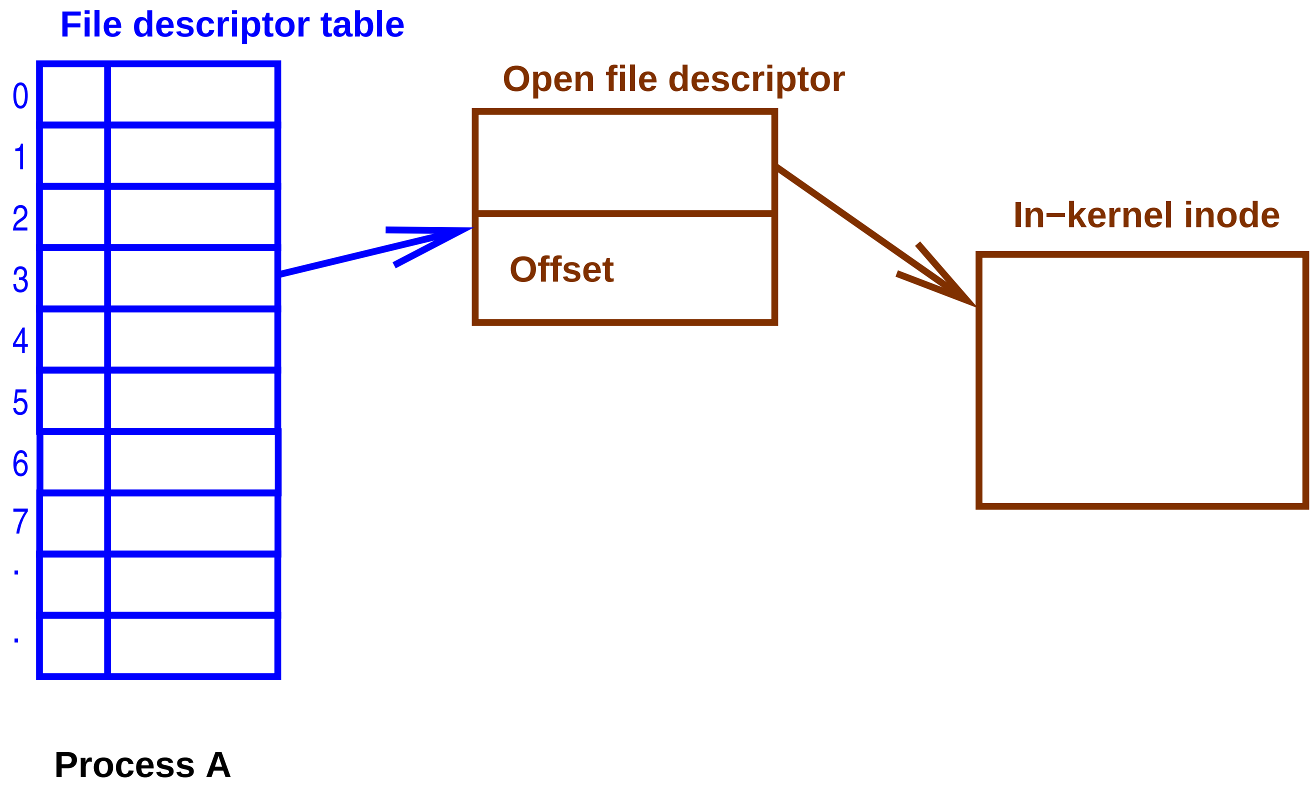
NICTA

- A process can clone itself by calling **`fork()`**.
- Most attributes *copied*:
  - Address space (actually shared, marked copy-on-write)
  - current directory, current root
  - File descriptors
  - permissions, etc.
- Some attributes *shared*:
  - Memory segments marked **`MAP_SHARED`**
  - Open files

Files and Processes:

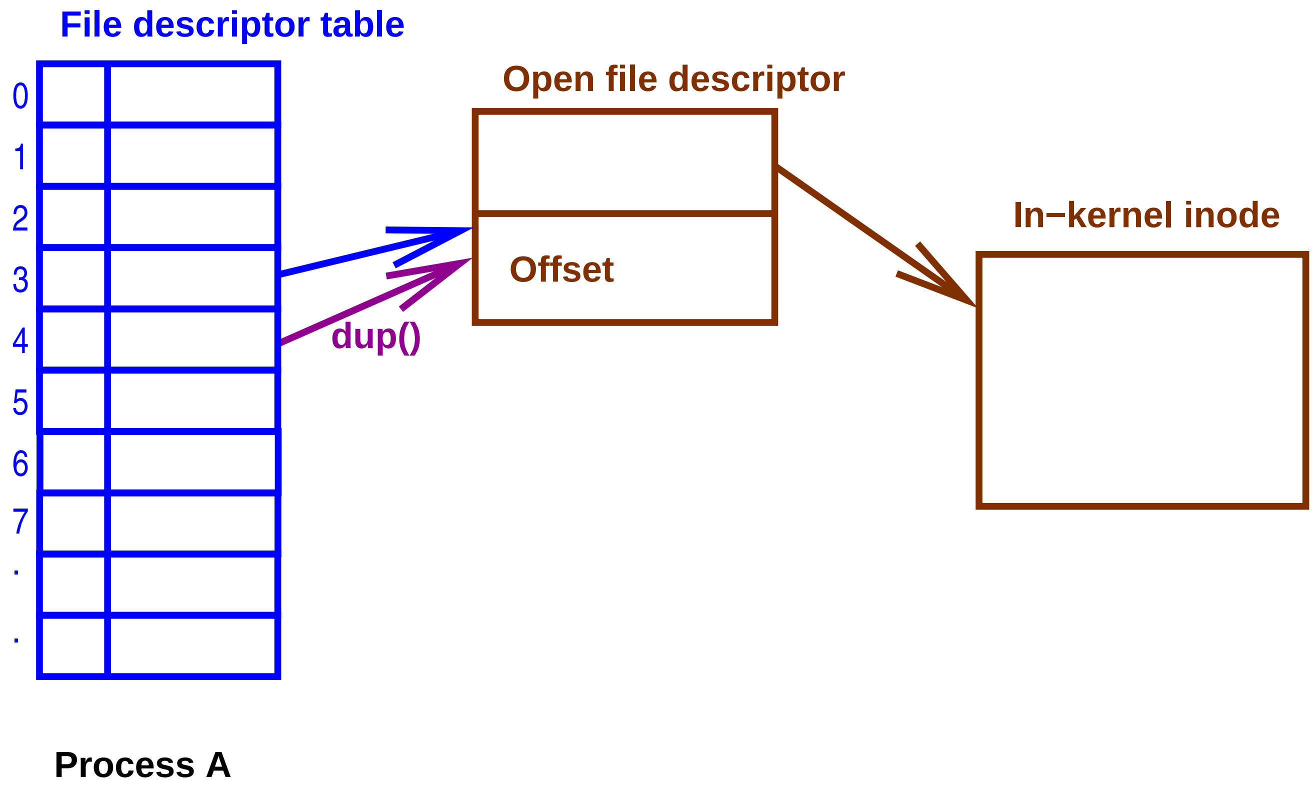


Files and Processes:

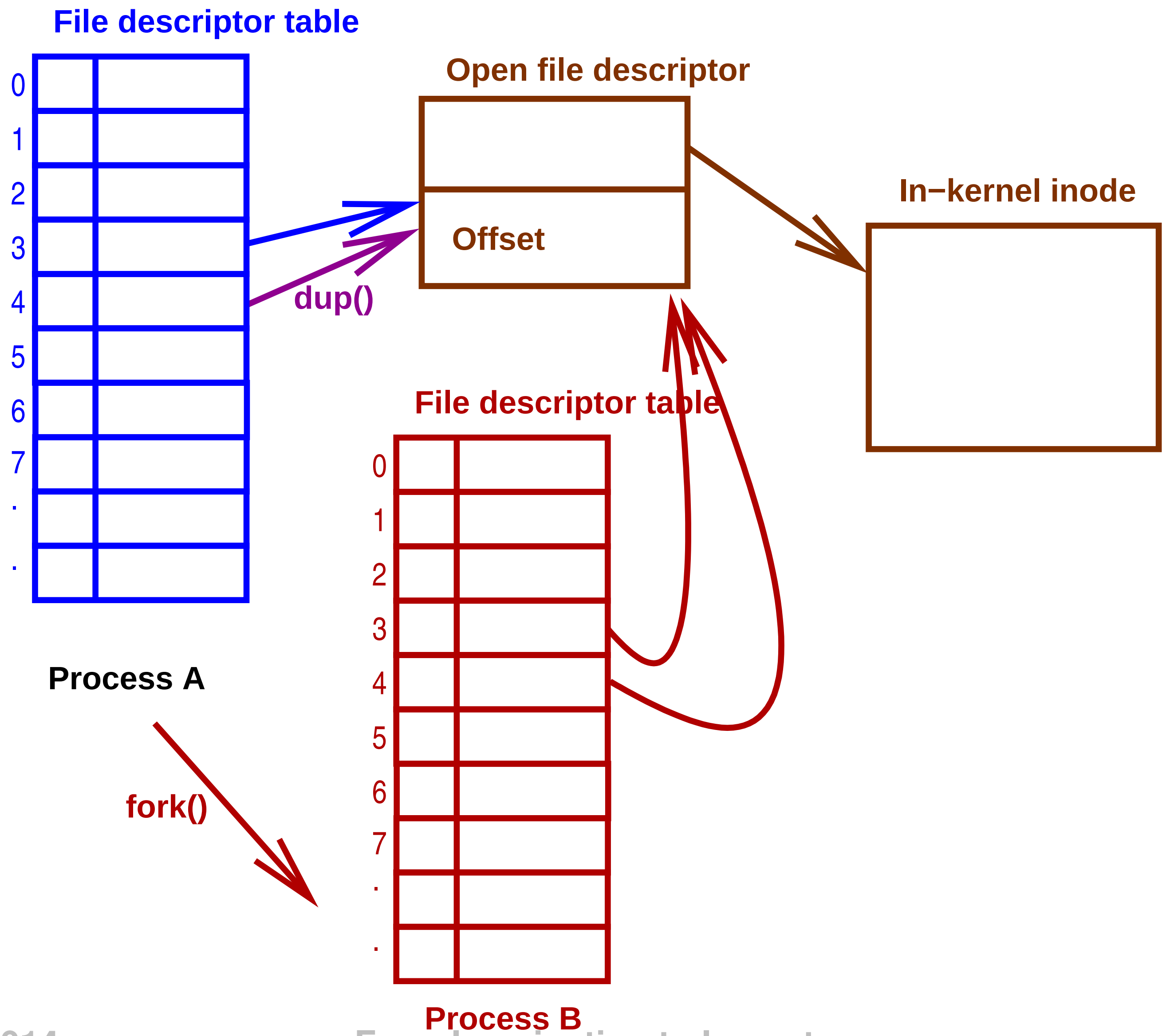


NICTA

Files and Processes:



Files and Processes:



fork() and exec()

```
switch (kidpid = fork()) {
case 0: /* child */
    close(0); close(1); close(2);
    dup(infd); dup(outfd); dup(outfd);
    execve("path/to/prog", argv, envp);
    _exit(EXIT_FAILURE);
case -1:
      /* handle error */
default:
    waitpid(kidpid, &status, 0);
}
```

STANDARD FILE DESCRIPTORS

0 Standard Input

1 Standard Output

2 Standard Error

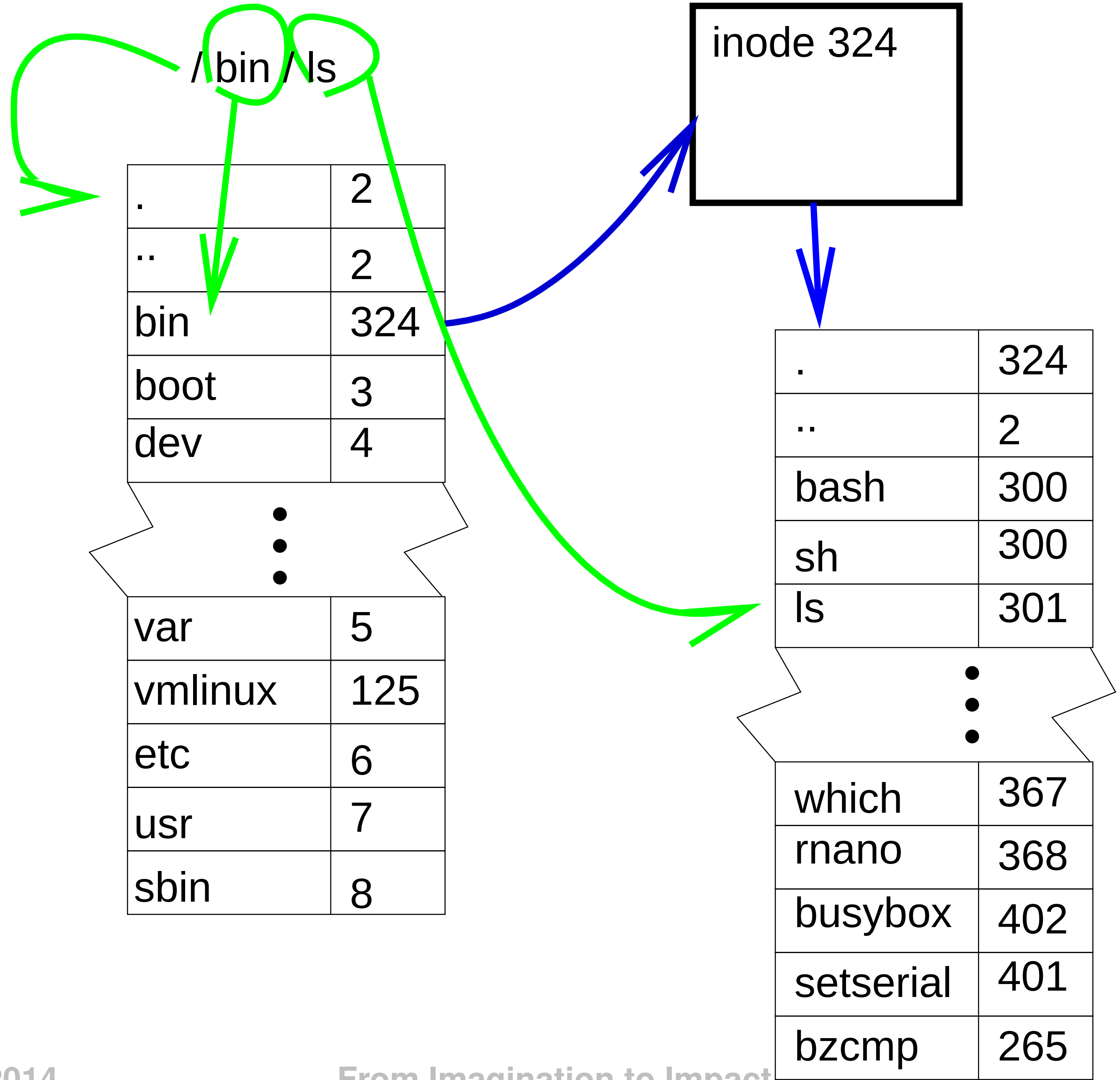
➜ Inherited from parent

➜ On login, all are set to *controlling tty*

File model

- Separation of names from content.
- 'regular' files 'just bytes' $\rightarrow$ structure/meaning supplied by userspace
- Devices represented by files.
- Directories map names to index node indices (`inums`)
- Simple permissions model

FILE MODEL



/ bin / ls
inode 324
. 2
.. 2
bin 324
boot 3
dev 4
var 5
vmlinux 125
etc 6
usr 7
sbin 8
. 324
.. 2
bash 300
sh 300
ls 301
which 367
rnano 368
busybox 402
setserial 401
bzcmp 265

NAMEI

- ➜ translate name $\rightarrow$ inode
- ➜ abstracted per filesystem in VFS layer
- ➜ Can be slow: extensive use of caches to speed it up
  dentry cache
- ➜ hide filesystem and device boundaries
- ➜ walks pathname, translating symbolic links

NAMEI

- translate name $\rightarrow$ inode
- abstracted per filesystem in VFS layer
- Can be slow: extensive use of caches to speed it up
  dentry cache — becomes SMP bottleneck
- hide filesystem and device boundaries
- walks pathname, translating symbolic links

EVOLUTION

KISS:

➜ Simplest possible algorithm used at first

EVOLUTION

KISS:

- ➜ Simplest possible algorithm used at first
  - ➜ Easy to show correctness
  - ➜ Fast to implement

Evolution

KISS:

- ➜ Simplest possible algorithm used at first
  - ➜ Easy to show correctness
  - ➜ Fast to implement
- ➜ As drawbacks and bottlenecks are found, replace with faster/more scalable alternatives

C DIALECT

- Extra keywords:
  - Section IDs: `__init`, `__exit`, `__percpu` etc
  - Info Taint annotation `__user`, `__rcu`, `__kernel`, `__iomem`
  - Locking annotations `__acquires(X)`, `__releases(x)`
  - extra typechecking (endian portability) `__bitwise`

C DIALECT

- Extra iterators
  - *type_name*`_foreach()`
- Extra accessors
  - `container_of()`

C Dialect

- Massive use of inline functions
- Some use of CPP macros
- Little `#ifdef` use in code: rely on optimizer to elide dead code.

SCHEDULING

Goals:

- O(1) in number of runnable processes, number of processors
  - good uniprocessor performance
- 'fair'
- Good interactive response
- topology-aware

SCHEDULING

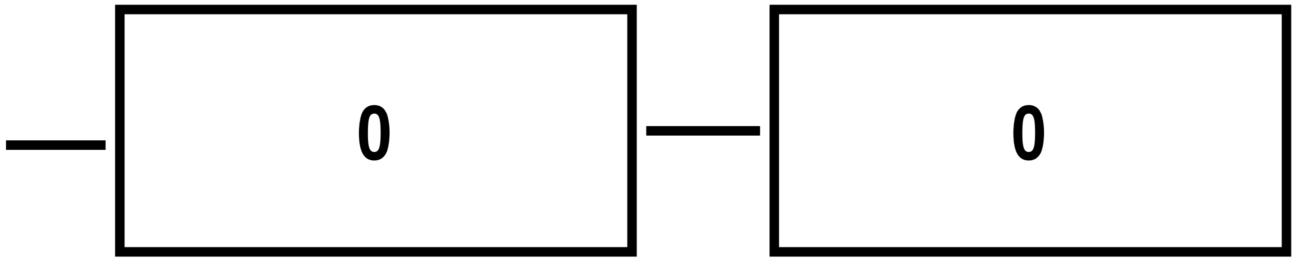
Implementation:

- Changes from time to time.
- Currently 'CFS' by Ingo Molnar.

Dual Entitlement Scheduler

Running

Expired



Scheduling

1. Keep tasks ordered by effective CPU runtime weighted by nice in red-black tree
2. Always run left-most task.

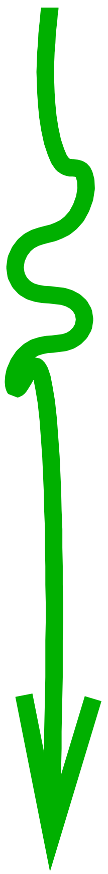
SCHEDULING

1. Keep tasks ordered by effective CPU runtime weighted by nice in red-black tree
2. Always run left-most task.

Devil's in the details:

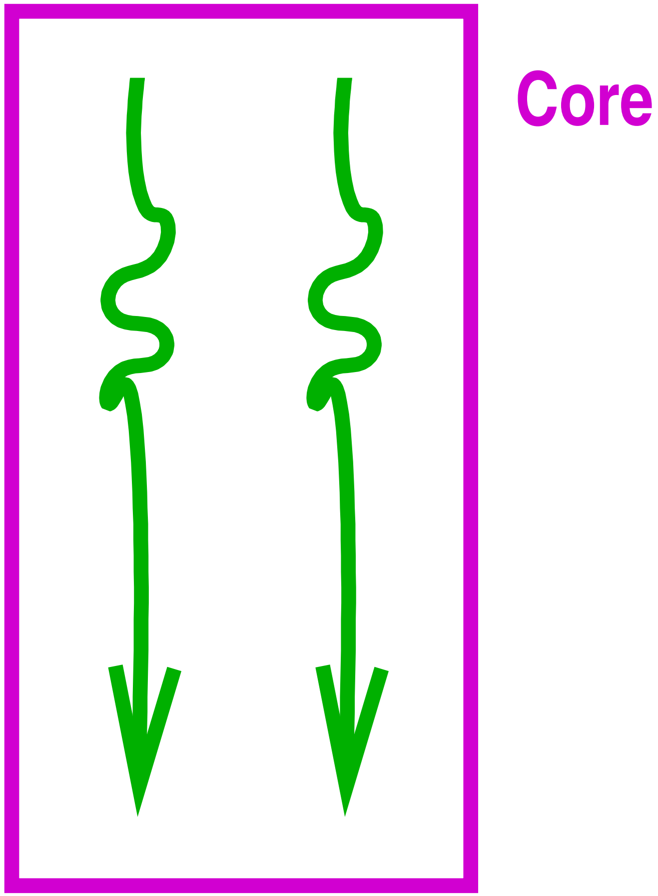
- Avoiding overflow
- Keeping recent history
- multiprocessor locality
- handling too-many threads
- Sleeping tasks
- Group hierarchy

SCHEDULING

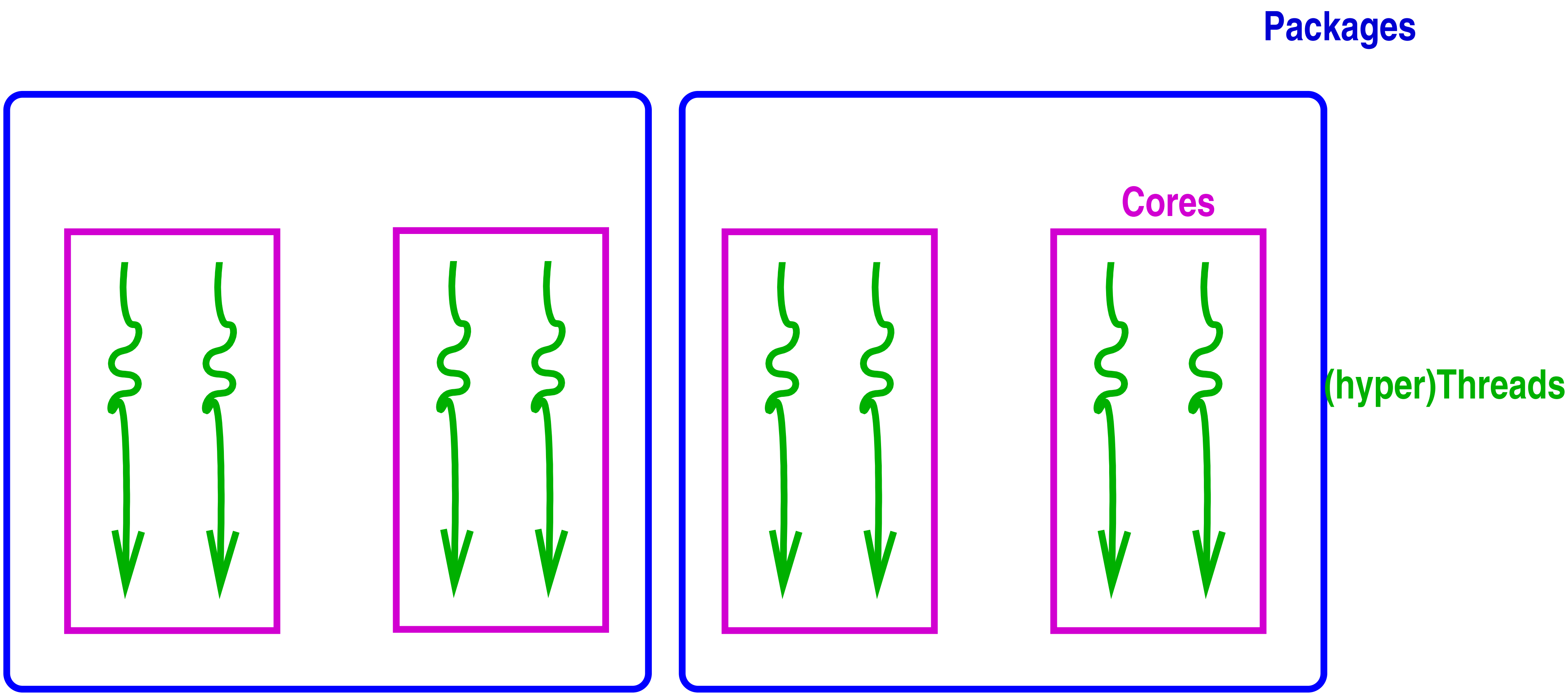


(hyper)Thread

SCHEDULING

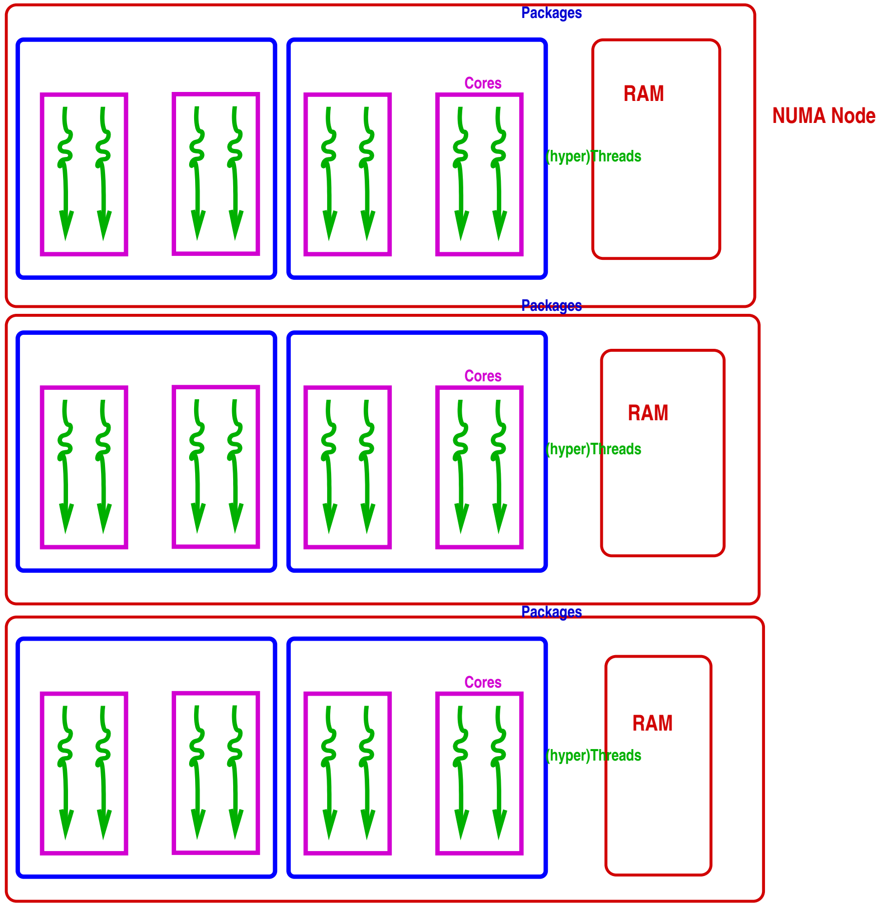


Scheduling



SCHEDULING

NICTA



NICTA

Locality Issues:

- Best to reschedule on same processor (don't move cache footprint, keep memory close)

NICTA

Locality Issues:

- Best to reschedule on same processor (don’t move cache footprint, keep memory close)
  - Otherwise schedule on a ‘nearby’ processor

SCHEDULING

Locality Issues:

- Best to reschedule on same processor (don’t move cache footprint, keep memory close)
  - Otherwise schedule on a ‘nearby’ processor
- Try to keep whole sockets idle

NICTA

Locality Issues:

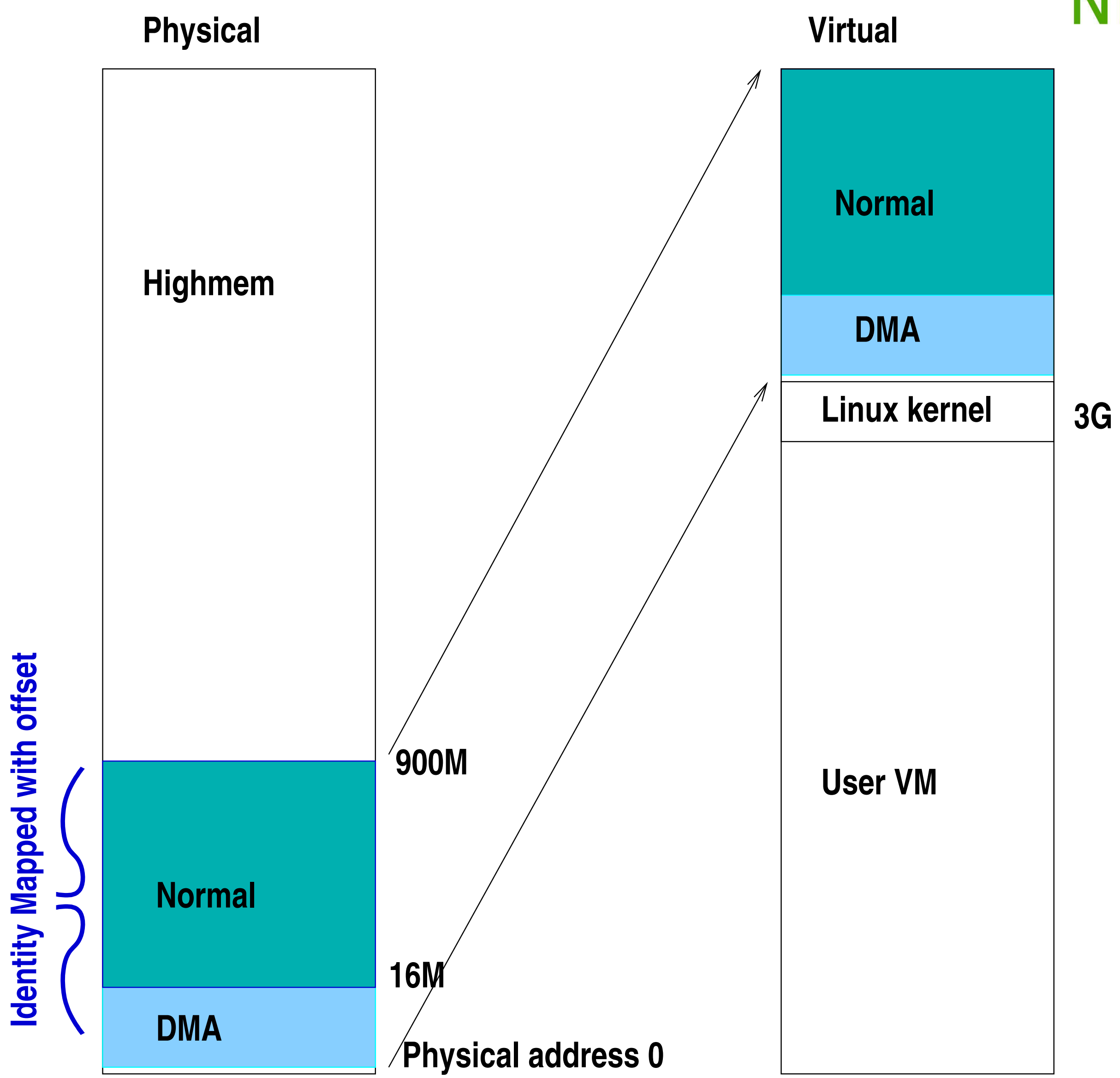
- Best to reschedule on same processor (don’t move cache footprint, keep memory close)
  - Otherwise schedule on a ‘nearby’ processor
- Try to keep whole sockets idle
- Somehow identify cooperating threads, co-schedule on same package?

SCHEDULING

- One queue per processor (or hyperthread)
- Processors in hierarchical 'domains'
- Load balancing per-domain, bottom up
- Aims to keep whole domains idle if possible (power savings)

Memory Management

Memory in *zones*



Memory Management

- Direct mapped pages become *logical addresses*
  - `__pa()` and `__va()` convert physical to virtual for these

Memory Management

- Direct mapped pages become *logical addresses*
  - `__pa()` and `__va()` convert physical to virtual for these
- small memory systems have all memory as logical

Memory Management

- Direct mapped pages become *logical addresses*
  - `__pa()` and `__va()` convert physical to virtual for these
- small memory systems have all memory as logical
- More memory $\rightarrow \Delta$ kernel refer to memory by `struct page`

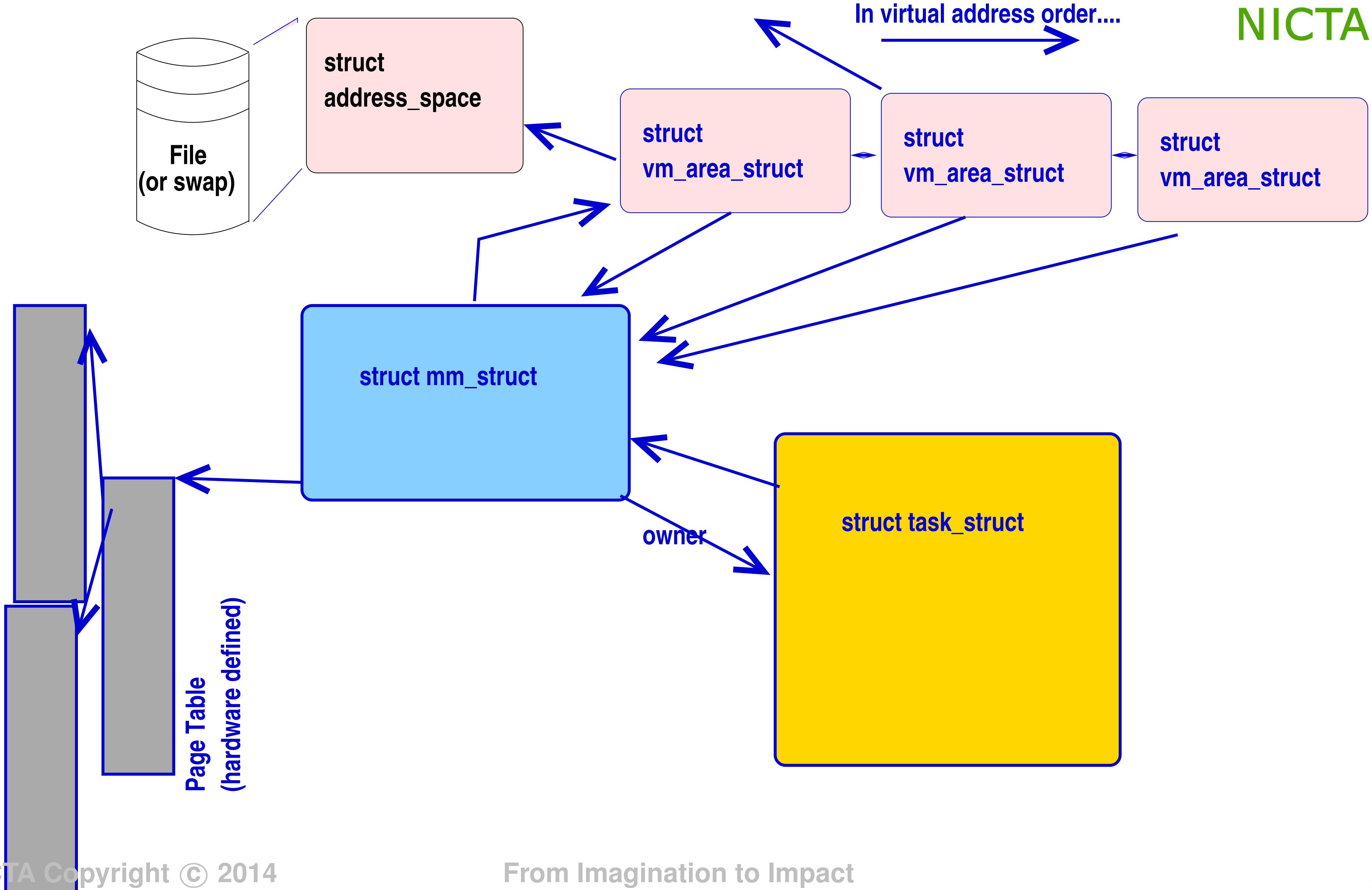
MEMORY MANAGEMENT

NICTA

`struct page`:

- Every frame has a `struct page` (up to 10 words)
- Track:
  - flags
  - backing address space
  - offset within mapping *or* freelist pointer
  - Reference counts
  - Kernel virtual address (if mapped)

MEMORY MANAGEMENT



Address Space:

- Misnamed: means collection of pages mapped from the same object
- Tracks inode mapped from, radix tree of pages in mapping
- Has ops (from file system or swap manager) to:

  dirty mark a page as dirty

  readpages populate frames from backing store

  writepages Clean pages — make backing store the same as in-memory copy

MEMORY MANAGEMENT

migratepage Move pages between NUMA nodes

Others. . . And other housekeeping

PAGE FAULT TIME

- Special case in-kernel faults
- Find the VMA for the address
  - segfault if not found (unmapped area)
- If it's a stack, extend it.
- Otherwise:
  1. Check permissions, SIG_SEGV if bad
  2. Call **`handle_mm_fault()`**:
     - walk page table to find entry (populate higher levels if nec. until leaf found)
     - call **`handle_pte_fault()`**

PAGE FAULT TIME

`handle_pte_fault()`: Depending on PTE status, can

- provide an anonymous page
- do copy-on-write processing
- reinstantiate PTE from page cache
- initiate a read from backing store.

and if necessary flushes the TLB.

DRIVER INTERFACE

Three kinds of device:

1. Platform device
2. enumerable-bus device
3. Non-enumerable-bus device

Driver Interface

Enumerable buses:

```
static DEFINE_PCI_DEVICE_TABLE(cp_pci_tbl) = {
{ PCI_DEVICE(PCI_VENDOR_ID_REALTEK,PCI_DEVICE_ID_
{ PCI_DEVICE(PCI_VENDOR_ID_TTTECH,PCI_DEVICE_ID_T
{ },
};
MODULE_DEVICE_TABLE(pci, cp_pci_tbl);
```

DRIVER INTERFACE

Driver interface:

`init` called to register driver

`exit` called to deregister driver, at module unload time

`probe()` called when bus-id matches; returns 0 if driver claims device

open, close, etc as necessary for driver class

Driver Interface

Platform Devices:

```
static struct platform_device nslu2_uart = {
.name = "serial8250",
.id = PLAT8250_DEV_PLATFORM,
.dev.platform_data = nslu2_uart_data,
.num_resources = 2,
.resource = nslu2_uart_resources,
};
```

DRIVER INTERFACE

non-enumerable buses: Treat like platform devices

SUMMARY

NICTA

- I've told you status today

SUMMARY

- I’ve told you status today
  - Next week it may be different

SUMMARY

- I've told you status today
  - Next week it may be different
- I've simplified a lot. There are many hairy details

FILE SYSTEMS

NICTA

I'm assuming:

- You've already looked at ext[234]-like filesystems
- You've some awareness of issues around on-disk locality and I/O performance
- You understand issues around avoiding on-disk corruption by carefully ordering events, and/or by the use of a Journal.

NORMAL FILE SYSTEMS

- Optimised for use on spinning disk
- RAID optimised (especially XFS)
- Journals, snapshots, transactions...

Flash Memory

- NOR Flash
- NAND Flash

Flash Memory

- NOR Flash
- NAND Flash
  - MTD
  - eMMC, SDHC etc
  - SSD, USB

Flash Memory

- NOR Flash
- NAND Flash
  - MTD — Memory Technology Device
  - eMMC, SDHC etc
  - SSD, USB

Flash Memory

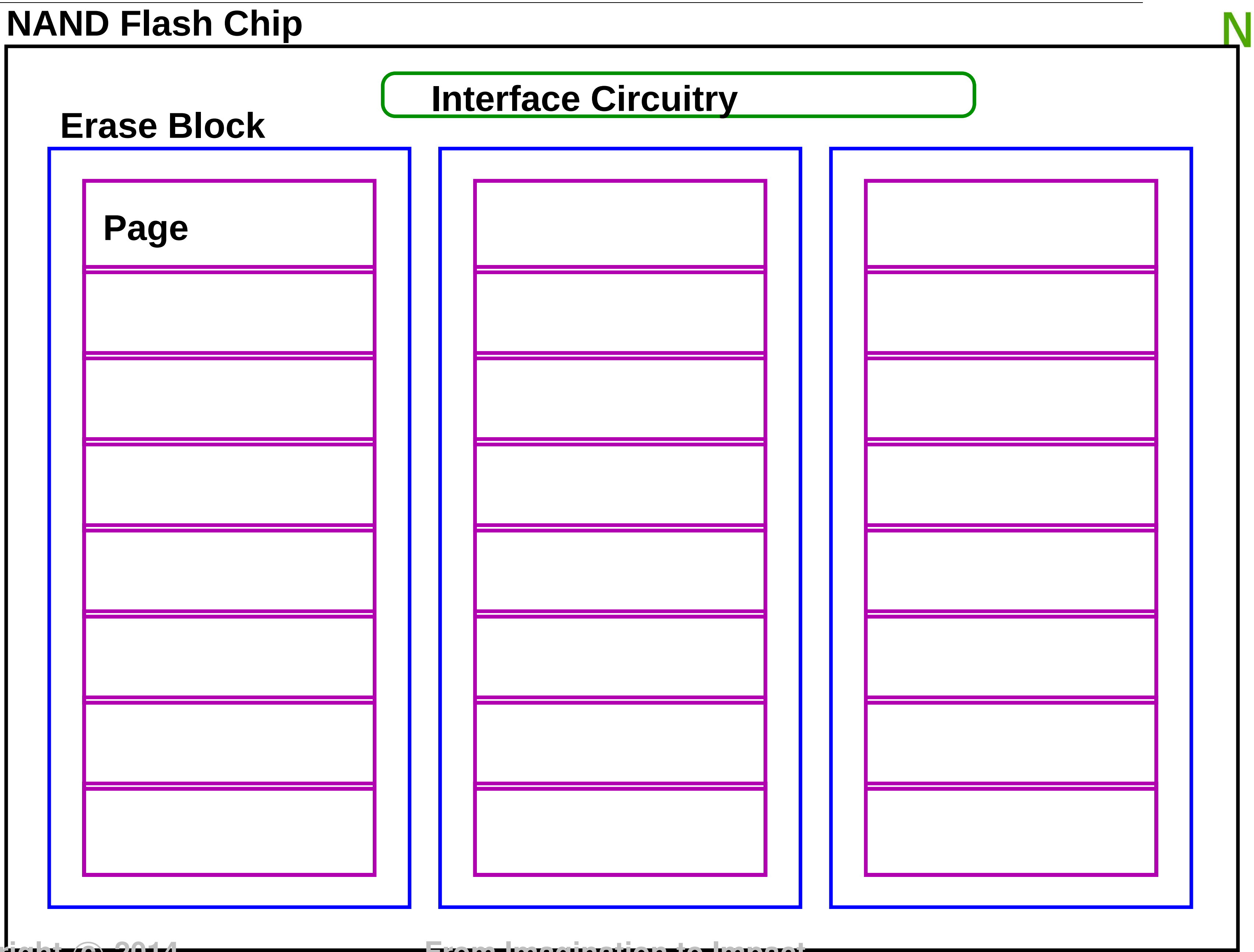
- NOR Flash
- NAND Flash
  - MTD — Memory Technology Device
  - eMMC, SDHC etc — A JEDEC standard
  - SSD, USB

FLASH MEMORY

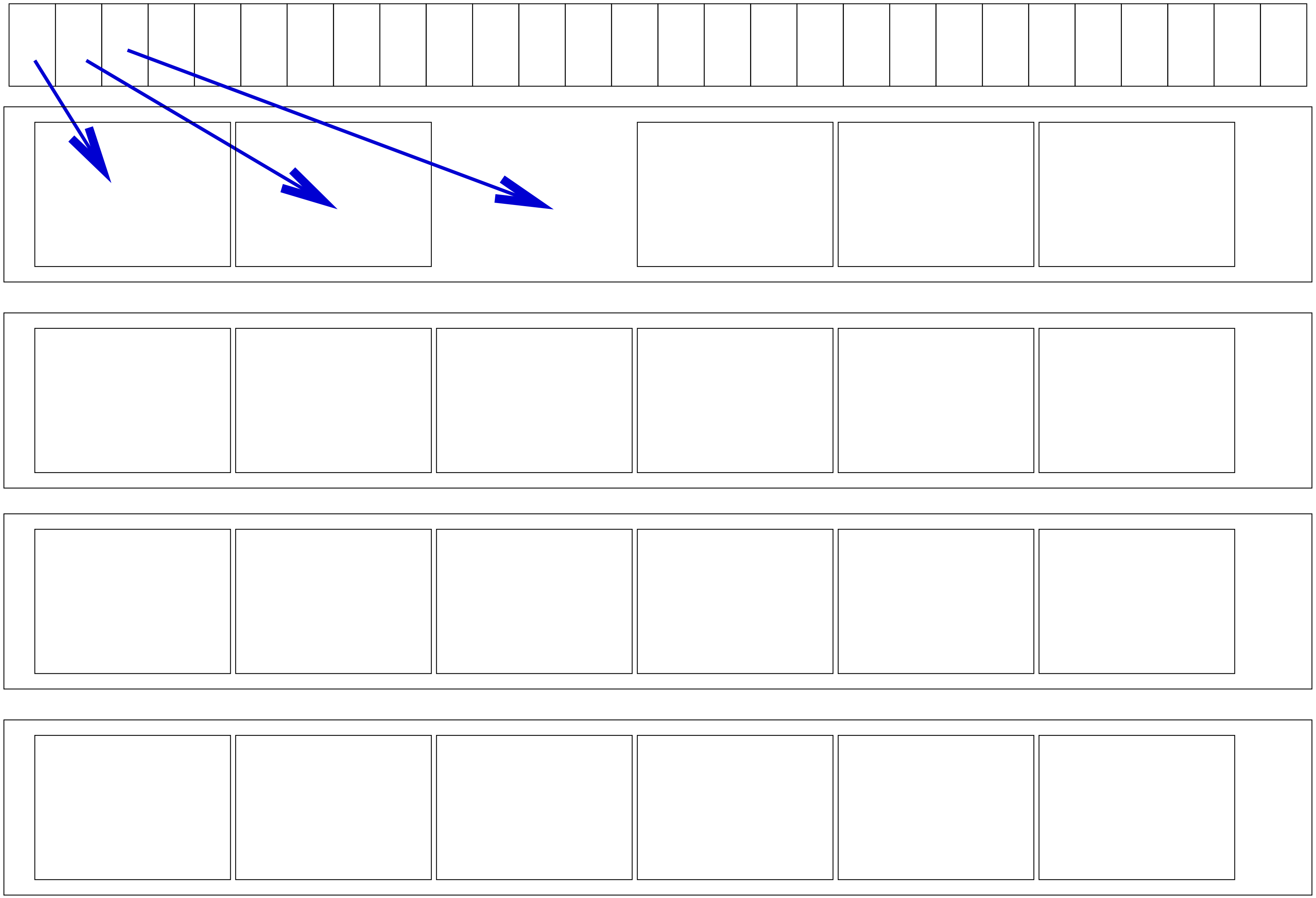
NICTA

- NOR Flash
- NAND Flash
  - MTD — Memory Technology Device
  - eMMC, SDHC etc — A JEDEC standard
  - SSD, USB — and other disk-like interfaces

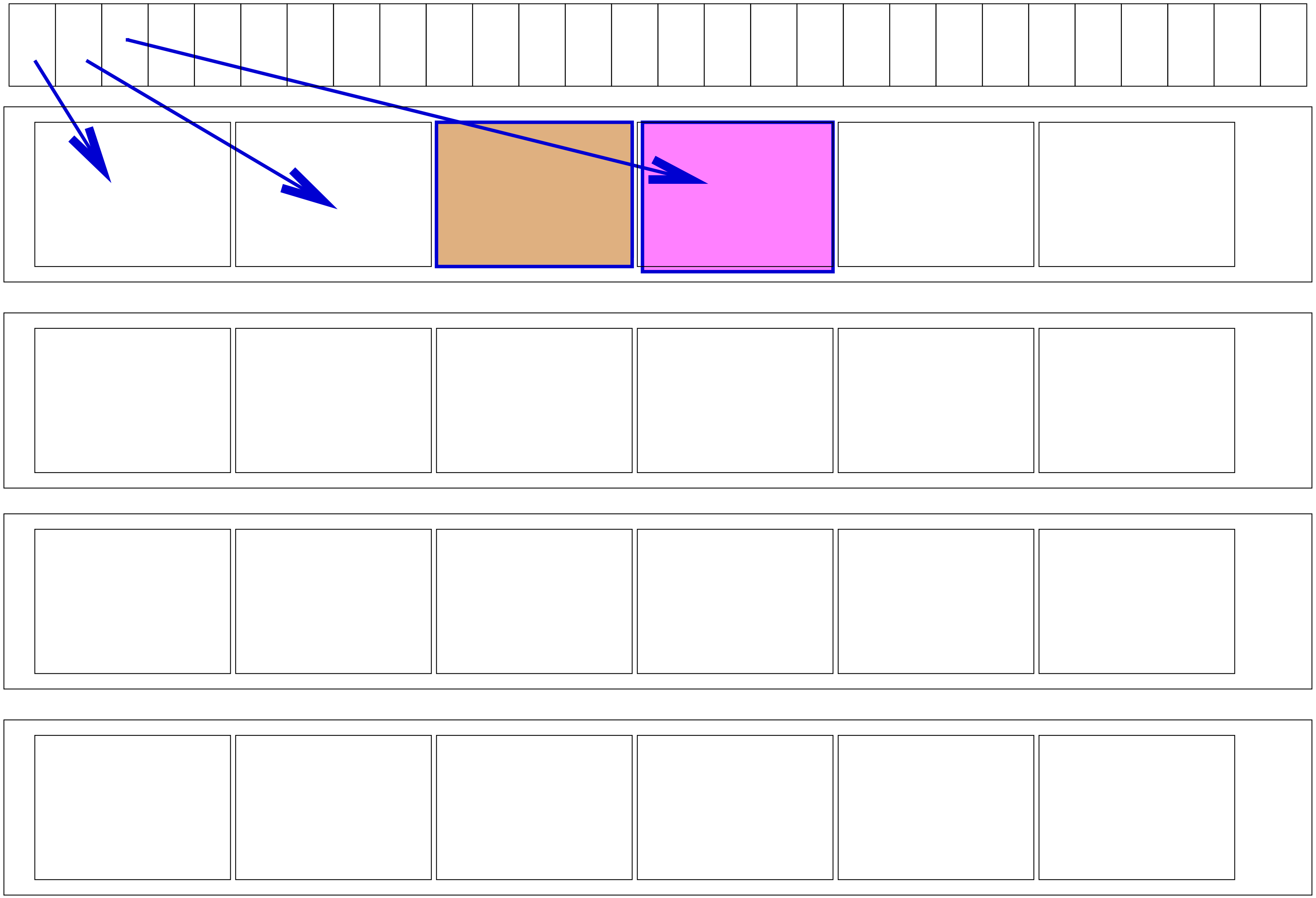
NAND CHARACTERISTICS



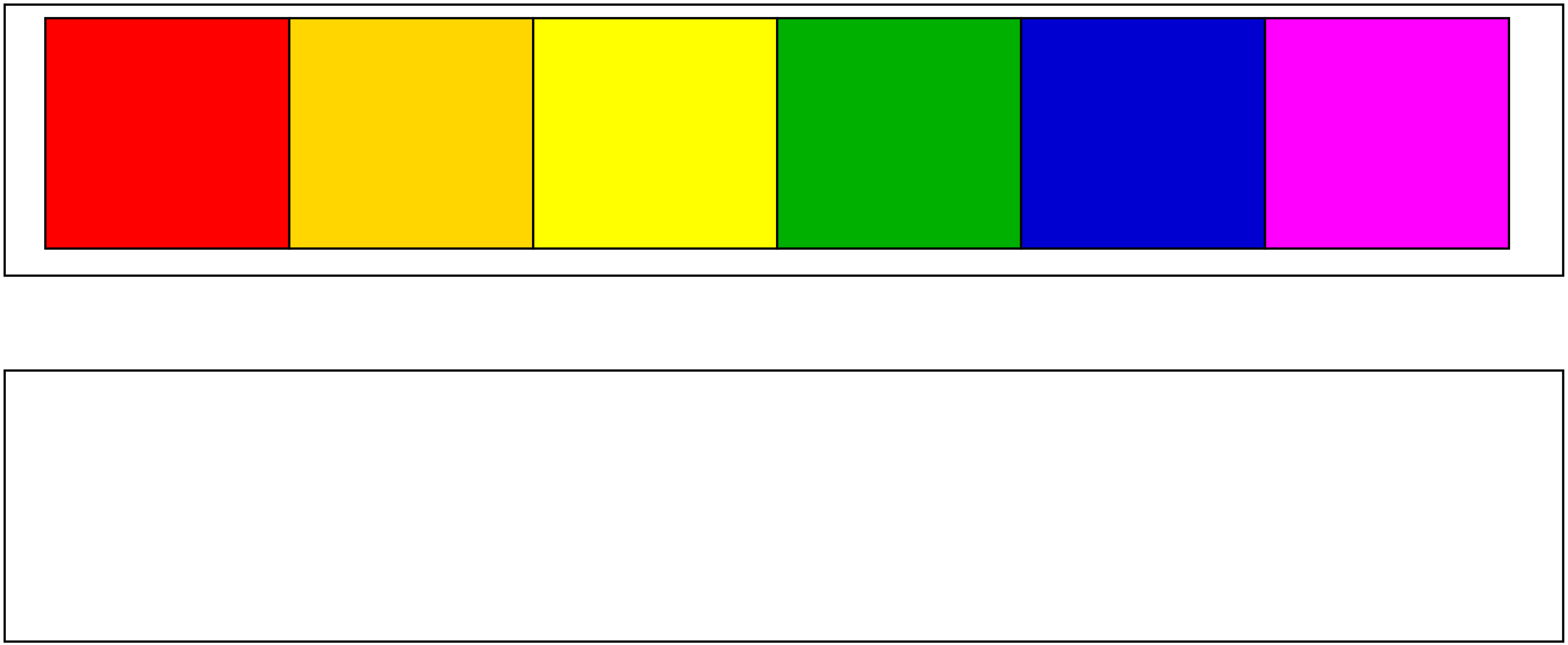
FLASH UPDATE



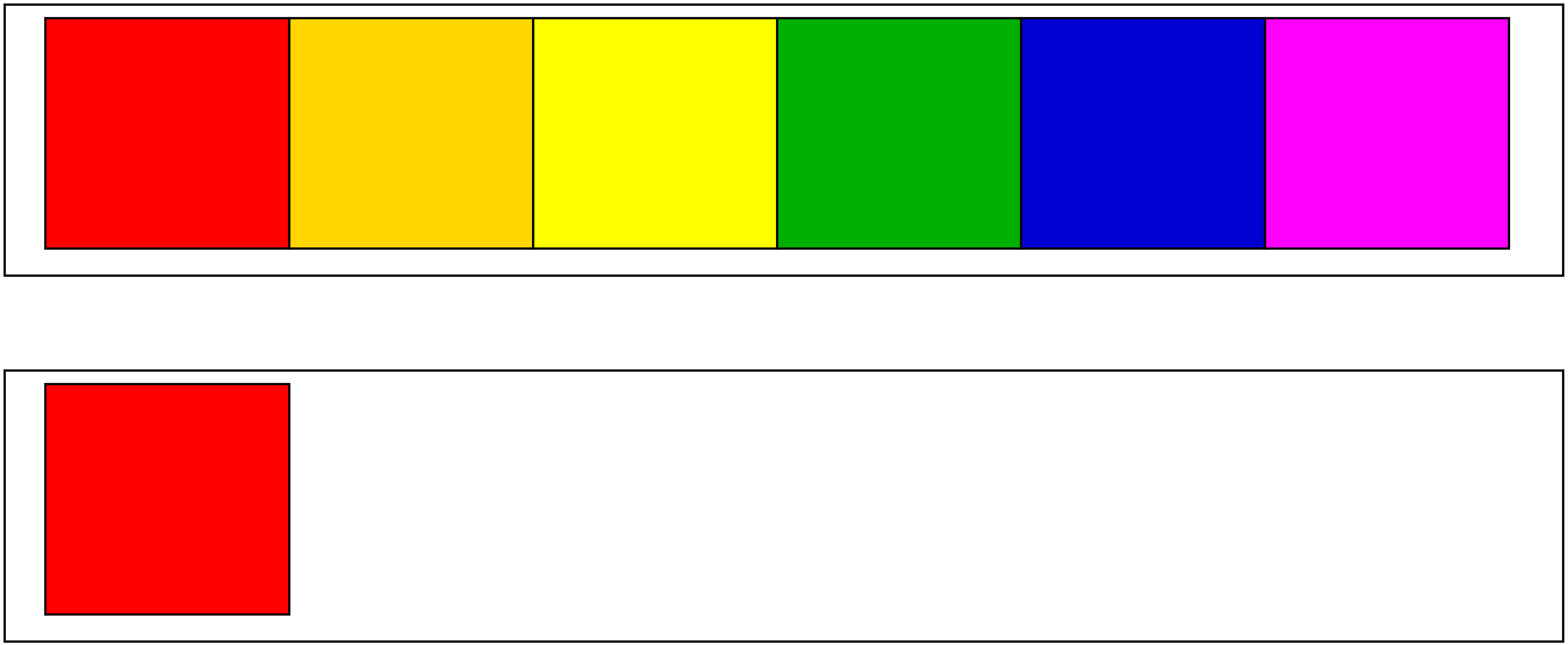
FLASH UPDATE



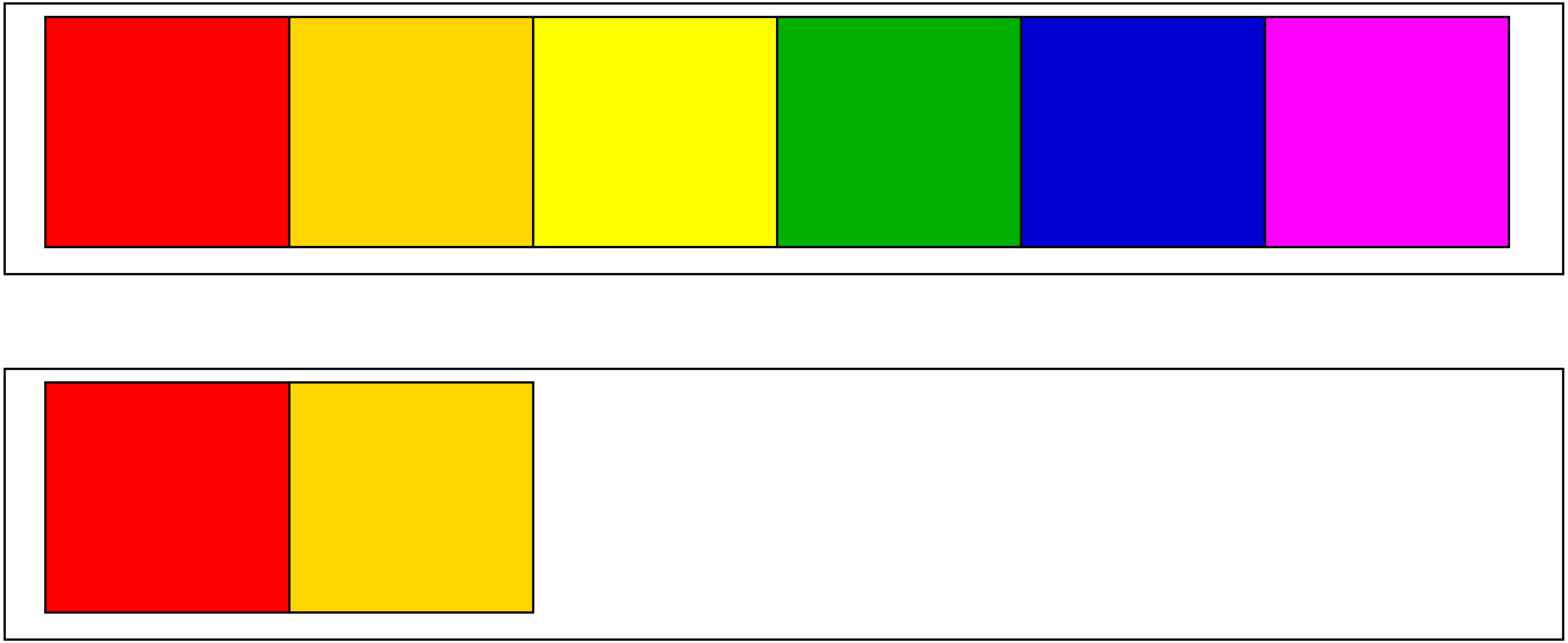
FLASH UPDATE



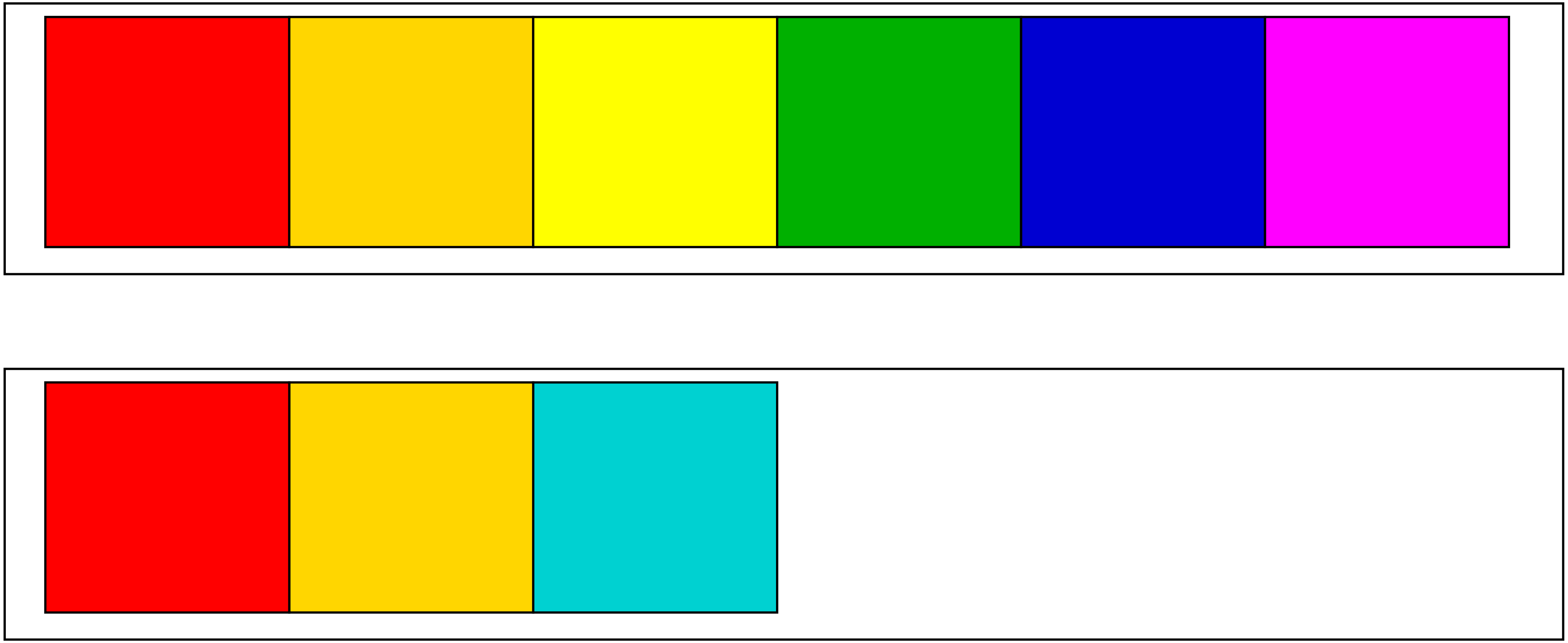
FLASH UPDATE



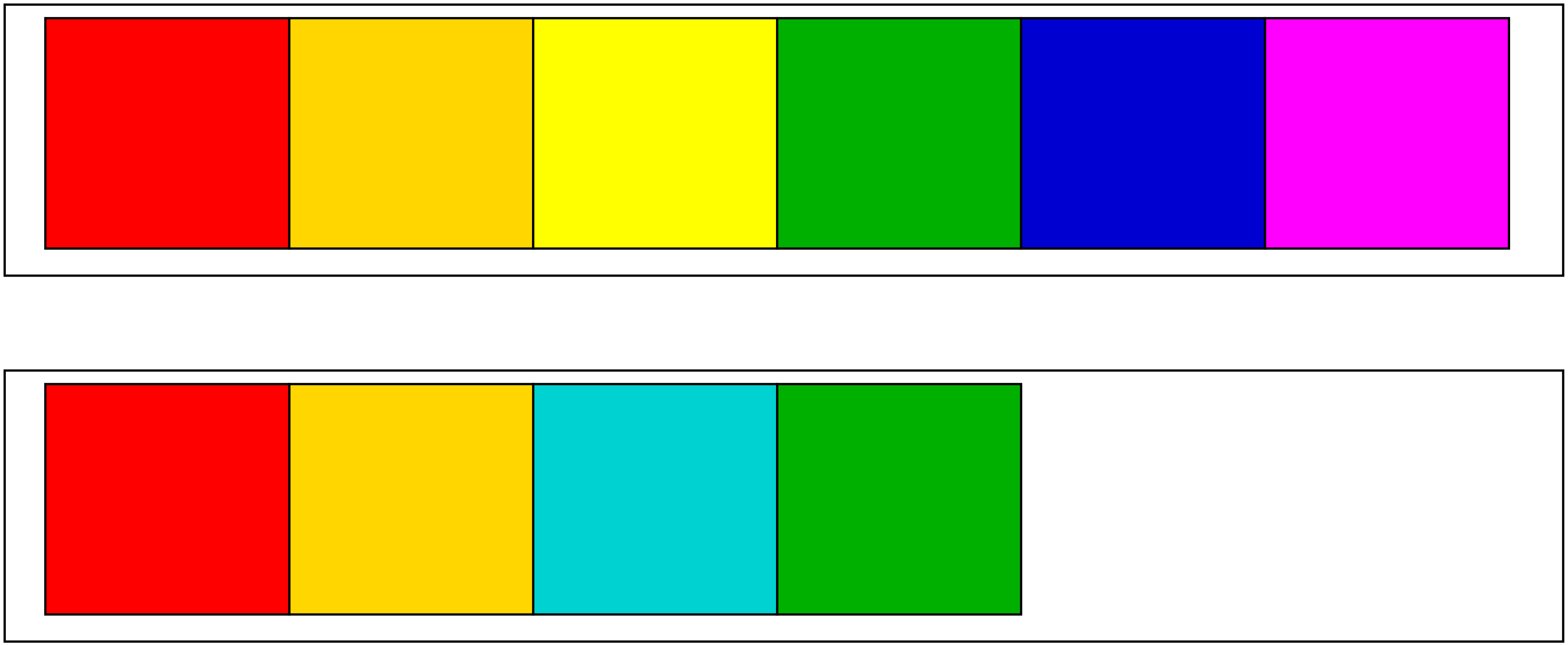
FLASH UPDATE



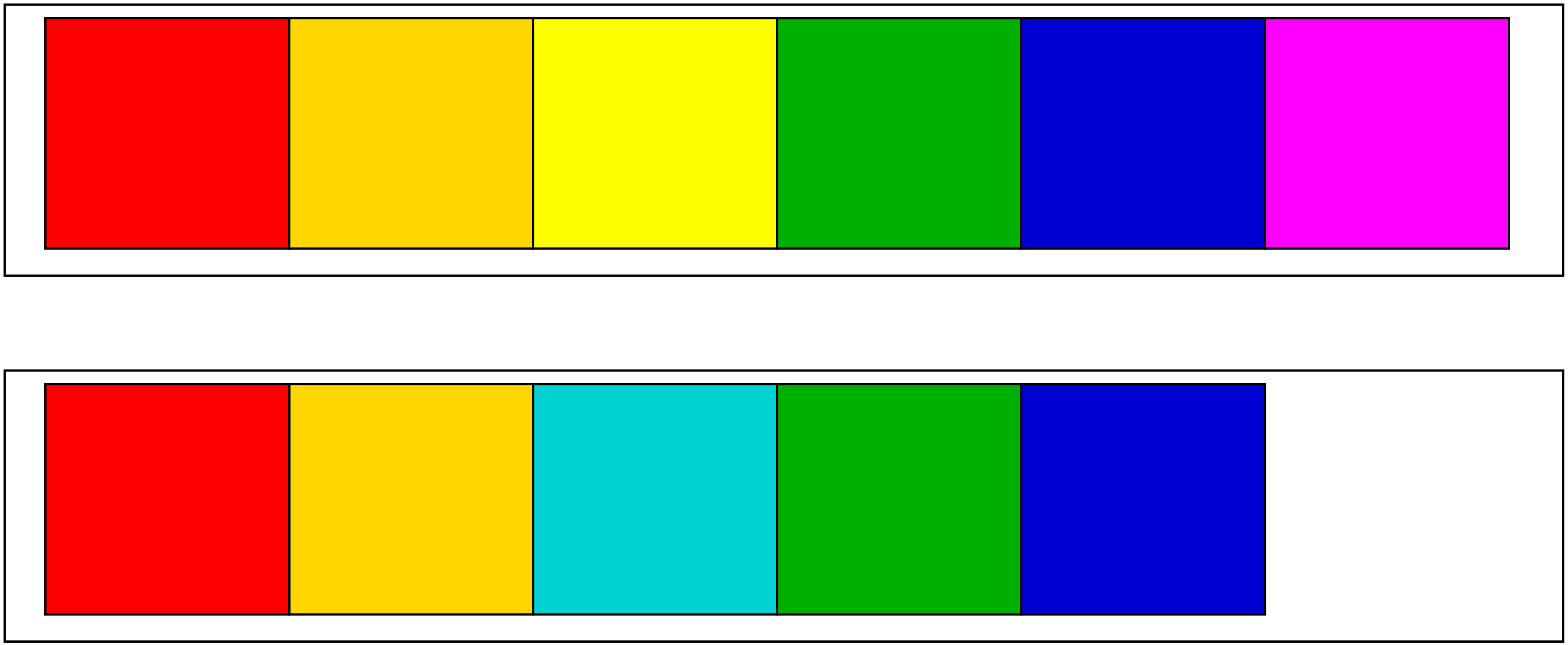
Flash update



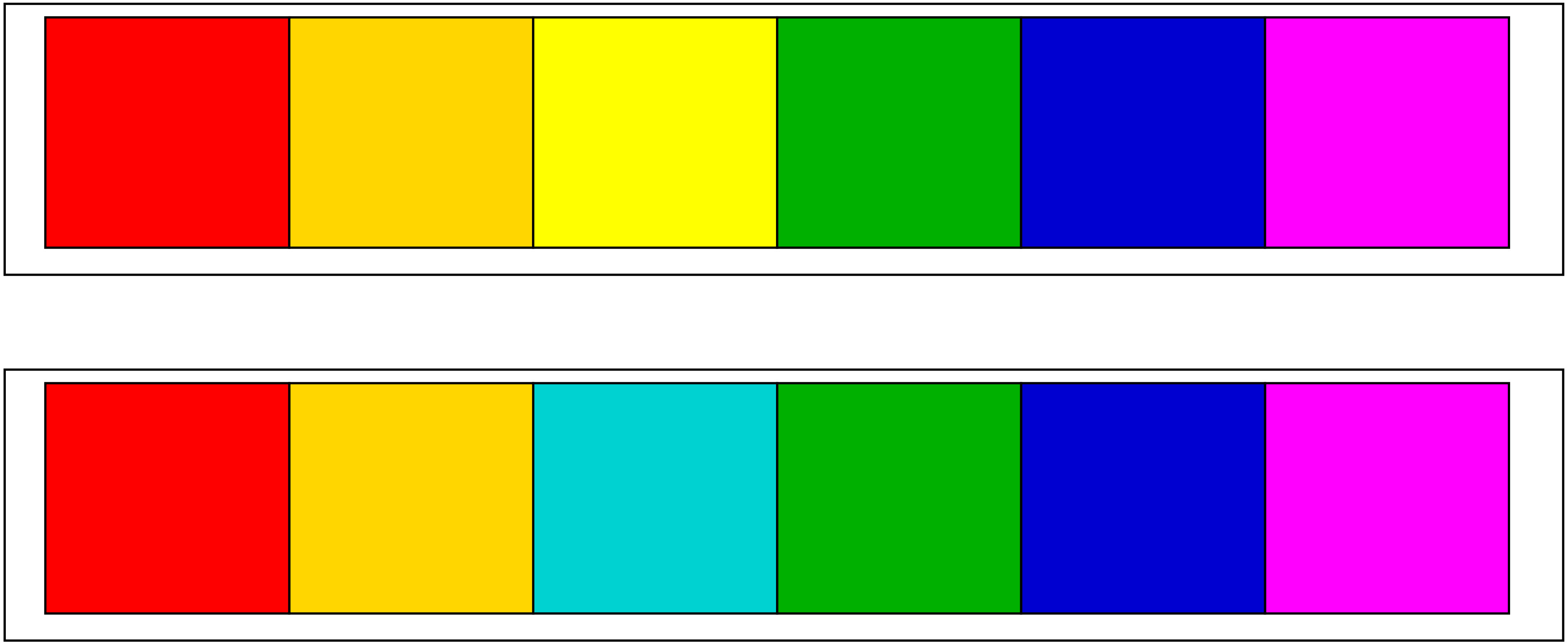
FLASH UPDATE



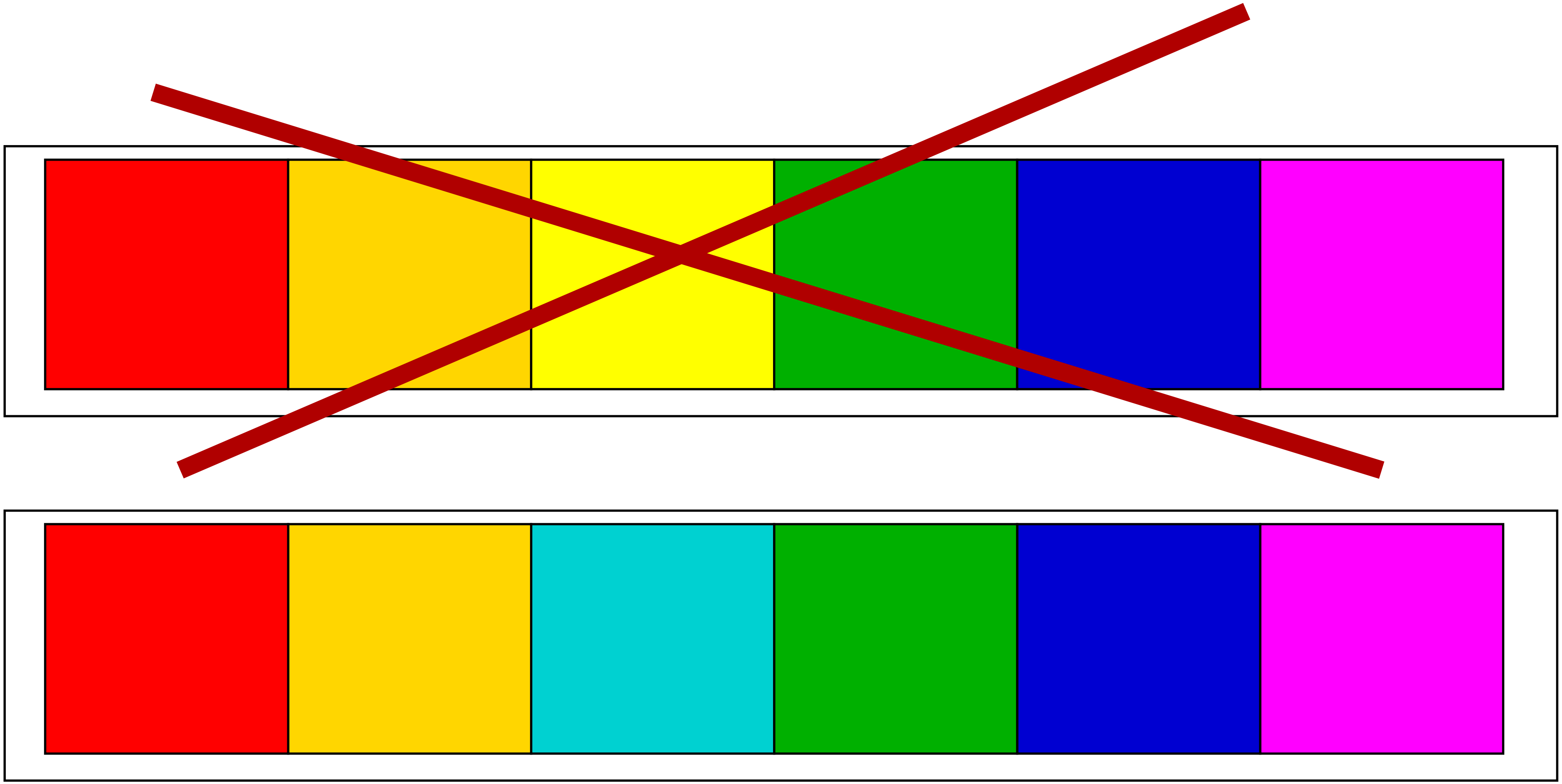
FLASH UPDATE



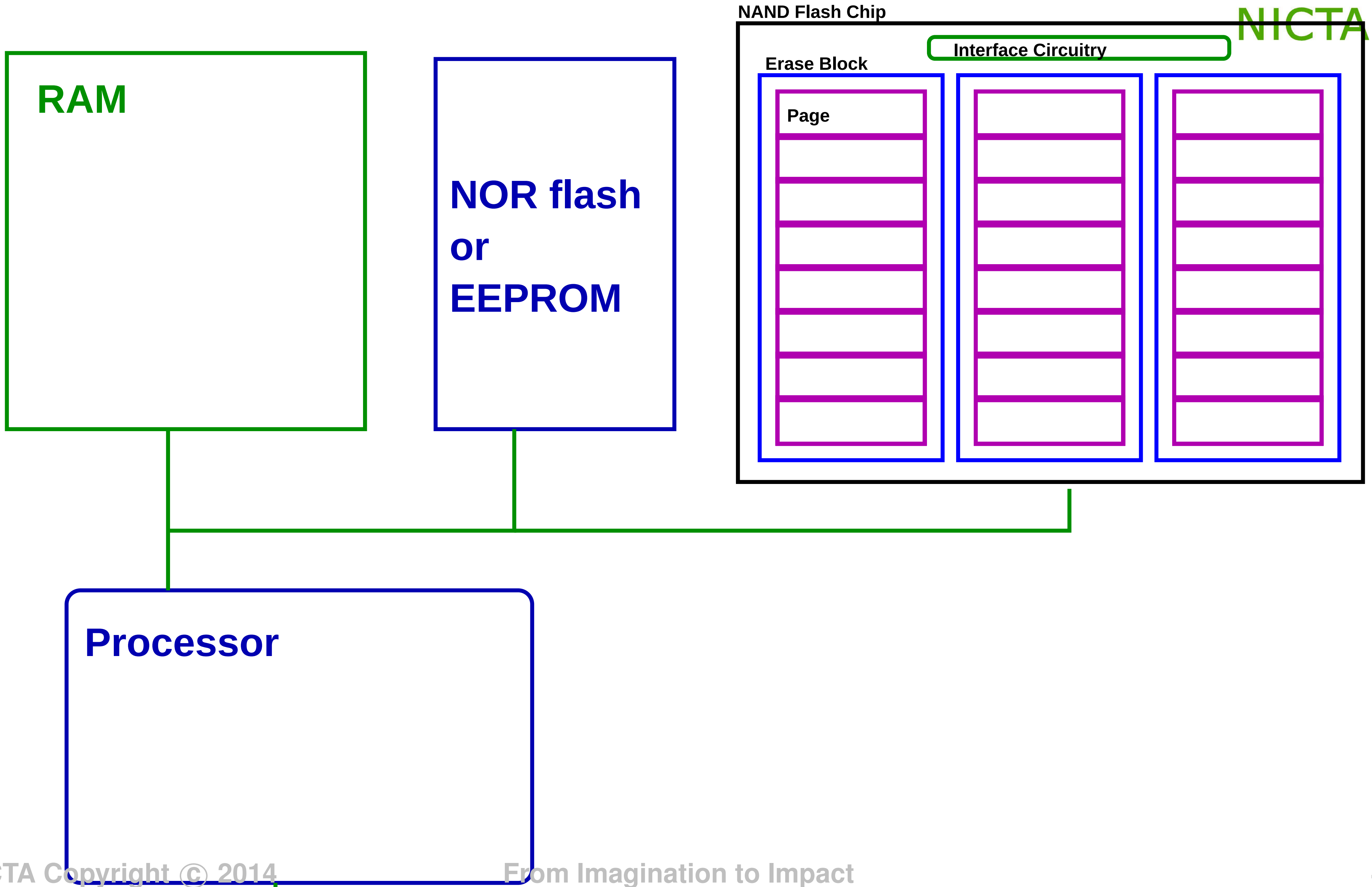
FLASH UPDATE



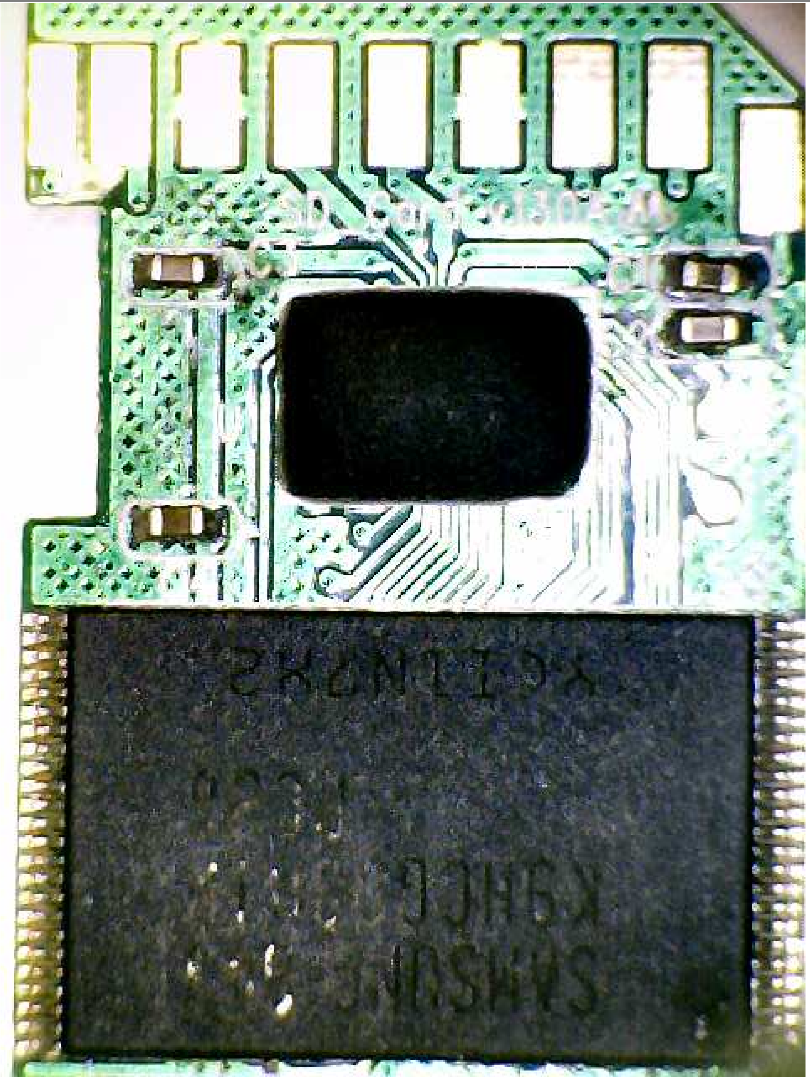
FLASH UPDATE

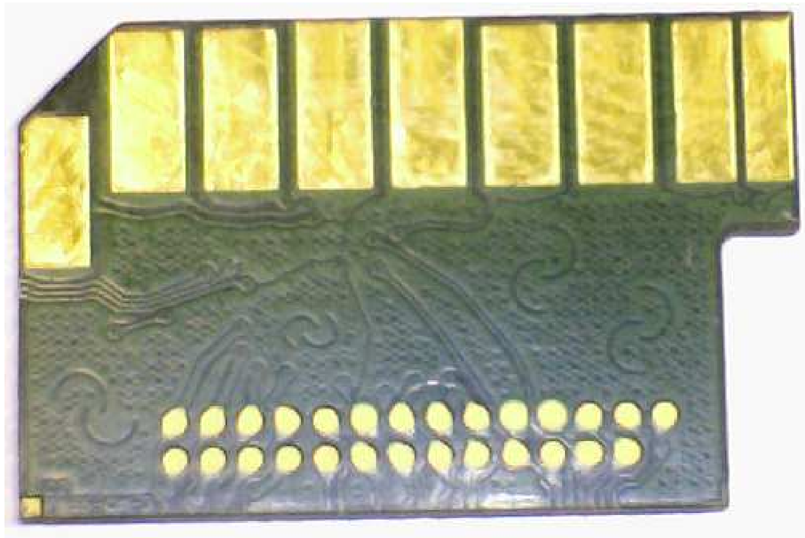


Flash Update



FLASH UPDATE





THE CONTROLLER:

- Presents illusion of ‘standard’ block device
- Manages writes to prevent wearing out
- Manages reads to prevent read-disturb
- Performs garbage collection
- Performs bad-block management

THE CONTROLLER:

- Presents illusion of 'standard' block device
- Manages writes to prevent wearing out
- Manages reads to prevent read-disturb
- Performs garbage collection
- Performs bad-block management

Mostly documented in Korean patents referred to by US patents!

WEAR MANAGEMENT

Two ways:

- Remap blocks when they begin to fail (bad block remapping)

WEAR MANAGEMENT

Two ways:

- Remap blocks when they begin to fail (bad block remapping)
- Spread writes over all erase blocks (wear levelling)

WEAR MANAGEMENT

Two ways:

- Remap blocks when they begin to fail (bad block remapping)
- Spread writes over all erase blocks (wear levelling)

In practice both are used.

Wear Management

Two ways:

- Remap blocks when they begin to fail (bad block remapping)
- Spread writes over all erase blocks (wear levelling)

In practice both are used.
Also:

- Count reads and schedule garbage collection after some threshhold

PREFORMAT

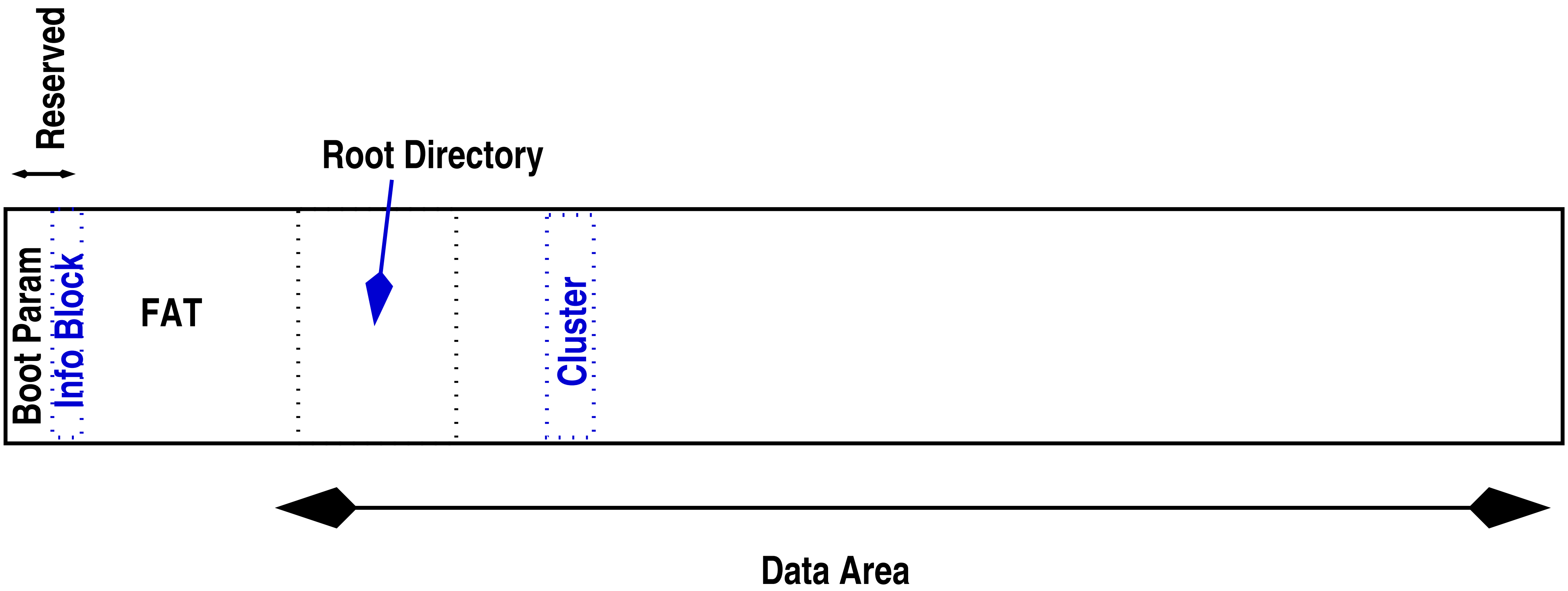
- Typically use FAT32 (or exFAT for sdxc cards)
- Always do cluster-size I/O (64k)
- First partition segment-aligned

PREFORMAT

- Typically use FAT32 (or exFAT for sdxc cards)
- Always do cluster-size I/O (64k)
- First partition segment-aligned

  Conjecture Flash controller optimises for the preformatted FAT fs

FAT FILE SYSTEMS



FAT FILE SYSTEMS

NICTA

Conjecture The controller has some number of buffers it treats specially, to allow more than one write locus.

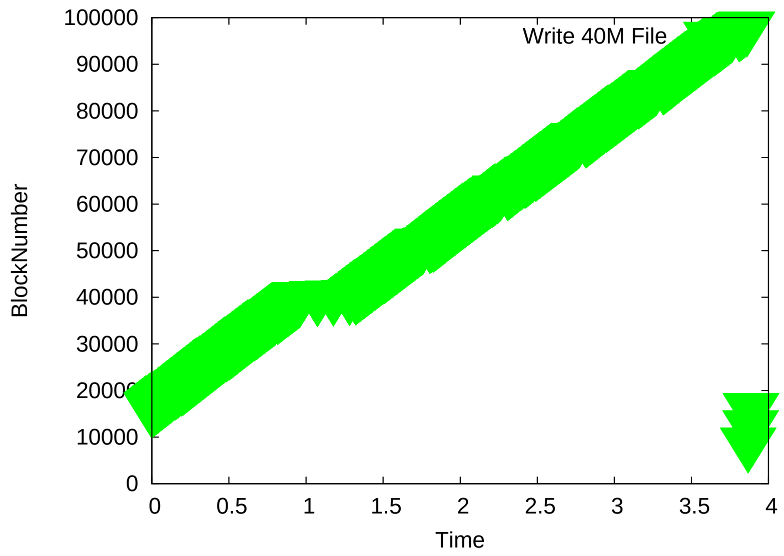
TESTING SDHC CARDS

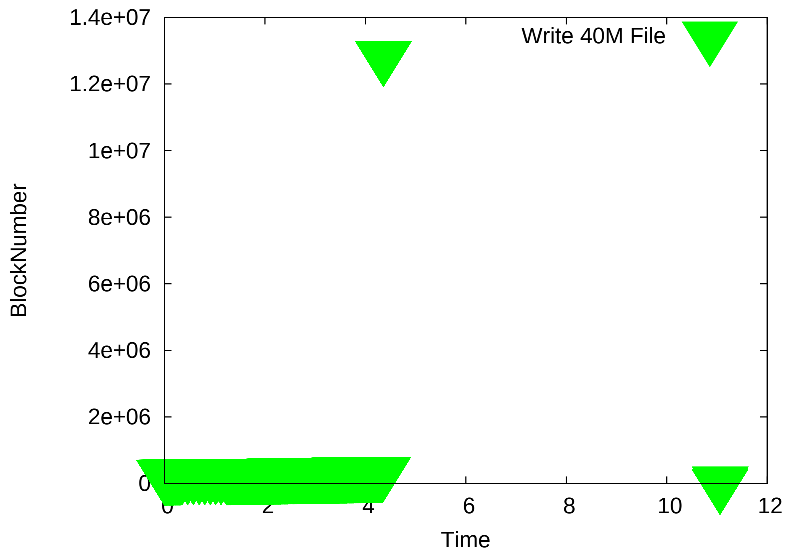
NICTA

SD Card Characteristics

Card	Price/G	#AU	Page size	Erase Size
Kingston Class 10	$0.80	2	128k	4M
Toshiba Class 10	$1.20	2	64k	8M
SanDisk Extreme UHS-1	$5.00	9	64k	8M
SanDisk Ex-	$6.50	9	16k	4M

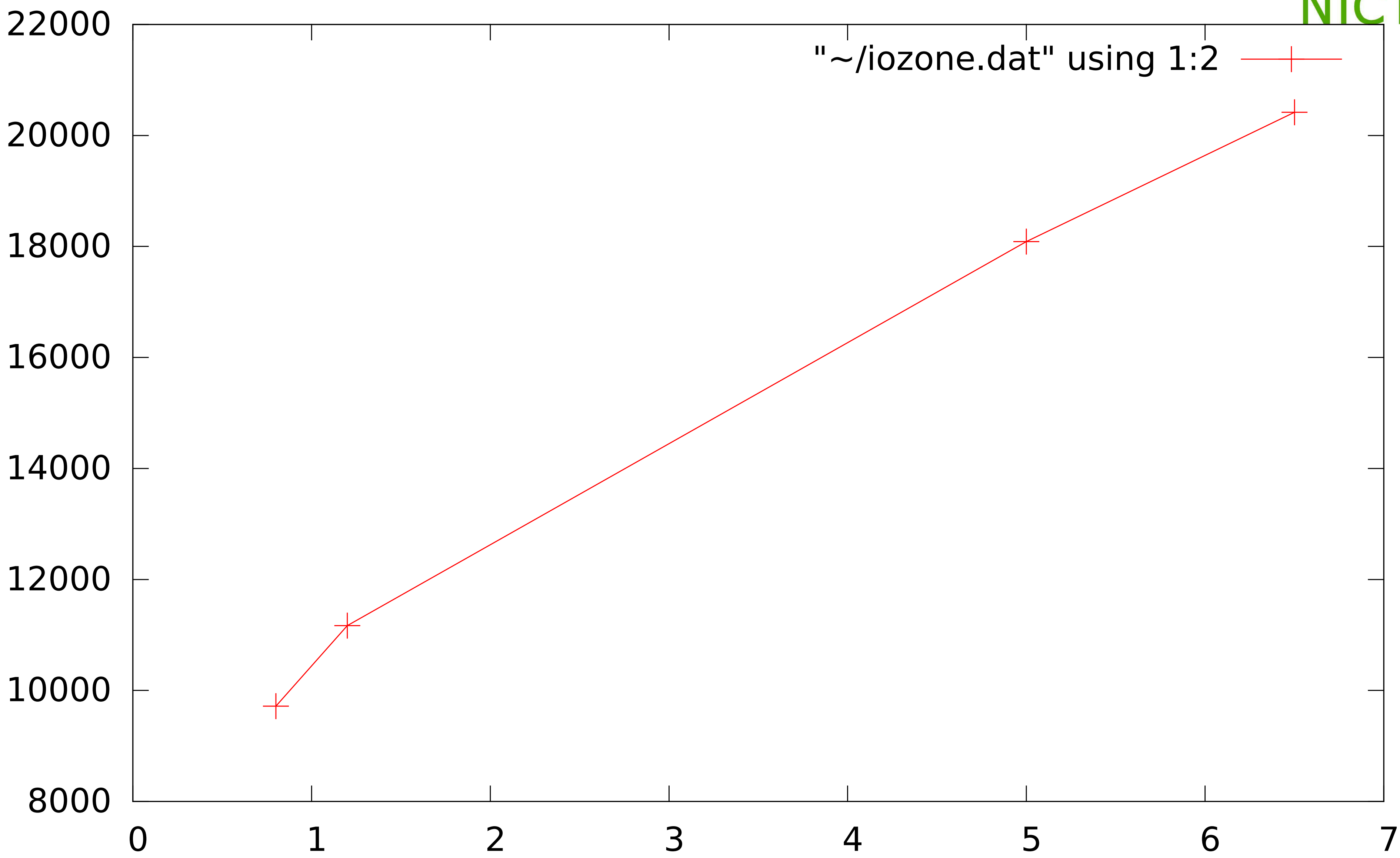
Write Patterns: File Create





(On Toshiba Exceria card)

WRITE PATTERNS: FILE CREATE



F2FS

- By Samsung

F2FS

- By Samsung
- ‘Use on-card FTL, rather than work against it’

F2FS

- By Samsung
- ‘Use on-card FTL, rather than work against it’
- Cooperate with garbage collection

F2FS

- By Samsung
- ‘Use on-card FTL, rather than work against it’
- Cooperate with garbage collection
- Use FAT32 optimisations

F2FS

- 2M Segments written as whole chunks

F2FS

- 2M Segments written as whole chunks — always writes at log head

F2FS

NICTA

- 2M Segments written as whole chunks — always writes at log head
  — aligned with FLASH allocation units

F2FS

- 2M Segments written as whole chunks — always writes at log head
  — aligned with FLASH allocation units
- Log is the only data structure on-disk

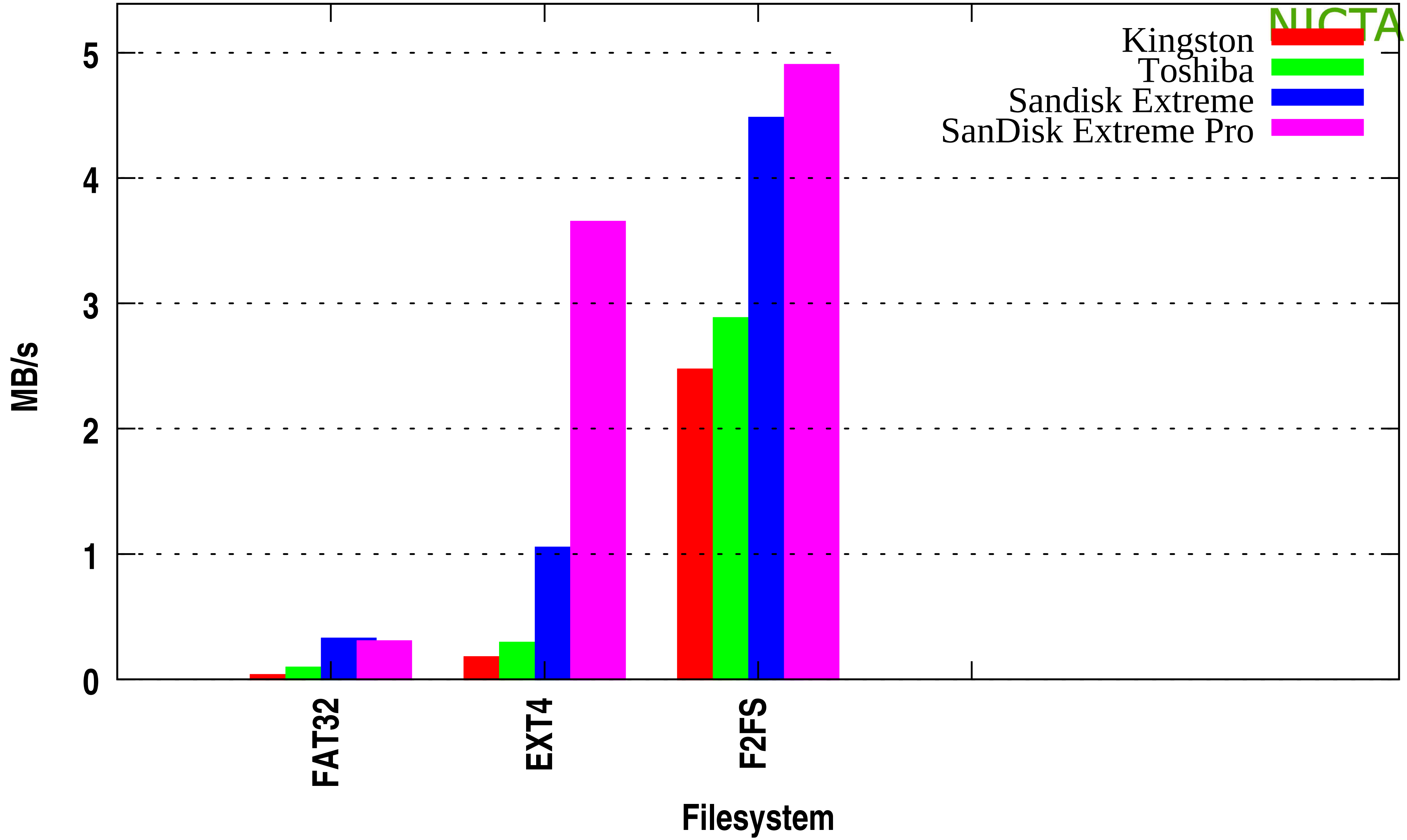
F2FS

- 2M Segments written as whole chunks — always writes at log head
  — aligned with FLASH allocation units
- Log is the only data structure on-disk
- Metadata (e.g., head of log) written to FAT area in single-block writes

F2FS

- 2M Segments written as whole chunks — always writes at log head
  — aligned with FLASH allocation units
- Log is the only data structure on-disk
- Metadata (e.g., head of log) written to FAT area in single-block writes
- Splits Hot and Cold data and Inodes.

BENCHMARKS: POSTMARK 32K READ



USING NON-F2FS

- Observation: XFS and ext4 already understand RAID

USING NON-F2FS

- Observation: XFS and ext4 already understand RAID
- RAID has multiple chunks, and a fixed stride, so...

USING NON-F2FS

NICTA

- Observation: XFS and ext4 already understand RAID
- RAID has multiple chunks, and a fixed stride, so...
- Configure FS as if for RAID

USING NON-F2FS

Still running benchmarks, see LCA talk next January for results!

NICTA

NICTA

The Multiprocessor Effect:

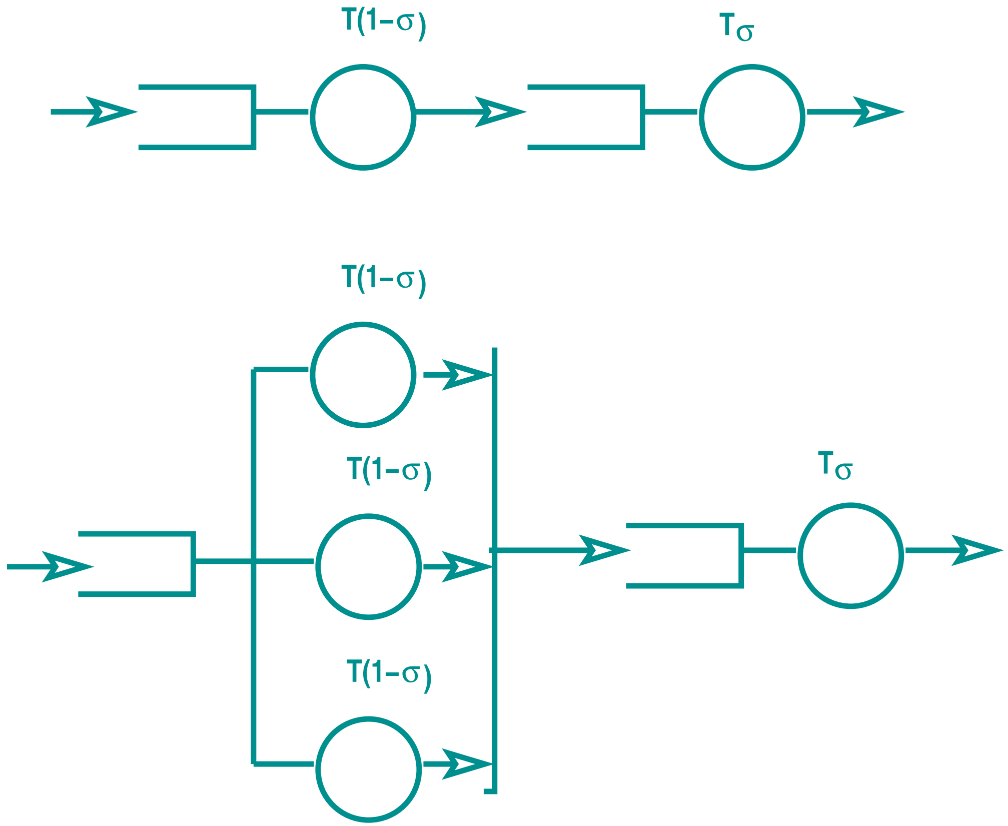
- Some fraction of the system's cycles are not available for application work:
  - Operating System Code Paths
  - Inter-Cache Coherency traffic
  - Memory Bus contention
  - Lock synchronisation
  - I/O serialisation

SCALABILITY

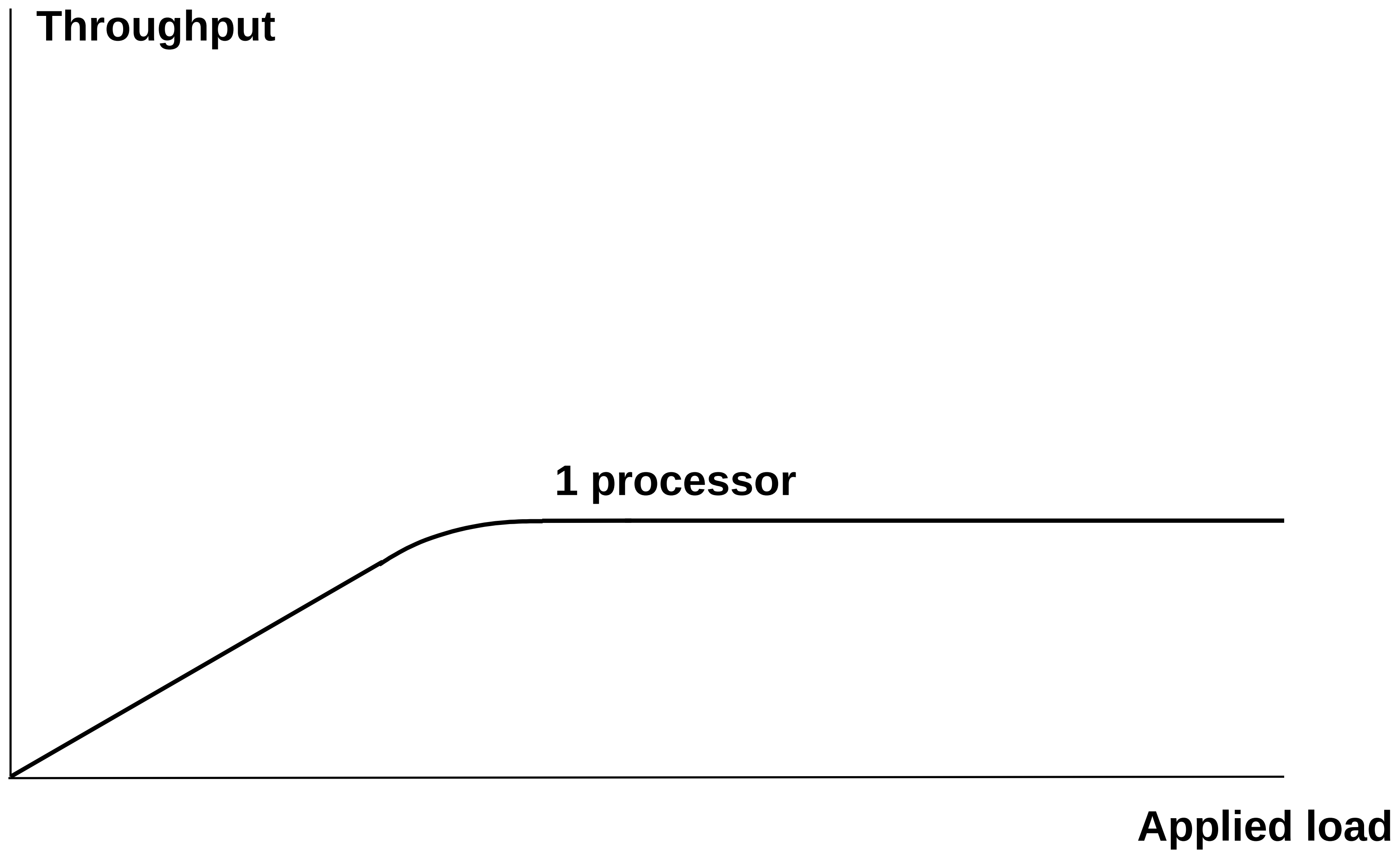
Amdahl's law:

If a process can be split such that σ of the running time cannot be sped up, but the rest is sped up by running on p processors, then overall speedup is

$$\frac{p}{1+\sigma(p-1)}$$

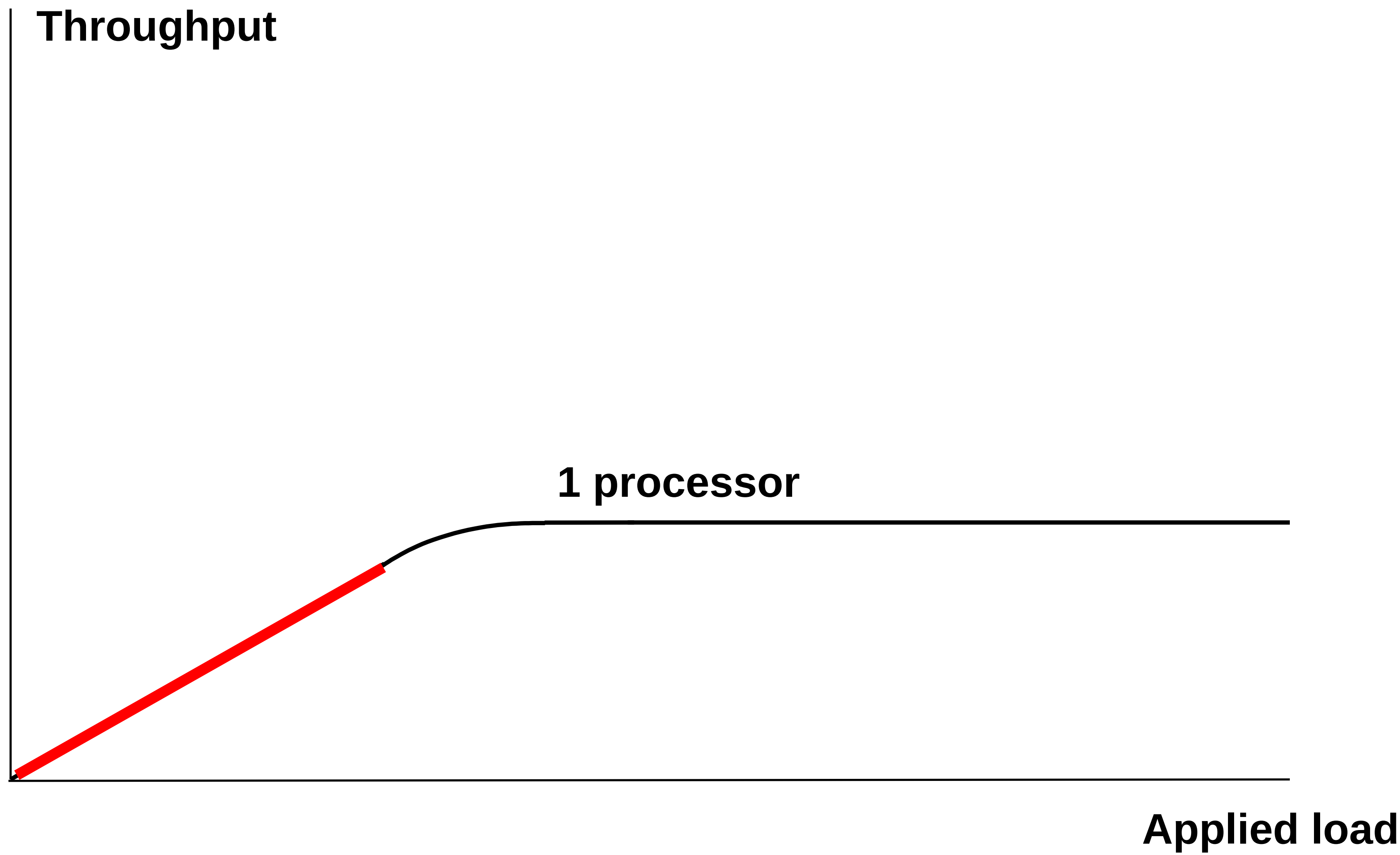


SCALABILITY

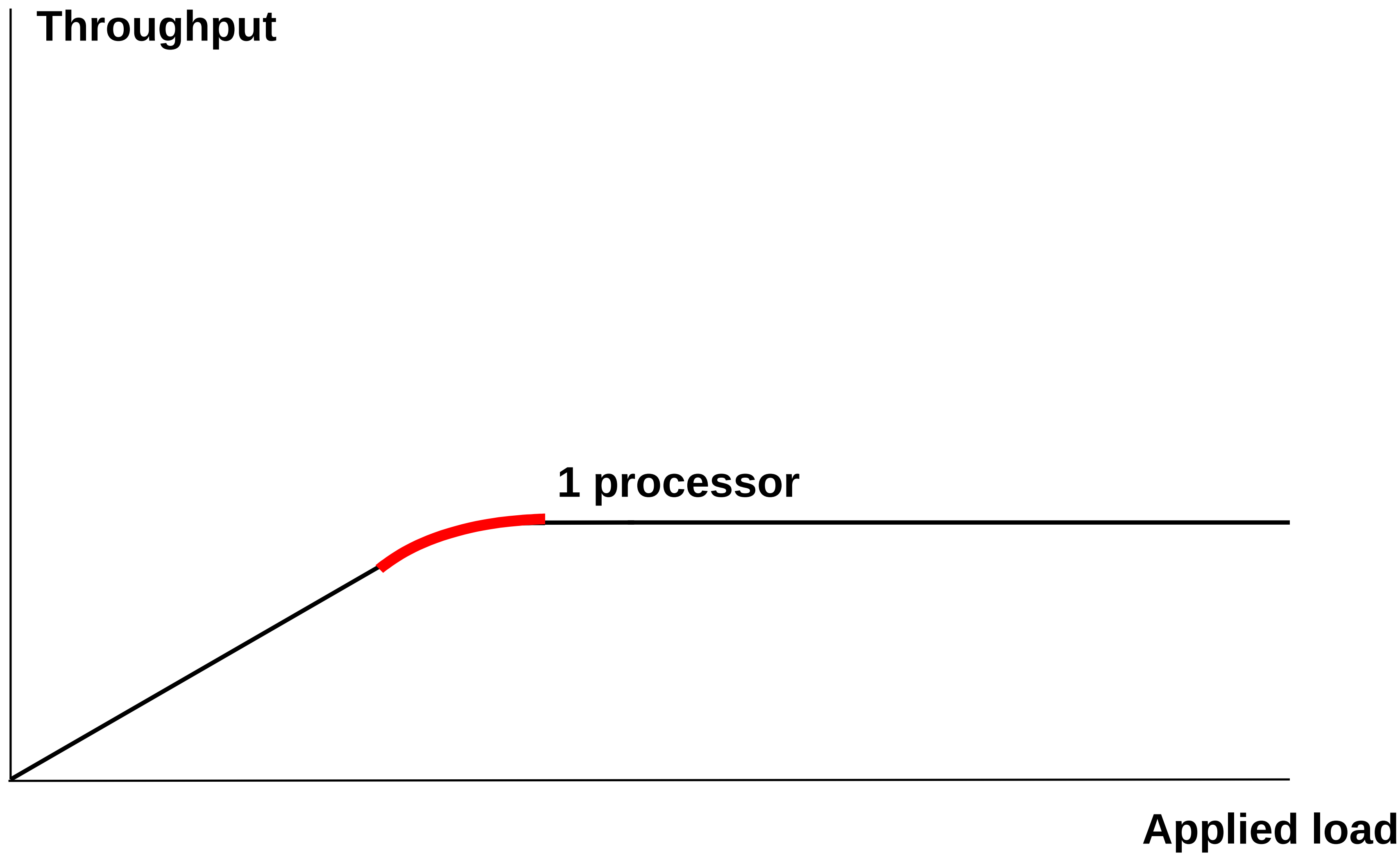


SCALABILITY

NICTA

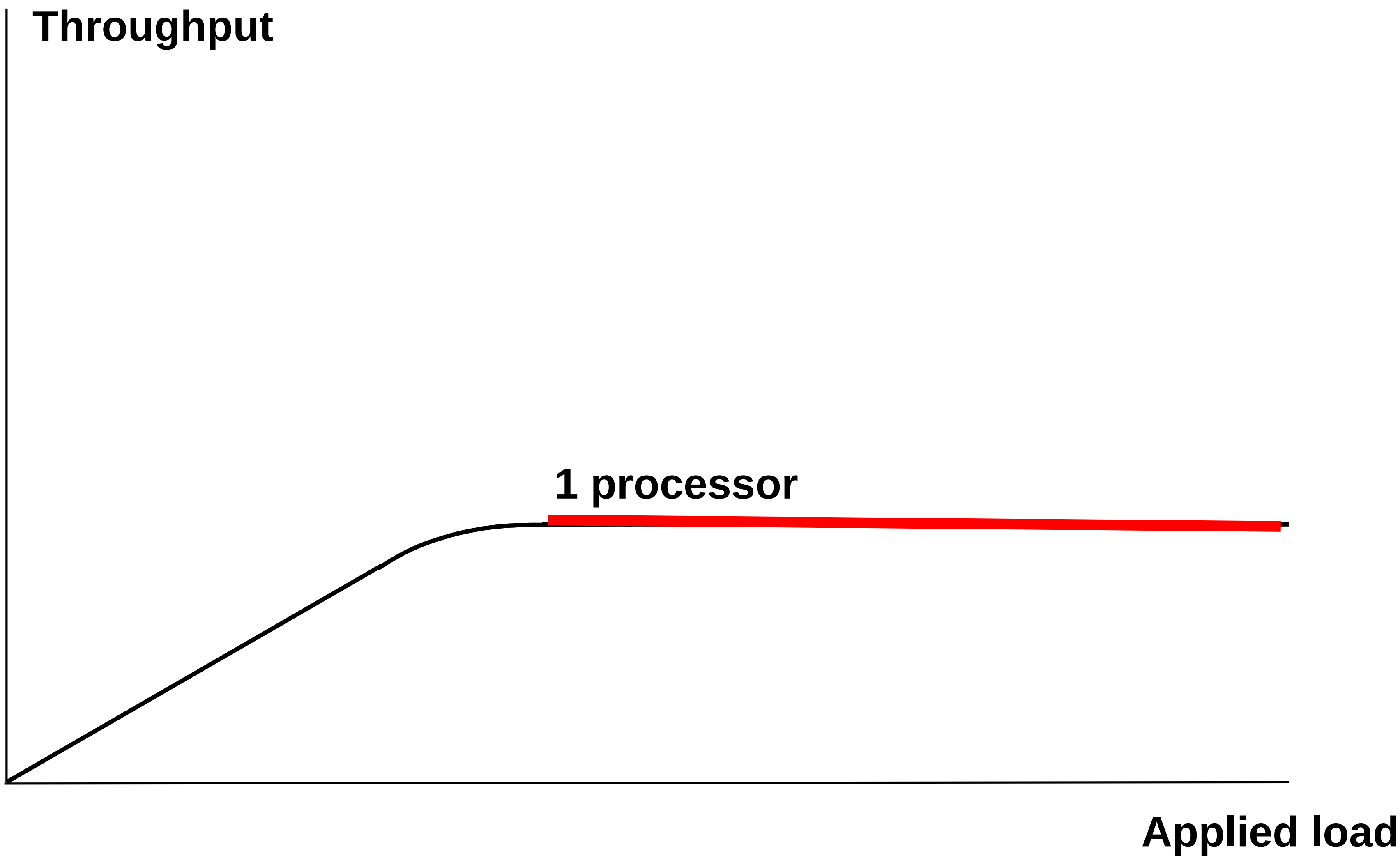


SCALABILITY

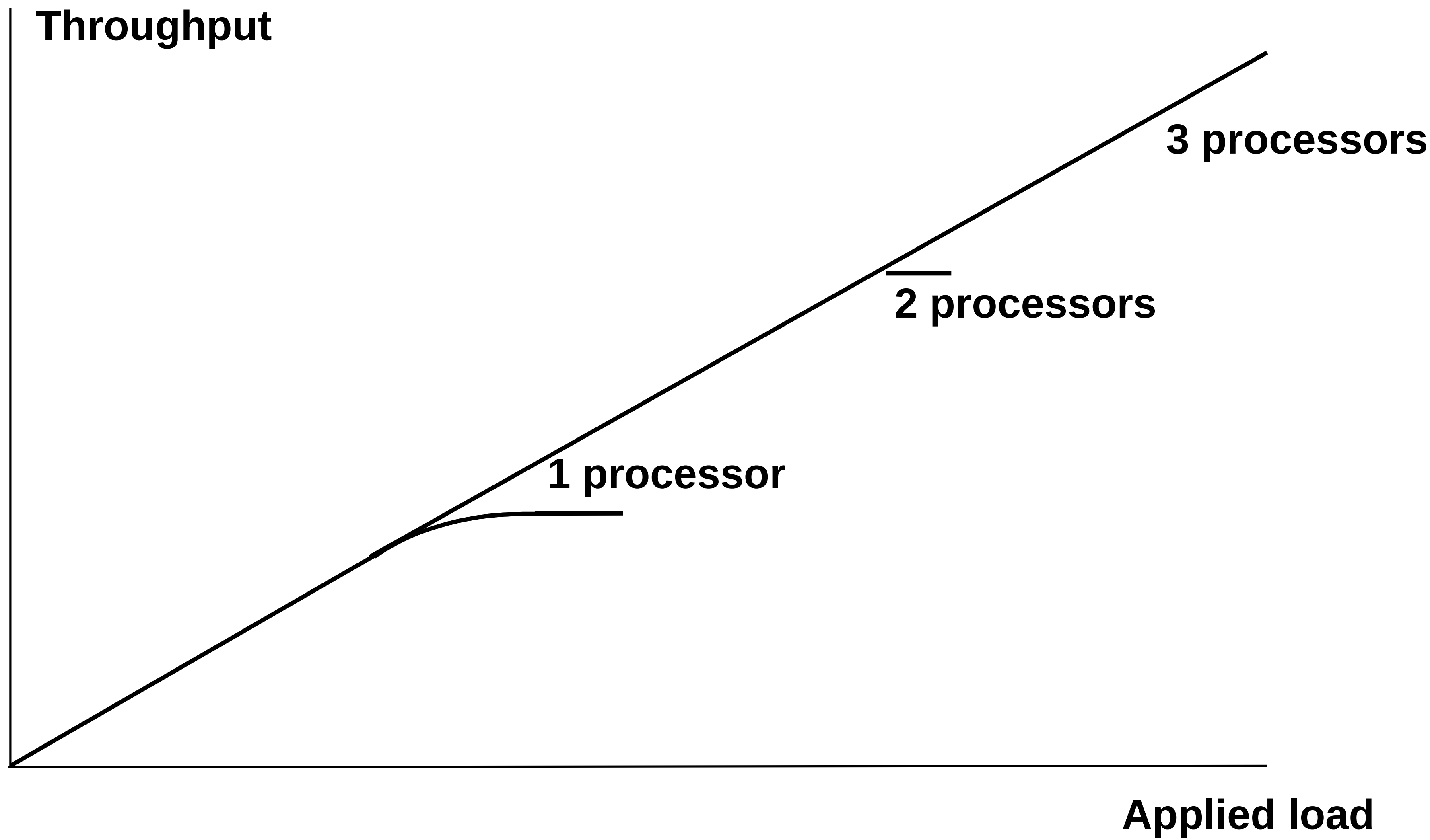


SCALABILITY

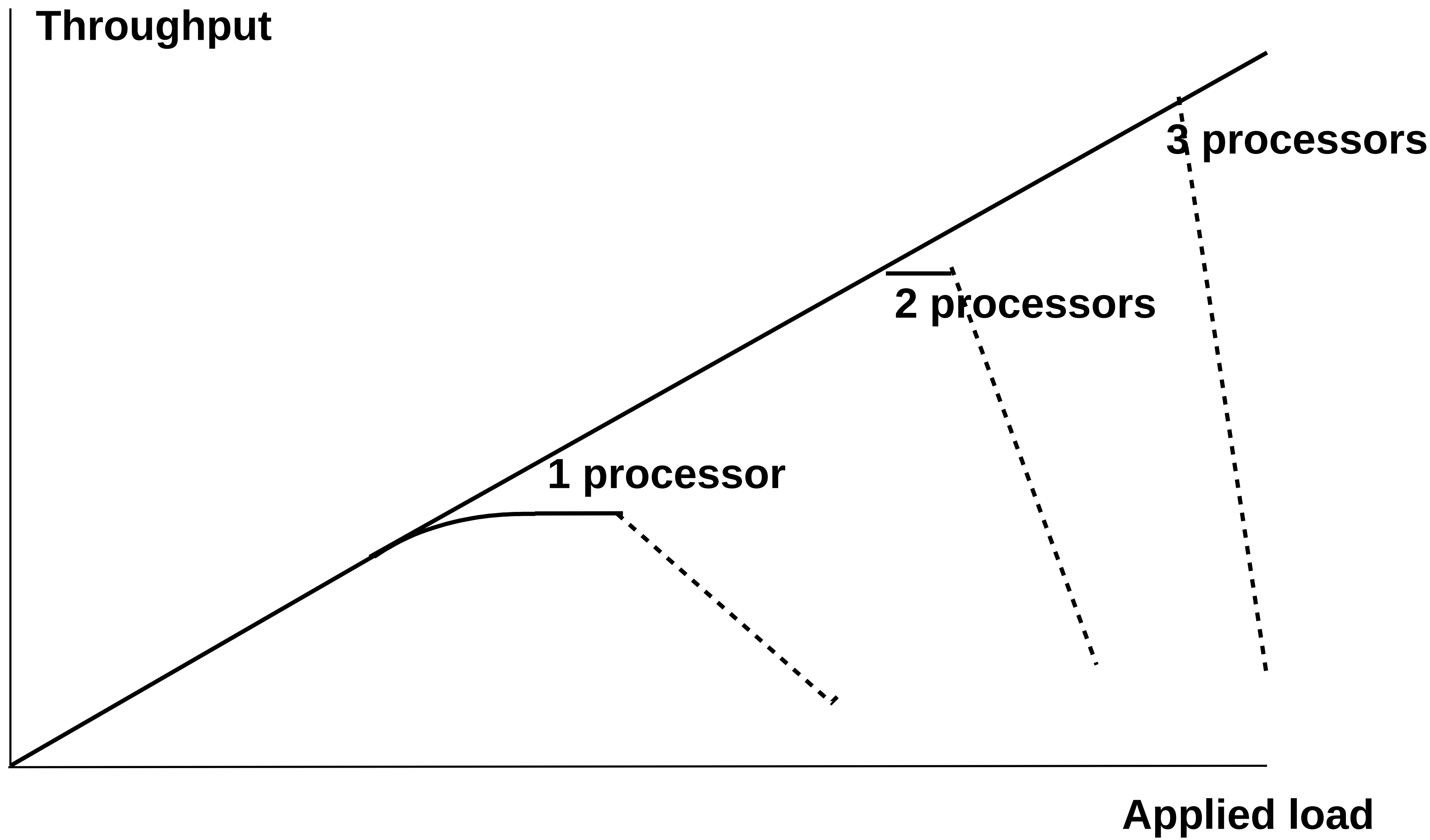
NICTA



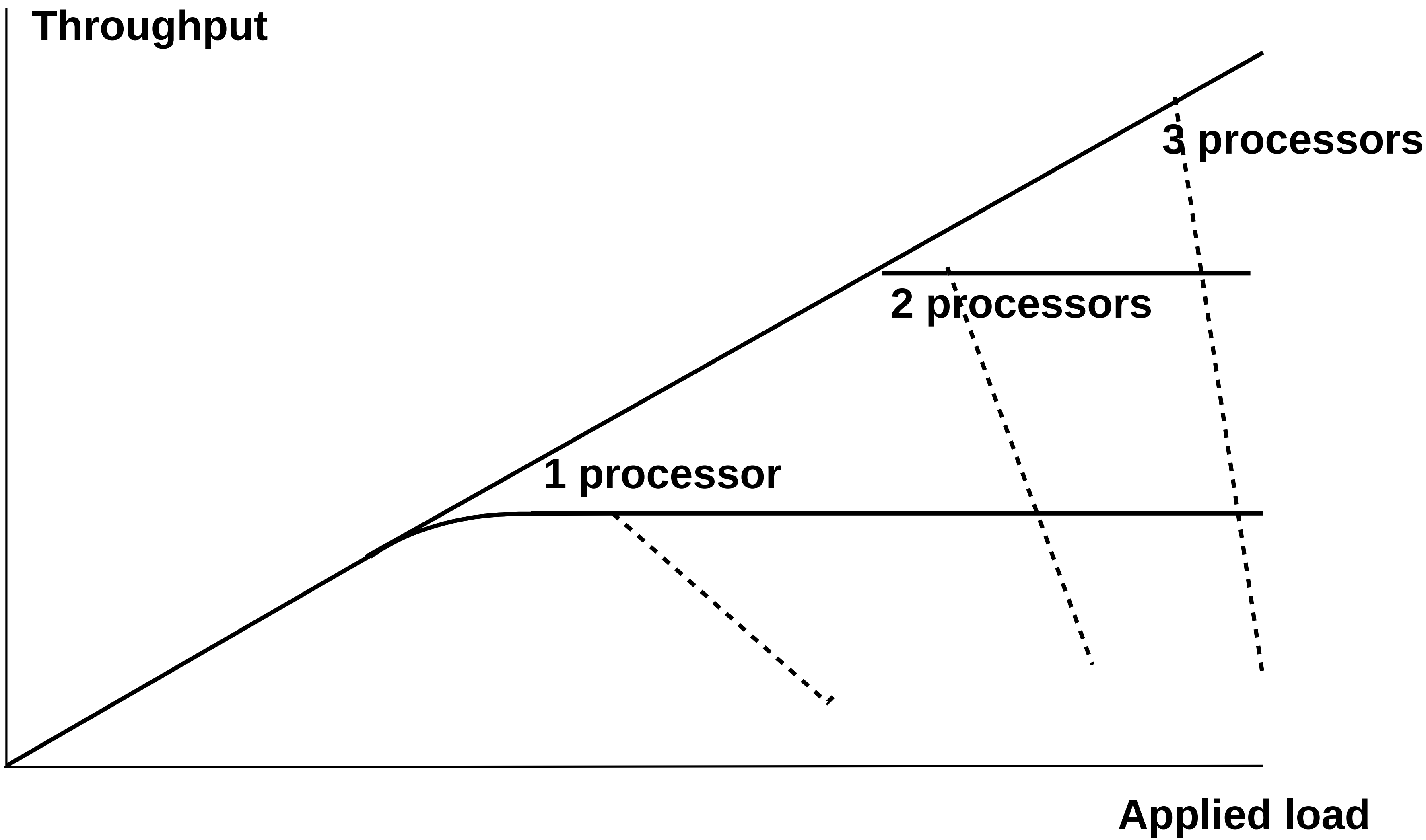
SCALABILITY



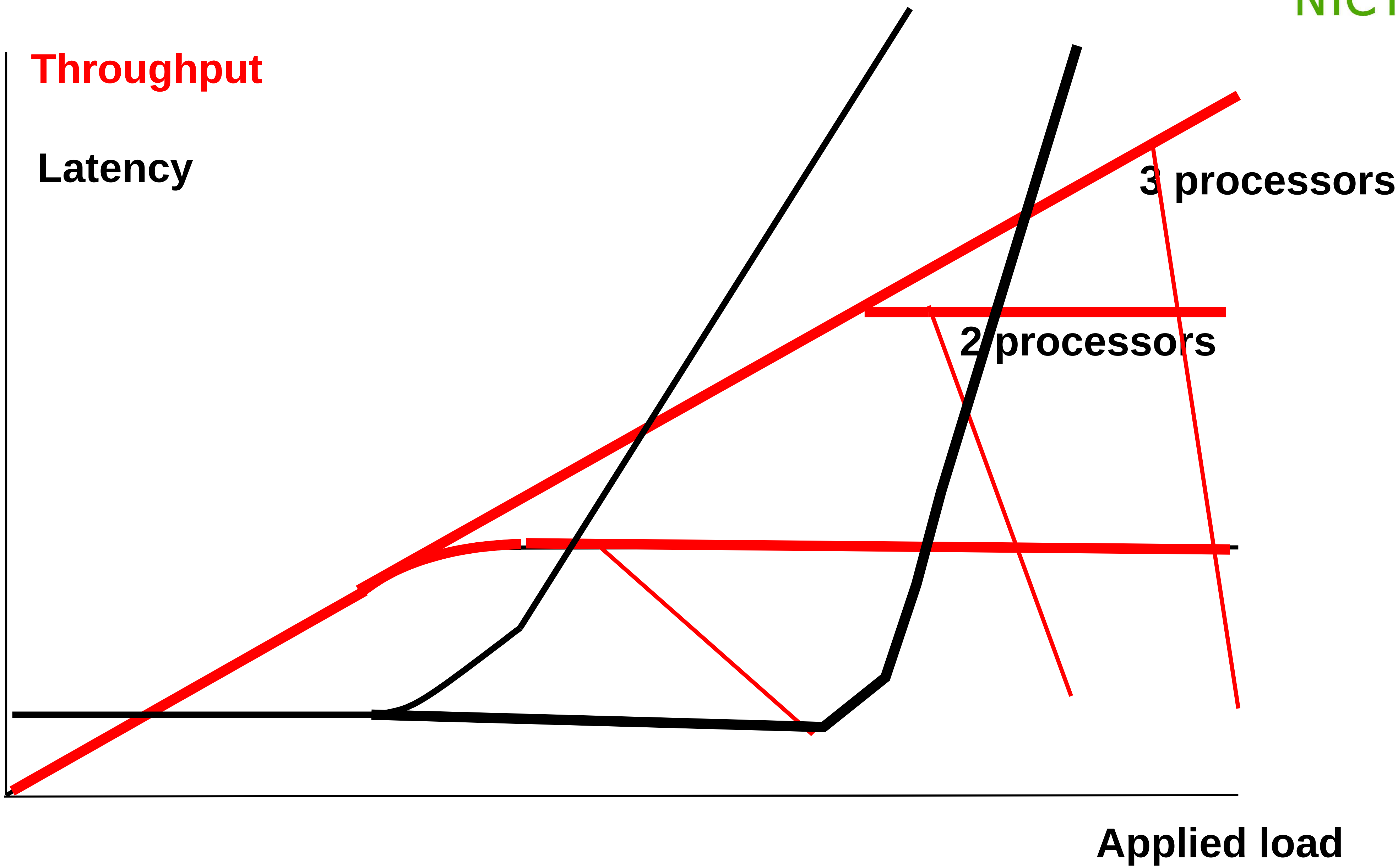
SCALABILITY



SCALABILITY



Scalability



Scalability

Gunther's law:

$$C(N) = \frac{N}{1 + \alpha(N-1) + \beta N(N-1)}$$

where:

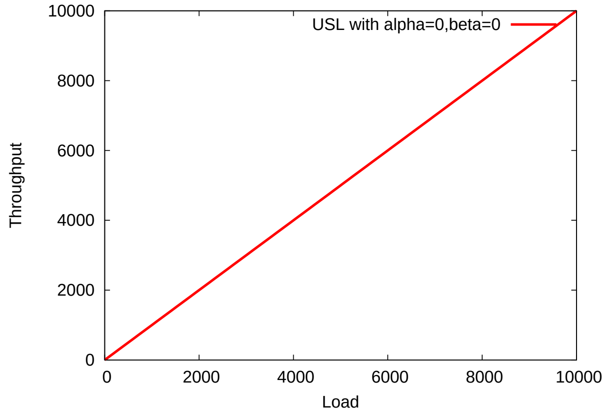
N is demand

α is the amount of serialisation: represents Amdahl's law

β is the coherency delay in the system.

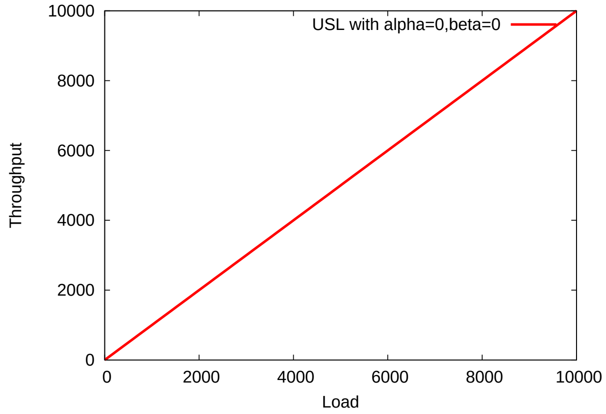
C is Capacity or Throughput

Scalability

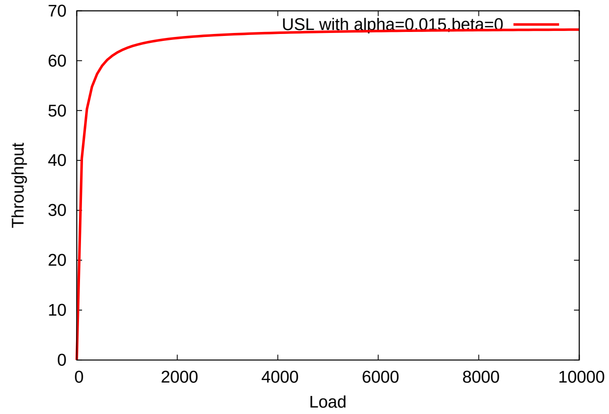


$$\alpha = 0, \beta = 0$$

Scalability

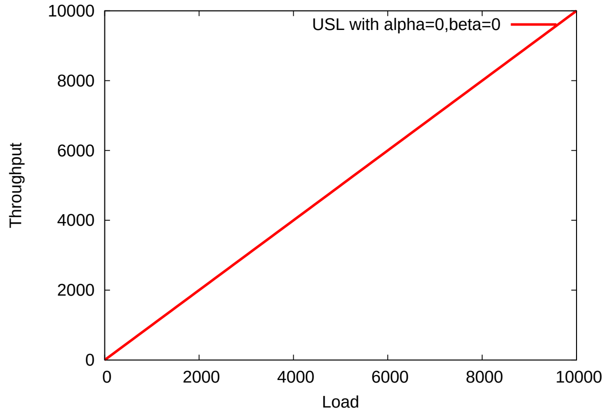


$$\alpha = 0, \beta = 0$$

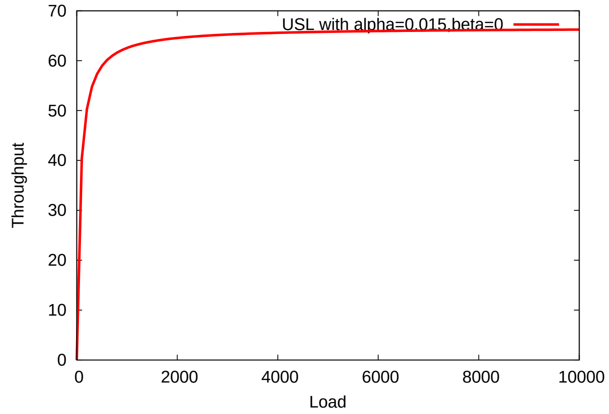


$$\alpha > 0, \beta = 0$$

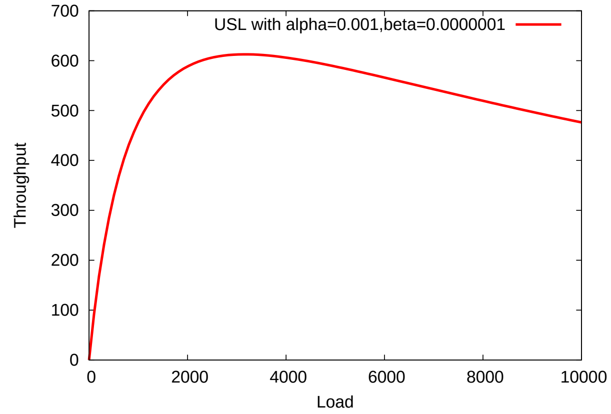
Scalability



$$\alpha = 0, \beta = 0$$



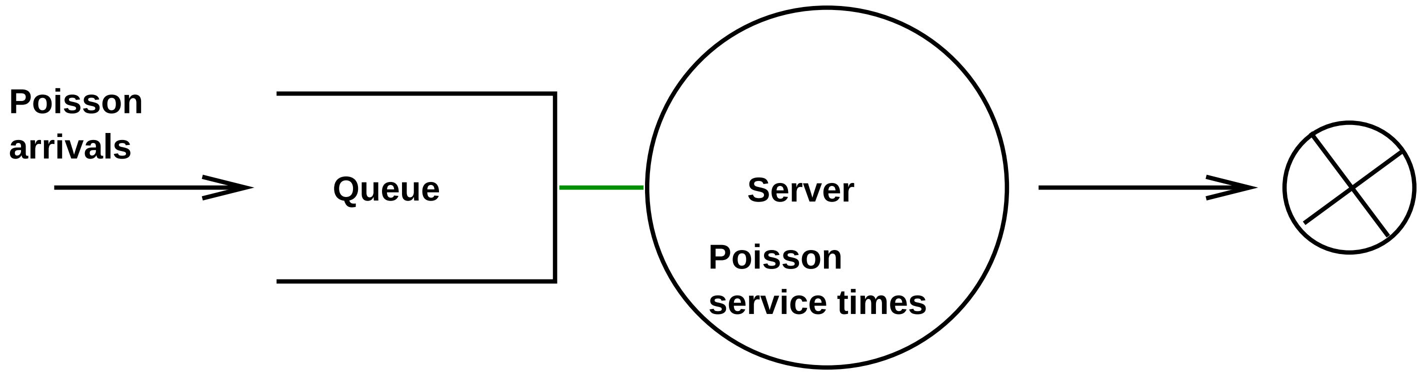
$$\alpha > 0, \beta = 0$$



$$\alpha > 0, \beta > 0$$

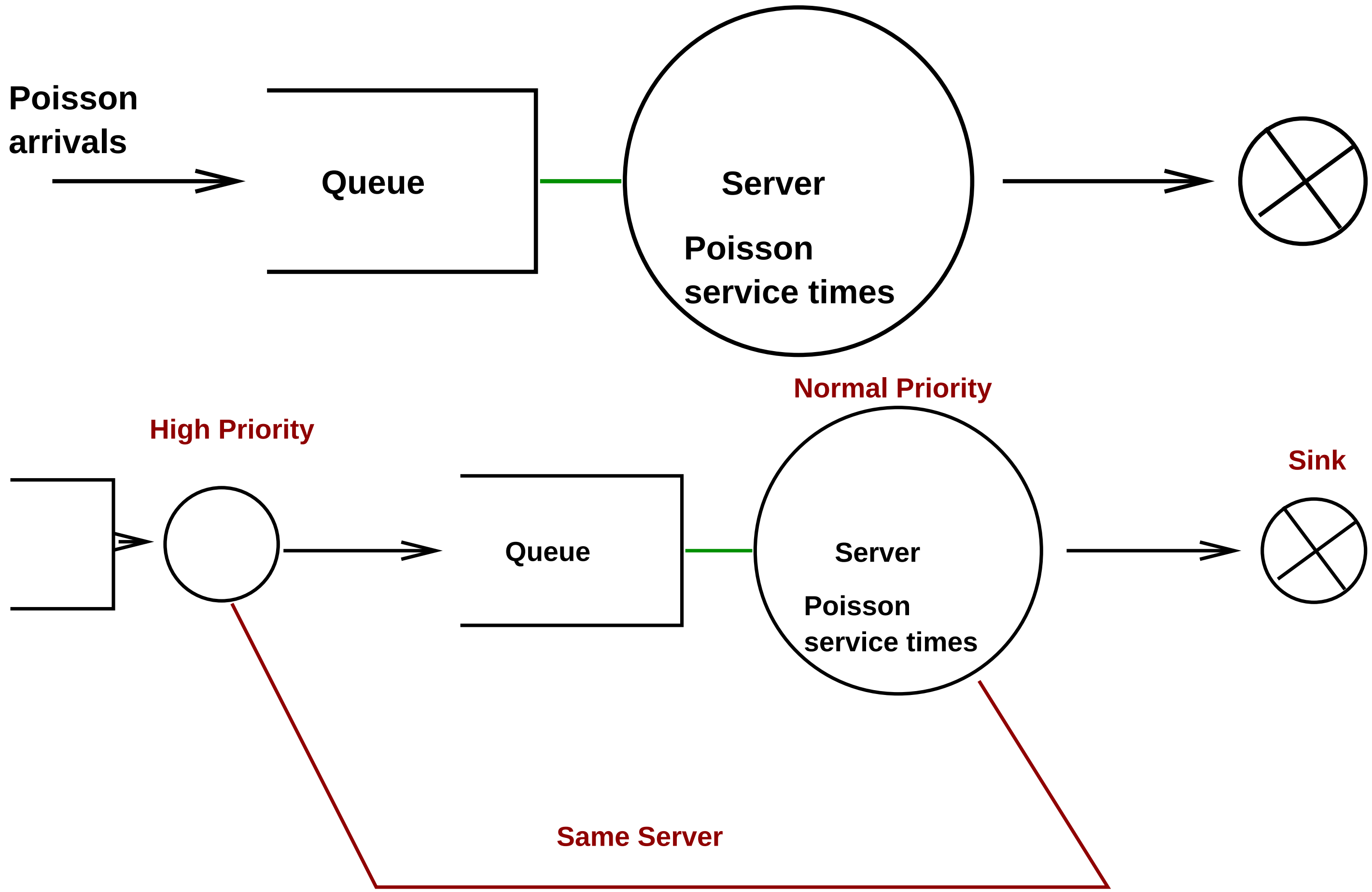
NICTA

Queueing Models:



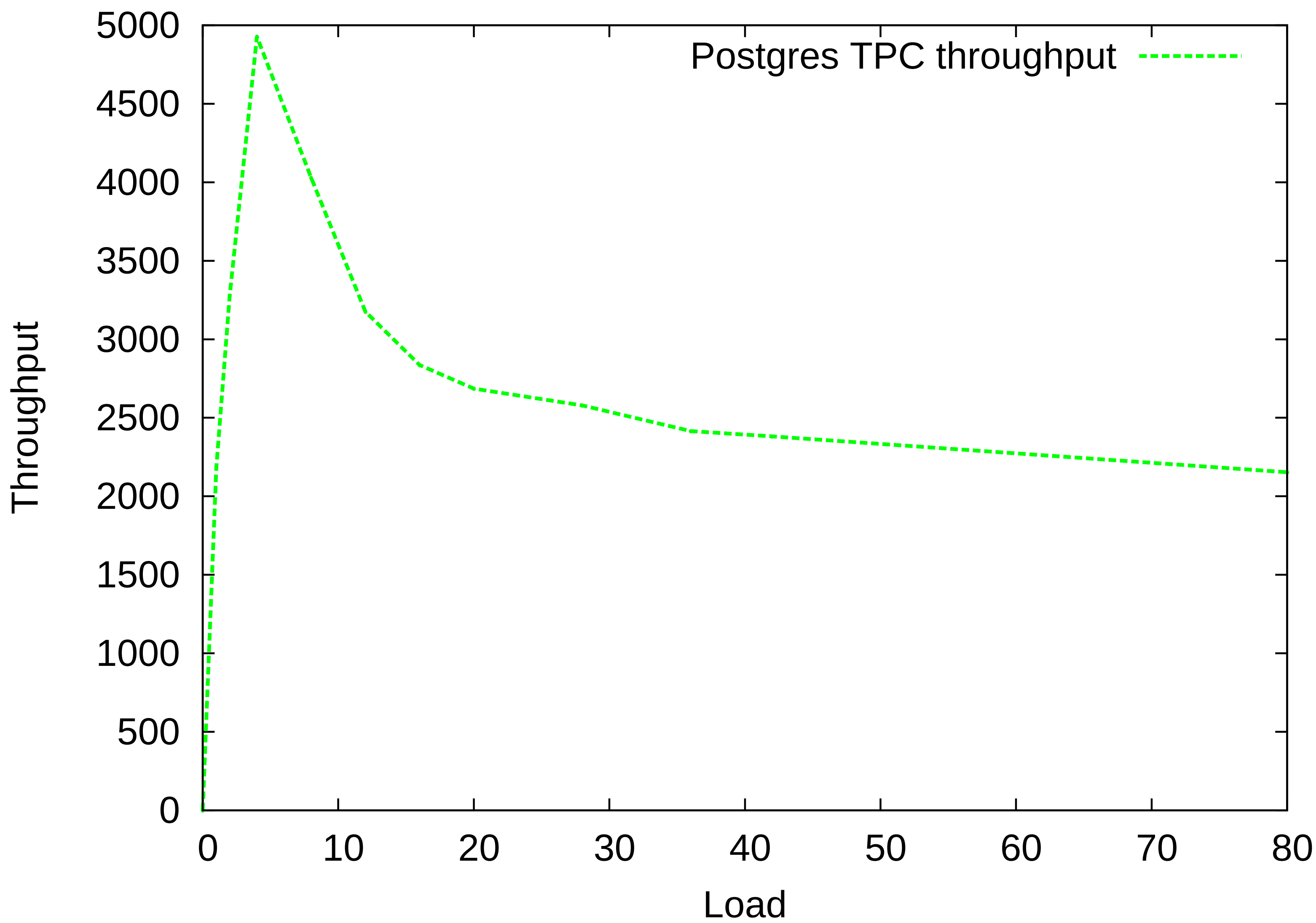
NICTA

Queueing Models:

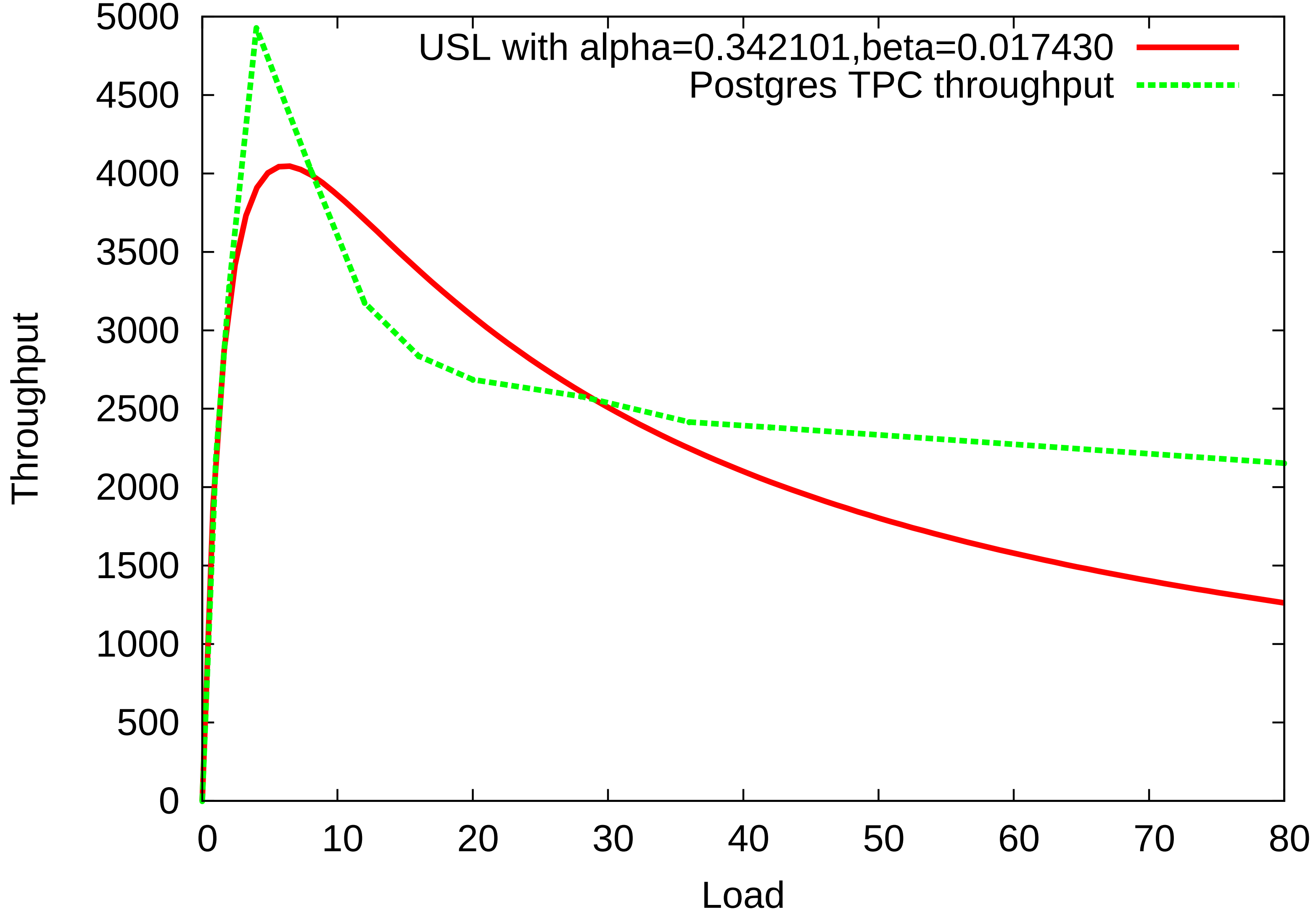


SCALABILITY

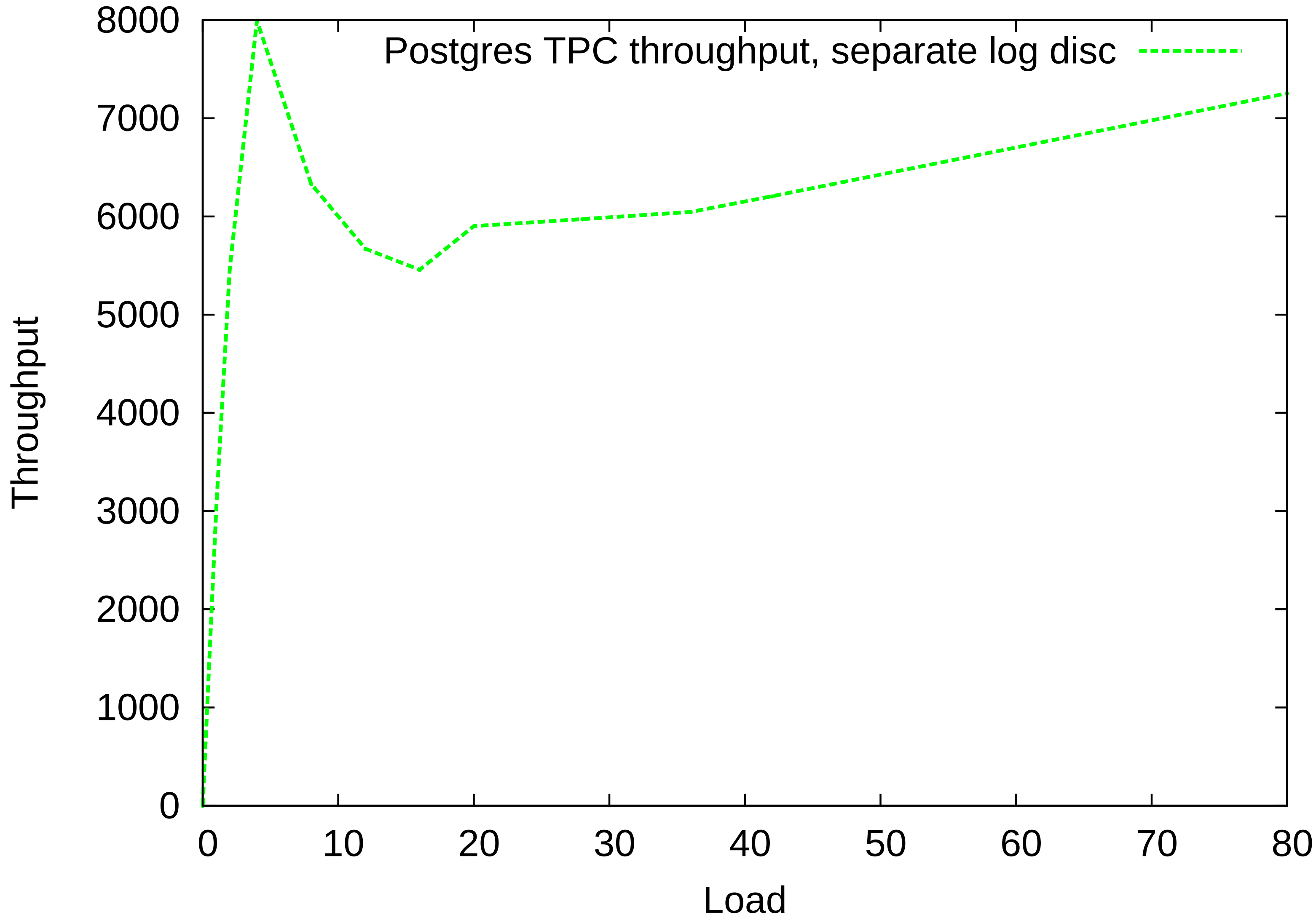
Real examples:



SCALABILITY

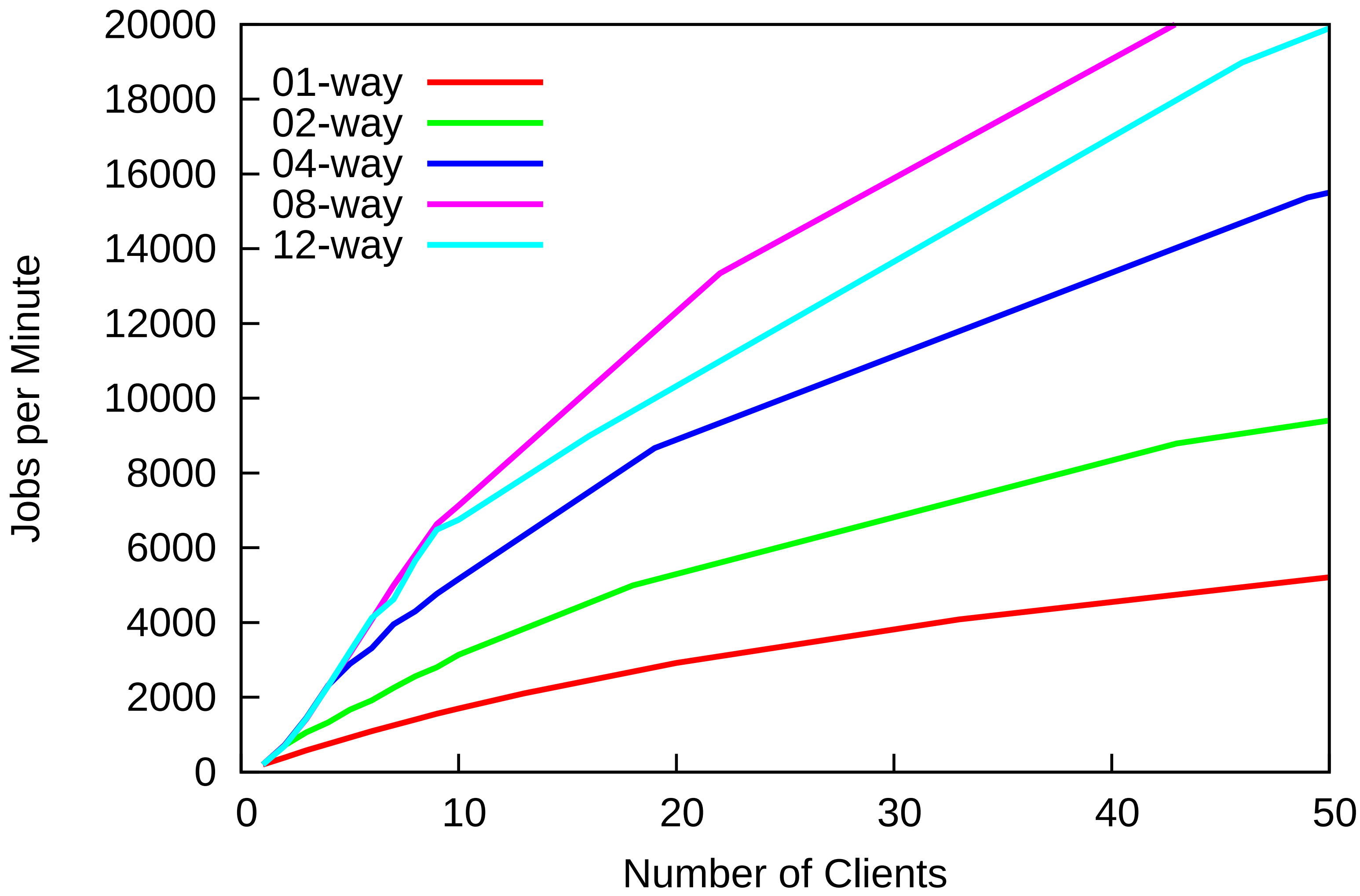


SCALABILITY



SCALABILITY

Another example:



SCALABILITY

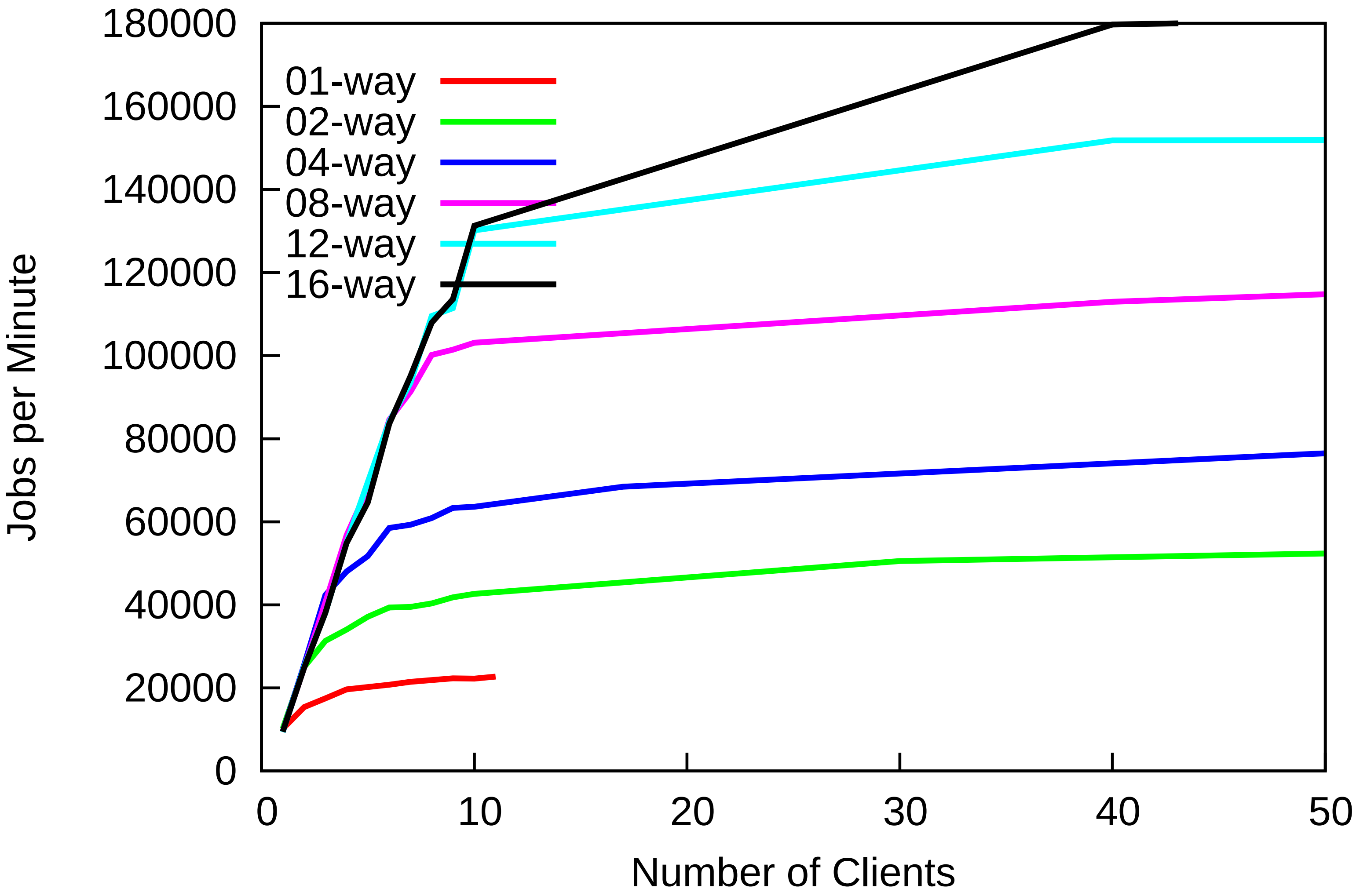
```
SPINLOCKS         HOLD          WAIT
UTIL   CON   MEAN( MAX )   MEAN( MAX )(% CPU)     TOTAL NOWAIT SPIN RJECT  NAME

72.3% 13.1% 0.5us(9.5us)   29us( 20ms)(42.5%)  50542055 86.9% 13.1%  0%
find_lock_page+0x30

0.01% 85.3% 1.7us(6.2us)   46us(4016us)(0.01%)      1113 14.7% 85.3%  0%
find_lock_page+0x130
```

SCALABILITY

```
struct page *find_lock_page(struct address_space *mapping,
                                unsigned long offset)
{
        struct page *page;
        spin_lock_irq(&mapping->tree_lock);
repeat:
        page = radix_tree_lookup(&mapping>page_tree, offset);
        if (page) {
                page_cache_get(page);
                if (TestSetPageLocked(page)) {
                        spin_unlock_irq(&mapping->tree_lock);
                        lock_page(page);
                        spin_lock_irq(&mapping->tree_lock);
```

. . .

SCALABILITY



TACKLING SCALABILITY PROBLEMS

- Find the bottleneck

TACKLING SCALABILITY PROBLEMS

- Find the bottleneck
  - not always easy

TACKLING SCALABILITY PROBLEMS

- Find the bottleneck
- fix or work around it

Tackling scalability problems

- Find the bottleneck
- fix or work around it
  - not always easy

TACKLING SCALABILITY PROBLEMS

- Find the bottleneck
- fix or work around it
- check performance doesn't suffer too much on the low end.

TACKLING SCALABILITY PROBLEMS

- Find the bottleneck
- fix or work around it
- check performance doesn't suffer too much on the low end.
- Experiment with different algorithms, parameters

Tackling scalability problems

NICTA

- Each solved problem uncovers another
- Fixing performance for one workload can worsen another

TACKLING SCALABILITY PROBLEMS

- Each solved problem uncovers another
- Fixing performance for one workload can worsen another
- Performance problems can make you cry

DOING WITHOUT LOCKS

Avoiding Serialisation:

- *Lock-free* algorithms
- Allow safe concurrent access *without excessive serialisation*

DOING WITHOUT LOCKS

Avoiding Serialisation:

- *Lock-free* algorithms
- Allow safe concurrent access *without excessive serialisation*
- Many techniques. We cover:
  - Sequence locks
  - Read-Copy-Update (RCU)

DOING WITHOUT LOCKS

Sequence locks:

- Readers don't lock
- Writers serialised.

DOING WITHOUT LOCKS

Reader:

```
volatile seq;
do {
    do {
        lastseq = seq;
    } while (lastseq & 1);
    rmb();
    ....
} while (lastseq != seq);
```

DOING WITHOUT LOCKS

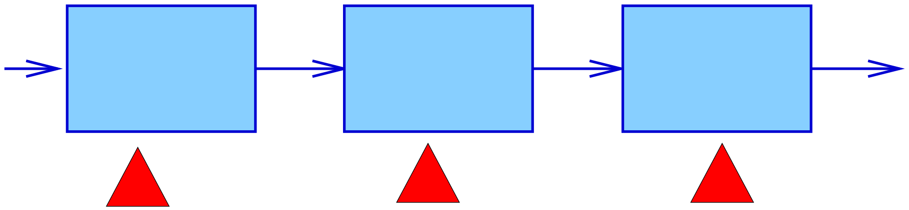
Writer:

```
spinlock(&lck);
seq++; wmb()
 ...
wmb(); seq++;
spinunlock(&lck);
```

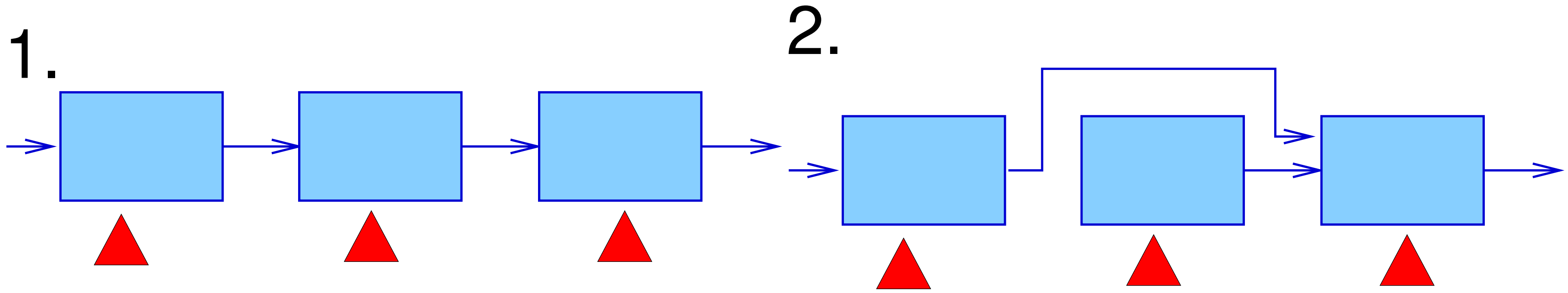
DOING WITHOUT LOCKS

RCU: ??

1.

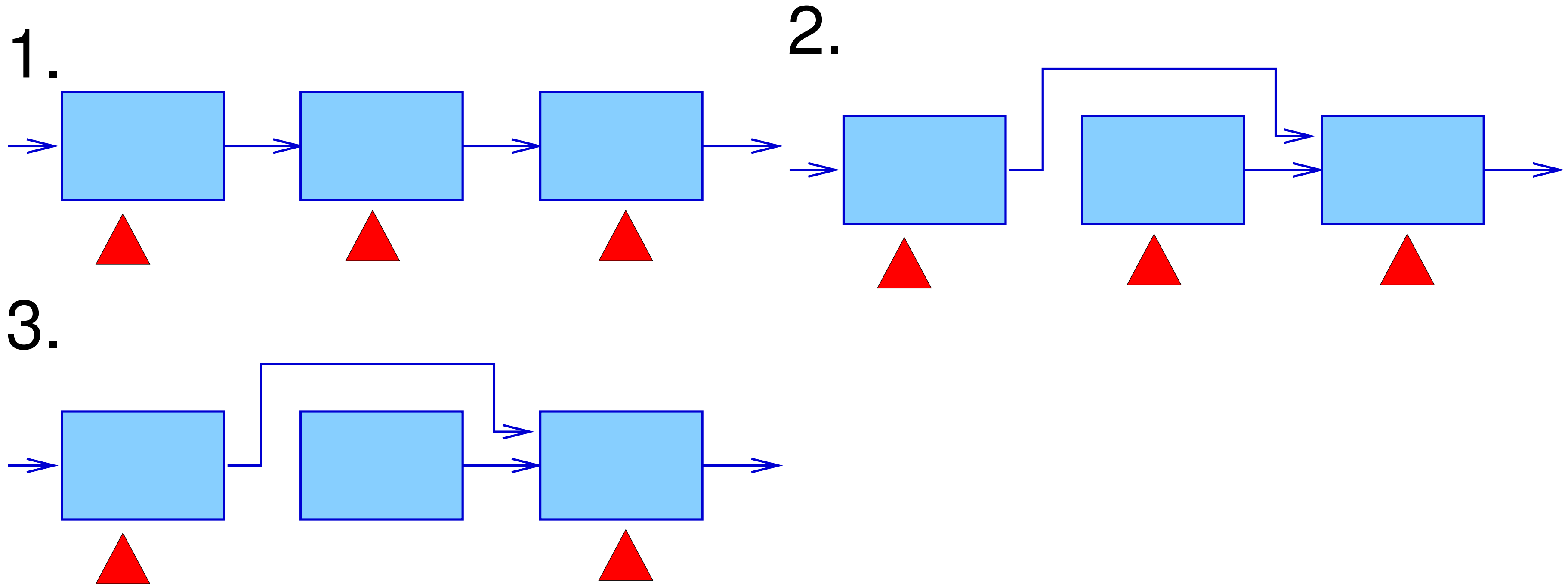


RCU: **??**



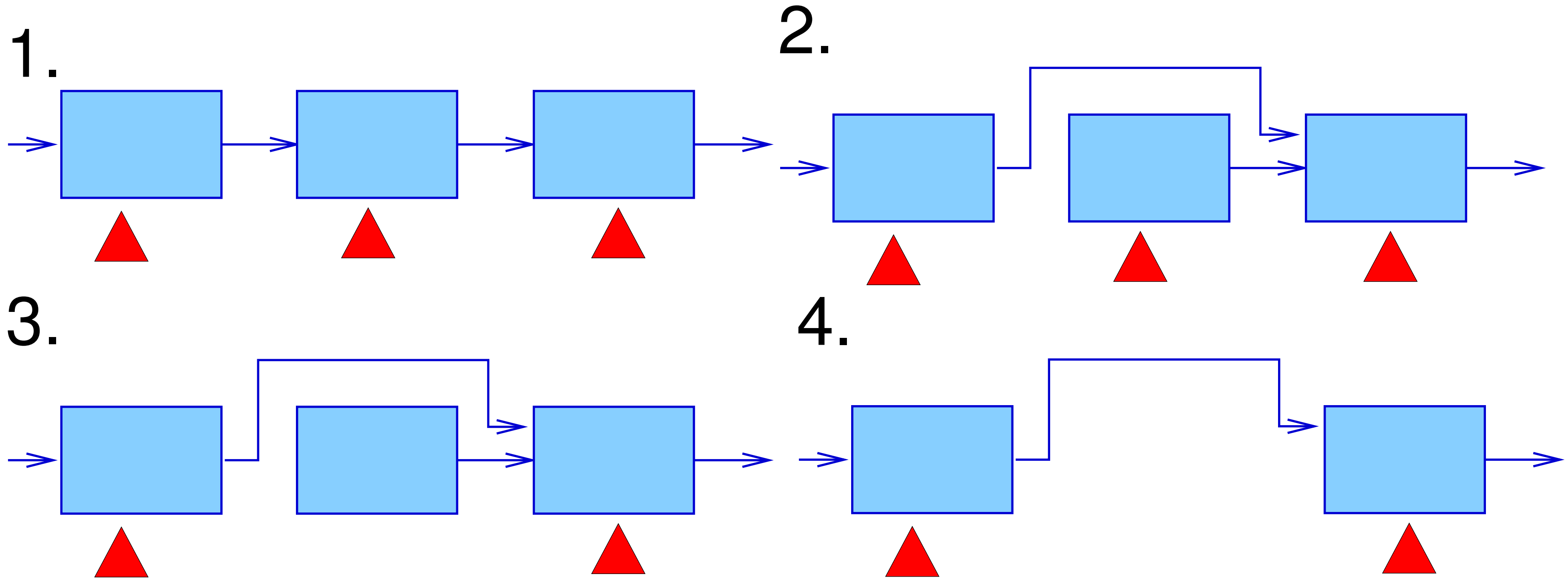
DOING WITHOUT LOCKS

RCU: **??**



DOING WITHOUT LOCKS

RCU: **??**



NICTA

References

McKenney, P. E. (2004), Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels, PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University.
URL:
http://www.rdrop.com/users/paulmck/RCU/RCUd

McKenney, P. E., Sarma, D., Arcangelli, A., Kleen, A., Krieger, O. & Russell, R. (2002), Read copy update, *in*

BACKGROUND READING

NICTA

'Ottawa Linux Symp.'.

URL:

http://www.rdrop.com/users/paulmck/rclock/r